Type Inference in a Declarationless, Object-Oriented Language

Eric J. Holstege

Computer Science Department
California Institute of Technology

5035:TR:82

CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science Department

Technical Report #5035

Type Inference in a Declarationless, Object–Oriented Language

by

Eric J. Holstege

In partial Fulfillment of the Requirements for the

Degree of Master of Science

June, 1982

The research described in this report was sponsored by the Defense Advanced Research Project, ARPA, Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597

ABSTRACT

In recent years, two developments in the design of programming languages have yielded significant improvements in a number of areas from the standard FBAPP programming model. These are the object-oriented paradigm, and variable polymorphism.

The object-oriented programming model allows the specification, hence restriction of the operations allowed on a data structure, something not possible with the more traditional PASCAL-style record structuring. This ability to encapsulate data from the outside world gives a greater security and error avoidance in very large software projects involving many programmers.

In addition, the object-oriented style is conceptually easy to program in, providing a useful framework for the subdivision of large problems into manageable pieces. This property is essential for the rapid and reliable implementation of large software systems.

Variable polymorphism refers to the ability of variables to change types at runtime. This is in contradistinction to typelessness (as in BLISS) where variables have no types associated with them. In most common languages, the programmer must declare the types of all the variables he uses; these types are then static throughout the execution of the program. Declarations allow the compiler to produce efficient code and to identify errors whose detection must otherwise be deferred until runtime; however, they sacrifices a good deal of the generality which is possible with less stringent variable binding schemes. On the other hand, languages which don't require declarations, and which allow variables to change types, such as SNOBOL and LISP, provide this generality by virtue of their extremely late binding, but thereby sacrifice efficiency.

SMALLTALK is perhaps the purest language which embodies both object-orientedness and declarationlessness. Unfortunately, these two features, while of great benefit in increasing programmer productivity and program reliability, suffer heavily from the point of view of runtime efficiency.

The project is to investigate ways to obtain the undeniable advantages of polymorphism and object-orientedness, without sacrificing runtime efficiency. More specifically, The goal is to build a compiler for a dialect of SMALLTALK for the VAX under UNIX (Berkeley 4.1bsd), which incorporates data-flow type inference algorithms enabling it to produce executable programs of an efficiency comparable to that of programs produced by compilers for more traditional but less powerful languages.

The optimization methods are described, test results are examined, and indications of future directions are given.

Contents

CHAPTER 1

INTRODUCTION

1.1. General Description of Problem and Goals

In the development of general purpose high-level programming languages, a number of advances can be pointed to as being of particular significance. Perhaps the most noticeable is the set of concepts embodied under the phrase "structured programming", which is now acknowledged to be a methodology which increases programmer productivity and program reliability.

In recent years the notion has surfaced that the next major advance in the evolution of general purpose programming languages is the object-oriented methodology. Languages such as SIMULA[1] [Birtwistle et al., 1973] and MAINSAIL [Wilcox et al., 1979] are based on the object-oriented paradigm. In addition, the new Department of Defense language ADA [Ledgard, 1980] [DARPA, 1981] has significant support for an object-oriented programming style. INTEL has recognized the new trend in its hardware; the iAPX 432 [INTEL, 1981], INTEL's most advanced microprocessor, is designed around the concept of object-oriented programming.

The reason for this growing interest is that the object is a good means of expressing not only the data abstraction available with conventional PASCAL [Jensen and Wirth, 1974] style records, but also a data encapsulation giving greater control over the manipulation of data. In PASCAL and similar

[1]SIMULA was originally designed to be a simulation language, but at Caltech it is used instead as a general-purpose object-oriented language.

languages, the record can specify the organization of data, but cannot control access to it. An object-oriented language class specifies the organization of data and also the operations which are to be allowed on it.

This protection of data from arbitrary corruption gives a greater security and error avoidance in very large software projects involving many programmers. In addition, it has been found at Caltech from the heavy use of SIMULA, and more recently MAINSAIL, that the object-oriented style is conceptually easy to program in, providing a useful framework for the subdivision of large problems into manageable pieces. This property is essential for the rapid and reliable implementation of large software systems.

The second major property to be examined is variable polymorphism, and concommitant declarationlessness. In most widely used general purpose languages, the programmer must declare the types of the variables he uses. These variables then remain of the specified type throughout the execution of the program. This allows the compiler to produce efficient code and to identify errors whose detection must otherwise be deferred until runtime; however, it sacrifices a good deal of the generality which is possible with less stringent variable binding schemes. On the other hand, polymorphic languages (which therefore don't require declarations), such as LISP and SNOBOL, gain considerable generality by virtue of the ability of program variables to change their types at runtime. They achieve this by extremely late binding, but thereby sacrifice efficiency.

SMALLTALK[2] [Byte, 1981] is an example of the latter category. It is a polymorphic object-oriented language developed at XEROX Palo Alto

[2]SMALLTALK here refers to Smalltalk 80.

Research Center. Variables can be of any type, or "class", and can change types merely by appearing on the left side of an assignment statement. Associated with each class is a set of operations, or "messages" to which variables of that class can respond, and the "methods" that go with these messages. A message is analogous to a function name in PASCAL; a method to a function definition. In PASCAL these are one-to-one, but in SMALLTALK all messages are potentially generic, i.e. the mapping from messages to methods can be one-to-many.

As an example, the expression $a+b$ is metaphorically interpreted to mean "send the message + with argument b to the object a (called the "receiver"). Which routine is actually invoked to do the + depends on the class of a at the time of execution. Thus, for every operation, a test must be made as to the class of the receiver, and whether that class actually knows how to do the operation. Things are further complicated by the fact that SMALLTALK has a subclassing mechanism equivalent to that of SIMULA, whereby each class has a superclass and if a message is not understood by an object of some class, its hierarchy of superclasses is searched.

The problem is that runtime variable type changes are desirable when the concern is generality, and extremely undesirable when efficiency is of primary importance. The goal is to have our cake and eat it; to gain the efficiency when the generality isn't being used, but to have the generality available when we need it. Specifically, the objective is to implement an object-oriented language featuring no declarations and runtime variable polymorphism in the freest possible way, by assignment, but to do so in such a way that efficiency is only sacrificed when necessary.

1.2. Vehicle for Examining the Problem

SMALLTALK provides the features needed to examine the problem, but unfortunately it also contains some totally extraneous ones which serve to obscure the heart of the issue. For that reason, the language used as a basis for implementing the optimization methods described later is based on SMALLTALK, but is a good deal simpler. Those features incidental to the problem have been thrown out or modified. The language, call it TINYTALK, is a polymorphic, object-oriented language using the SMALLTALK message sending metaphor as its procedural and expressional form, but with a different form for control structures. It is important to note that TINYTALK is a research tool and not a full-fledged general purpose language[3].

1.3. How to Have Our Cake and Eat It

The problem boils down to how to avoid runtime type testing without having the programmer have to tell the compiler the answers. The solution which springs to mind is to make the compiler smart enough to work out the types of variables at various points just from the program itself. For example, if a is set equal to *2* somewhere, a is an integer until it gets changed. Then a subsequent $a+b$ need not result in a runtime test of the type of a; the compiler can tell it is integer and that that the form of + needed is the one associated with class *Integer*[4]. This is a simple example of the sort of type inference of which TINYTALK is capable. The ideal result of applying such type inference procedures to a program is that the only variables which will have runtime type tests performed on them are those

[3]In particular, the runtime support at present has no input facilities; these not being needed to examine the effectiveness of the type inference optimizations.

[4]Class names are traditionally written with the first letter capitalized, whereas normal variable names are entirely lower case.

that require it by the fact that they really can, in the normal execution of the program. be one of a number of types at the same place at different times.

1.4. Scope of the Project

The project consists of the design and implementation of a TINYTALK compiler for the VAX-11/780[5] running Berkeley UNIX[6] 4.1bsd, the design and implementation of the algorithms necessary to do the type inference alluded to above, and the testing of the resulting system to determine its effectiveness.

1.5. Previous Work

There are basically two general methods of type inference [TEMPO]. The first, the functional approach, is used mainly for languages like LISP, where the metaphor is function application. The second, data-flow, is used with imperative languages like PASCAL. TINYTALK is a cross between an applicative and an imperative language. It has the imperative control flow structuring, but the message metaphor is basically applicative, with syntactic trickery to make it look like normal expressions.

The functional approach has the feature that the possible types which a function can return are expressed as a type expression in which the types of the parameters appear as free variables. This avoids the possibility of it being necessary to re-analyze a function for each invocation. With data-flow approaches, procedures which can take parameters of varying types, as can all TINYTALK messages, may require re-analysis. In the TINYTALK compiler, message return types are also represented as type expressions with the

[5]VAX is a trademark of Digital Equipment Corporation.
[6]UNIX is a trademark of Bell Laboratories.

parameter types as free variables, making re-analysis unnecessary. Other than this, the approach taken in TINYTALK is data-flow.

The type inference algorithms developed and implemented in this project are based primarily on the concepts and ideas presented in [TEMPO] with reference to type inference in the language TEMPO. This work differs, however, in a number of respects. First, the algorithms presented here must deal with several kinds of variables: local, global, field, argument, and so on, whereas the algorithm used on TEMPO deals essentially only with local variables, the easiest of those encountered in TINYTALK.

Second, the objective in TEMPO is to minimize not only runtime type testing, but also the need for any runtime type tag at all wherever possible. Since the structure of an object-oriented language such as TINYTALK requires the existence of a runtime type tag for other reasons anyway, TEMPO's second goal is not germane here. This implies modification of the algorithms used in TEMPO.

Third, and more notably, [TEMPO] gives no information on type inference methods for TEMPO procedures, and, since TINYTALK is heavily procedural, dealing with procedures is essential.

CHAPTER 2

DESCRIPTION OF TINYTALK

2.1. General Features

TINYTALK features a message sending syntax and semantics virtually identical to that of SMALLTALK. It has a builtin *if-then-else*, control construct, and *while*, and *for* looping constructs. Classes can be defined, messages defined in these classes, and methods associated with these messages. Superclassing is as in SMALLTALK. The main program has variables accessible only within its body (these are called globals). Instances of classes have field variables as in SMALLTALK. Methods have arguments and local variables as in SMALLTALK.

2.2. Detailed Description

In the following description of TINYTALK, some familiarity with SMALLTALK is assumed.

2.2.1. Program Structure

A program consists of an optional sequence of class definitions and class enhancements, followed by a main program. Each class definition creates a new class of object. It specifies the class's superclass, the data attributes (called fields) of instances of the class, the messages to which instances will respond, and the actual operation (method) to be performed when responding. A class enhancement adds a message to an already defined class.

2.2.2. Class Definition and Enhancement

A class definition is of the form:

```
class <classname>
[ subclassof <classname> ]
[ fields <idlist> ]
begin
<message definition> { <message definition> }
end
```

where [*x*] indicates 0 or 1 occurrences of *x* and { *x* } indicates 0 or more occurrences of <x>.

A class enhancement allows the addition of a message and associated method to an existing (usually builtin) class. It is of the form:

```
class <classname>
understands
<message definition>
```

The <classname> must be the name of an existing class.

2.2.3. Message Definition

A message definition is used to connect a message with a particular class; objects which are instances of that class will then "understand" the message, and the associated method will be invoked. There are three types of messages: unary messages, binary messages, and keyword messages. A unary message definition has the form:

```
message <message name> [ | <local variable list> ]
<block>
```

The <local variable list> is a list of the names of the local variables of this method. It is optional, and if present, is preceded by the character '|'. A binary message definition has the form:

message <operator> <formal parameter> [| <local variable list>]
<block>

See appendix A for the exact definition of an <operator>; basically it is a sequence of one or more characters from the set { !$%&*+,-/:;<=>?@\^_`|~ }, with certain sequences reserved. The <formal parameter> is just an identifier. For example,

message ++ n | a b
begin
...
end

defines a method associated with the message ++ having formal parameter *n* and local variables *a* and *b*. A keyword message definition has the form:

message <message name$_1$>: <formal$_1$> <message name$_2$>: <formal$_2$> ...
[| <local variable list>]
<block>

For example:

message at: index put: value | x y
begin
...
end

In this example, the message name is *at:put:*, the formal parameters are *index*, and *value*, and *x* and *y* are local variables.

In all three forms, <block> is the method body; just as in PASCAL a block forms the body of a procedure.

2.2.4. The Block

In TINYTALK, a block is just a sequence of statements separated by '.' and enclosed by **begin** and **end**, or in parentheses. It is thus entirely analogous to a PASCAL block, with one major difference. A TINYTALK block yields a value; the value of the last statement in it. All other statements in the block are "voided", that is, their value is thrown away. Thus, for example

(a←"hello". 3)

has the value 3, and as a side effect sets *a* to the string *"hello"*. As another example,

(a. b.)

has no value at all, because the last statement in the block is the null statement. If such a block were used in a context where a value was needed (e.g. on the right hand side of an assignment), a compile time error would result. **All parenthesized expressions are blocks.** They contain one statement. For example, the block *(a+b)* contains as its statement list a single statement and hence has as its value the value of that statement, namely *a+b*.

2.2.5. Statements

2.2.5.1. The Unit

The most common type of statement in TINYTALK is the unit. A unit is analogous to an assignment statement, procedure call, or expression in PASCAL. The basic form is:

{ <id> ← } <term> [<keyword selector> { ; <keyword selector> }]

The ← is the assignment operator. A term is:

<factor> { <binary selector> }

A factor is:

```
<primary> { <unary selector> }
```

A primary is an integer, real, character, or string constant, a variable, a predefined constant, *self*, **or a block**. Predefined constants include *true*, *false*, and *nil*. The pseudo-variable *self* refers to the receiver of a message inside the method activated upon receipt.

Working backwards, a unary selector is just the name of a unary message. Then, an example of a factor is

```
x negate invert
```

where *x* is the receiver of the unary message *negate*, and the result is then to receive the unary message *invert*.

A binary selector is analogous to the invocation of an operator in ALGOL 68 or PASCAL (except that in PASCAL there are no user defined operators). It has the form:

```
<binary message> <factor>
```

Thus, for example:

```
+ y negate
```

is a binary selector involving sending the message + with argument the value of the factor *y negate* to some receiver. An example of a term, then is

```
2+3*4
```

in which *2* is the receiver of the binary message + with argument *3*, and the result subsequently receives the message * with argument *4*. Note that there is therefore no notion of priority in binary operators; all are the same priority and evaluation is left-to-right.

A keyword message is analogous to a procedure call in PASCAL. It has the form:

```
keyword: <term> { keyword: <term> }
```

For example,

```
at: 2 put: x + y
```

sends the message *at:put:* with arguments *2* and *x+y* to some receiver.

The most general right hand side of an assignment has the form:

```
<term> [ <keyword selector> { ; <keyword selector> } ]
```

Here, <term> is evaluated, and then the messages indicated by the several <keyword selector>s are sequentially sent to the result, with the specified arguments. Note that <term> is evaluated only once. As a complete example,

```
q ← r ← x << 4 at: 1 put: z; at: 2 put: 14 * q; at: 3 put: m negate
```

means: evaluate the term *x << 4*, (which is the binary message << with argument *4* being sent to the receiver *x*), and save it in a secret place. Then send the keyword message *at:put:* with arguments *z* and *2* to the secret place, and throw away the return result of the message. Next, send *at:put:* again to the secret place, but this time with arguments *2* and *14 * q*, and again discard the result. After that, once again send *at:put:* with arguments this time of *3* and *m negate*. Finally, take the return result of this last message sent and assign it to *q* and to *r*.

The value yielded by a unit is the result returned by the message in the last keyword selector, if any, or the value of the term. In the above example, it is *x<<4 at: 3put: m negate*; this is also the value assigned to *q* and *r*.

2.2.5.2. Primaries

As mentioned in the previous section, primaries include integer, real, string, and character constants; the builtin constants *true*, *false* and *nil*; *self*; blocks; and variables.

2.2.5.2.1. Constants

Integer and real constants are standard. A character constant is a single character or metacharacter in single quotes. A string constant is a sequence of characters or metacharacters in double quotes. A metacharacter is a two character sequences, the first of which is a backprime or backslash. These allow the specification of non-printing characters within a character or string constant. See appendix A.

The builtin constants are all of class object; *true* and *false* are returned as the results of boolean messages and *nil* is the initial value of variables.

2.2.5.2.2. Variables

In TINYTALK (as in most languages) the unit of data is a variable. A variable can be one of several modes:

local

Local variables are variables declared local to a method, and go away when the method returns. They are entirely analogous to local variables in PASCAL except that they are initialized to *nil*.

global

Global variables are, in this implementation, accessible only from the main program. They are illegal in methods (but see section 6.5). They are also initialized to *nil*. A Global variable is created merely by using it in the body of the main program.

field

Field variables represent the fields of an object. A field variable used inside a method refers to the corresponding field of the receiver. Field variables are analogous to SIMULA data attributes. When a new object is created, its fields are initialized to *nil*. Field variables are illegal in the main program.

class

A class is the name of one of the user defined or predefined classes. It is an instance of class *Class*, and is really a kind of constant.

argument

An argument variable represents the value, within a method, of an actual parameter. It is illegal in the main program. Arguments are passed by value, which means that although the argument can be assigned to in the method, the value in the caller remains unchanged; however, the argument's fields can be changed by the callee, since they are variables in their own right.

self

The variable *self* appearing in a method refers to the receiver. It is treated the same way as an argument.

2.2.5.3. The IF Statement

The **if** statement is exactly as in PASCAL:

if <unit> **then** <statement 1> [**else** <statement 2>]

The unit is evaluated and if it yields the predefined constant *true* then statement 1 is executed; otherwise if the **else** part exists, statement 2 is executed.

The value of an **if** statement is the value of whichever of statement 1 or statement 2 is executed. If there is no **else** part, the value of an **if** statement is the predefined constant *nil*. Thus, for example,

```
a ← (if x==2 then 8 else (x←y. 6))
```

will set *a* equal to *8* if *x* does equal *2*, and set *a* equal to *6* with the side effect of setting *x* equal to *y* if *x* doesn't equal *2*. The **else** part:

```
( x ← y. 6)
```

is a statement which is a unit, which is a term, which is a factor, which is a primary, which is a block, which is a statement list and has the value of the last statement in it, which is a unit; a term; a factor; a primary; the integer constant *6*.

2.2.5.4. The WHILE Statement

The **while** statement is again standard:

```
while <unit> do <statement>
```

So long as the unit yields *true*, the statement is executed. The value yielded by a **while** statement is *nil*.

2.2.5.5. The FOR Statement

The **for** statement is exactly as in C [Kernighan and Ritchie, 1978], This was done with the intention of making timing tests between the two languages more easily interpreted. The syntax is:

```
for <unit 1> . <unit 2> . <unit 3> do <statement>
```

The semantics of such a **for** statement is defined to be equivalent to the corresponding **while** statement:

```
<unit 1>
while <unit 2> do begin
   <statement>
   <unit 3>
end
```

Thus a **for** statement also yields *nil* as its value. Any of the three units in the control portion of the **for** statement can be left out; the first and third default to nothing and the second to *true*.

2.2.5.6. The RETURN Statement

The **return** statement is used to return values from methods. It has the form:

```
return <unit>
```

or

```
^ <unit>
```

The method exits yielding the value of the unit. A method may also terminate by falling out the bottom; in this case it returns its receiver. That is, there is an implicit

```
return self
```

as the last statement of each method. The **return** statement is not legal in the main program.

2.3. Builtin Classes and Messages

The present TINYTALK runtime system has a number of builtin classes and a fairly minimal set of builtin messages which these classes understand. The predefined classes are *Object*, *Class*, *Array*, *Integer*, *Real*, *String*, *Char*.

2.3.1. Class *Object*

Class *Object* is the eventual superclass of all classes. If no **subclassof** clause appears in a class definition, the superclass defaults to *Object*. Class *Object* understands the messages == for equality testing, and <> for inequality testing. These test only whether the receiver and argument are the *same* object, not whether they are different objects with the same values for their fields. Note, however, that == and <> are also defined for *Integer*s, *Real*s, *Char*s, and *String*s, so as to compare the values, not the objects themselves. *Object*s understand the message ',' (comma) which forms an array of two elements from its receiver and argument. The message ',' is also understood by *Array*s to mean concatenate the argument onto the receiver.

2.3.2. Class *Class*

Class *Class* is an unusual one in that the only way to make new instances of it is to do so at compile time with class definitions. A class definition, for example

```
class Point
    ...
```

creates a variable *Point* which is an instance of class *Class*. Class *Class* understands two related messages: *new* and *new:to:*. These are used to create new instances of things. Thus the expression

```
    Point new
```

yields a new instance of class point with all fields initialized to *nil*. The message *new:to:* is reserved for *Arrays*; the expression

```
    Array new: 2 to: 10
```

makes a new array containing 9 elements numbered 2 through 10. It is a runtime error to send the message *new:to:* to any instance of class *Class* except *Array*. It is a runtime error to send the message *new* to *Class*, since this is tantamount to making a new class at runtime, which would make type inference, to say the least, difficult. (The hypothetical new class would be useless anyway, since there would be no way to associate any messages with it).

2.3.3. Class *Array*

All arrays are one dimensional, and can contain as their elements objects of any class, including other (or the same) arrays. Class *Array* understands messages to concatenate an element to an array, to access and set an element of an array, and to get the lower and upper bounds. The ',' (comma) message concatenates the argument to the receiver array, extending it by one element (its upper bound grows by one). The *lb* and *ub* unary messages return the lower and upper bound, respectively, of their receiver. The message *at:* returns the argument'th element of the receiver. The message *at:put:* sets the argument1'st element of the receiver to argument 2, and returns *self*.

2.3.4. Class *Integer*

Class *Integer* understands the usual binary messages: +, -, *, /, for addition, subtraction, multiplication, and division. These yield *Integers* if the argument is *Integer*, *Reals* if the argument is *Real*, otherwise a runtime error results. *Integers* also understand % (integer remainder), and the usual comparison operators ==, <>, <=, >=, <, > so long as their argument is *Integer* or *Real*. These all return *true* or *false*. The messages *asReal* and *asChar* can be used to convert to *Real* and *Char*; *asChar* returns a *Char* having the ascii value of its receiver. *Integers* know how to print themselves with the message *print*.

2.3.5. Class *Real*

Reals also understand the arithmetic and comparison operators, as well as the conversion message *asInteger*, which truncates. *Reals* know how to print themselves with *print*.

2.3.6. Class *Char*

Chars can be converted to *Integers* and *Strings* with *asInteger* and *asString*, and printed with *print*.

2.3.7. Class *String*

Substrings can be taken with *from:to:* which should be given *Integer* arguments. Copies can be made of *Strings* using *copy*. The length can be found with *length*. The message *at:* returns as a *Char* the argument'th element of a *String*. The message *at:put:* can set an element of a *String* to a particular *Char*. *Strings* can be printed with *print*. The message *print:* uses its receiver as a printf[1] -style format string and prints its argument, which should be an *Integer*, *Real*, *Char*, *String*, or an array of these. Thus, for example, if *a* equals 31,

    "The time is %d:%d.%d'n" print: (12,a,19)

will print

    The time is 12:31.19

onto the standard output. Here, *print:* is being given as its argument the array *(12,a,19)* formed with the comma message described above under class *Object* and class *Array*.

2.4. Comparison to Smalltalk 80

TINYTALK is quite simple in comparison to SMALLTALK 80. One immediately notes the absence of global variables accessible inside messages, and of class variables. See however section 6.5.

Another major difference is the lack of the SMALLTALK "block" (entirely different from the syntactic entity described in section 2.2.4). In SMALLTALK, the block forms the basis of all control structures. Blocks are instances of class *Block*. They are basically anonymous methods, or routine texts to use ALGOL 68 terminology. They can be executed by being sent the message *value*. Blocks with parameters exist; they are sent one of the messages *value:*, *value:value:*, ... depending on the number of parameters. Blocks can access and modify local variables. They can be nested, and inner blocks can access and modify the parameters of outer ones.

[1]*Printf* is a UNIX routine used to produce formatted output; its first argument is a string specifying the format [UNIX].

The concept of block as described above can cause considerable implementation headache. As an example, consider the SMALLTALK fragment which represents a definition of the message *mess:* which contains the SMALLTALK version of an **if** construct:

```
mess: x | a b
(
   (a = b) ifTrue: [ ^ 12 ] ifFalse: [ ^ 15 ]
)
```

(In SMALLTALK the = is the comparison operator). What this means is that *(a = b)* is evaluated, yielding one of the constants *true* or *false*, which are instances of class *Object*. This result is then sent the message *ifTrue:ifFalse:* with two blocks as arguments. This message examines its receiver, and if it is *true*, sends the message *value* to its first argument, otherwise *value* is sent to its second argument. Now ask the question: what does the return in [^*12*] or [^*15*] return from? Not from the block itself, which after all is an anonymous method. Not from the *ifTrue:ifFalse:* message, that would be useless. It should return, in this case, from *mess:*, the statically enclosing method.

However, since a block is an object, it can be assigned to a variable; that variable can be returned to a calling method; it can then be executed there. Consider another example. Supposing some method contains the line:

```
^ [ ^ 3 ]
```

The caller gets returned an object of class *Block* whose code represents the operation "^ *3*". The caller is, of course, entitled to send to this object the message *value*, which results in the execution of ^ *3*. However, now the return cannot be made from the statically enclosing method since it has already gone away. Where is the return from? Not necessarily the method who sent the *value* message, since this might be an *ifTrue:ifFalse*; nor the next non-anonymous dynamically enclosing method, for the same reason[2].

Now consider the statement

```
^[ x ← 2 ]
```

where x is a local variable of the method. The caller gets returned a block, to which he is once again entitled to send the message *value*. This results is the statement $x \leftarrow 2$ being executed, but where is x? It is a local variable of a method which is done executing; whose stack frame is gone.

One solution to the problem of referencing no-longer-existing locals is to keep activation records on the heap instead of in stack frames[3], and to make blocks contain pointers to the activation records of any locals they reference. Note that since blocks can themselves have parameters (essentially initialized locals), inner blocks can access these as well. Thus blocks must potentially keep pointers to several activation records. This scheme means that activation records won't be deleted until no blocks need them.

Needless to say, this whole design is an expensive one to use for a construct that forms the basis of all the control structures in the language, and was deemed not worth implementing in TINYTALK given the primary goal of enhancing runtime efficiency. More significantly from the point of view of the goals of this project, the intent of blocks is primarily to enable the user to create his own control structures, and user definable control structures make data flow analysis, at the very least, rather difficult. The other primary

[2]If anyone knows the answer as defined in Smalltalk 80, I'd be interested to know.

[3]Smalltalk 76 does this for other reasons, namely that activation records, called "contexts" are also objects.

decrease in flexibility incurred by not putting activation records on the heap is that coroutines cannot be implemented with the stack model used in TINYTALK.

One final note of difference is the idea of the recursive definition of a unit; specifically the ability of a primary to be a block, allowing arbitrary nesting of statements. This deliberately eliminates the difference between a statement and an expression; an idea taken directly from ALGOL 68 [Lindsey and van der Meulen, 1980] and, I think, an important one. From the point of view of the implementation of a data flow type inference algorithm, however, it causes difficulties. In TEMPO, statements are the elements of analysis; this is not practical in TINYTALK because statements can contain other statements in parenthesized subcomponents, to an arbitrary depth of nesting. The solution to this dilemma is discussed in section 4.2.

2.5. Comparison to Simula

The polymorphic nature of TINYTALK (and of SMALLTALK) gives it a significant conceptual advantage over SIMULA, which is of the same basic metaphor, but which requires that variables be declared and that they remain of one type throughout execution.

Variable polymorphism gives, in particular, considerably more freedom in the creation of data structures. In SIMULA, if it is desired that a field (data attribute) be able to contain data of one of several types, a class must be created which is a superclass of all of those types, and the field must be declared to be of that class. This is the only reason for the existence of class *Thing* in Caltech SIMULA culture. Everything is a subclass of class *Thing*. Then the programmer must play around with VIRTUAL declarations and QUA to say what he means.

In TINYTALK, if a field must be of one of several types, there is no problem; you just put whatever you want there. TINYTALK does not need the concept of VIRTUAL, or the QUA operator, because this comes free with the polymorphic nature of the variables.

CHAPTER 3

RUNTIME REPRESENTATION

3.1. Target Machine

The target machine for the TINYTALK compiler is the VAX 11/780 running Berkeley UNIX 4.1bsd. The compiler produces as output an assembly language program which is then assembled using the standard UNIX assembler, *as*[1], and linked with runtime libraries using the standard UNIX linker, *ld*.

3.2. Runtime Representation of Objects

An object at runtime is actually a data structure kept on the heap which has the following form (words are 32 bits):

word -2	reference count
word -1	pointer to class of which this is an instance
word 0	field 0
word 1	field 1
...	
word n	field n

Figure 3.1 - Format of an Object in Memory

For details of the meaning of the reference count, see section 3.6.

It is important to distinguish between variables and objects. An object is a block of memory on the heap or in static storage in the format of figure 3.1. A variable is a single word containing a pointer to word 0 of such an

[1]UNIX takes the viewpoint of why should each compiler produce object files directly, if that is what the job of the assembler is. The UNIX assembler is designed to be a back end for the compilers, and all of them use it.

object. The location of a variable depends on what kind it is; locals and arguments are on the stack; globals are in static storage; fields are on the heap along with the objects they point at.

3.3. Runtime Representation of Classes

Since a class is an object, it has the standard two words of reference count and class before it; the complete layout is:

word -2	reference count
word -1	pointer to class *Class*
word 0	pointer to message table
word 1	name of this class
word 2	number of fields in objects of this class

Figure 3.2 - Format of a Class in Memory

The name of the class is actually not a string, but a small integer which is an index into a string table. This representation is as in the XEROX implementation of Smalltalk 76, and the small integers are called ***unique strings***.

The format of a message table is:

word 0	length of table
word 1	message 1 unique string
word 2	method 1 start address
word 3	message 2 unique string
word 4	method 2 start address
...	...

Figure 3.3 - Format of a Message Table

The message table is actually a hash table with the unique string being the key variable in a hashing function. Thus any one of the unique string entries can be zero indicating an empty hash table entry.

3.4. Runtime Representation of Objects of Builtin Classes

Integers, reals, and chars are represented in the obvious way; they have one field, which contains the value. Strings have two fields, the length and a pointer to the text itself. Arrays have four fields, the lower and upper bounds, a pointer to the data, and an allocated space count. This last gives the size of the area allocated to the array. It is usually larger than *upper bound - lower bound* + 1, since Arrays are allocated and lengthened in chunks to speed up appending elements. Instances of class *Object* have no fields at all, just the reference count and class pointer.

3.5. Implementation of Methods and Messages

A method is just a VAX procedure, with arguments passed via the stack. The sequence of steps to send a message is as follows:

(1) Push the arguments onto the stack in reverse order.

(2) Push the receiver's class.

(3) Push the message unique string.

(4) Call the lookup routine, which uses the values put on the stack in (2) and (3) as arguments. This routine is part of the runtime system, and looks up the message in the hash table dictionary of the class. If not found, it looks in the superclass's message table, and continues this process until the message is found, or until class *Object* is searched with no success. If the message is found, the associated method start address is returned (see figure 3.3), otherwise a runtime error occurs.

(5) Push the receiver.

(6) Call the method returned by the lookup routine in step 4.

The objective of the type inference is to allow the compiler to avoid the necessity of the lookup (steps 2 through 4) and to call the correct method directly (step 6), or even, in the case of some builtin methods, to replace the entire sequence with an inline expansion of the method.

The prolog for a method consists merely of creating room on the stack for local variables and initializing them to *nil*. Methods return values in a register reserved for this purpose. The exact return sequence is:

(1) Decrement the reference count (and delete the object if zero) of all local variables.

(2) Decrement the reference count of all arguments.

(3) Move the return value into the reserved register.

(4) Return.

3.6. Memory Management

3.6.1. Memory Allocator

The present version of TINYTALK uses a reference count strategy for controlling the creation and deletion of objects. It has been shown that, in the standard case, reference counting gives worse overall total performance than garbage collection, but better per unit time worst case performance. That is, more cycles are wasted in memory manipulation, but the load is more evenly spread. Section 6.2 gives some ideas on using inference information to eliminate unnecessary reference counting operations, which may shift the balance in favor of reference counting.

The memory allocator uses a circular first fit strategy [Knuth 1973], but has overlaid on that a special interface which speeds up the creation and deletion of objects. We will call the first the *real* allocator, the second the *quick* allocator. The quick allocator operates as follows: an array of linked lists of free objects is maintained independently of the circular free list. The index into the array is the size of the object; there is one list for each size up to some limit MAXSIZE. When an object of with a size less than MAXSIZE is freed, if the list for that size has less than MAXLEN elements in it, the object is prepended to that list instead of being returned to the circular free list. If the object is too big, or its list is full, it is freed as normal. When an object of some size less than MAXSIZE is requested, the appropriate list is first examined to see whether it has such an object, and only if not is the real allocator invoked. It is expected that in many cases most of the requests will be dealing with objects of one of the builtin classes, particularly *Integers* or *Reals*, and that allocations and frees of these tend to stay fairly in sync (see sections 5.3 and 5.4).

3.6.2. Strategy for Manipulating Reference Counts

The following rules are used to generate instructions to maintain correct reference counts during the execution of the program:

(1) Increment the reference counts of variables appearing in expressions.

(2) Decrement the reference count of a unit being voided. A unit is voided when it is followed by a period, or is in the controlling part of a control structure (**if**, **while**, or **for** statement).

(3) Decrement the reference counts of local variables, arguments, and *self* upon method exit.

(4) Increment the reference count of the right hand side of an assignment for each left hand side.

(5) Decrement the reference count of the old value of the variable on the left hand side of an assignment.

(6) In a unit (or right-hand-side) of the form $t\ m_1\ ;\ m_2\ ;\ \ldots\ ;\ m_n$ decrement the reference count of the return value of $t\ m_1$, $t\ m_2$, ..., $t\ m_{n-1}$ and of t itself, but not of $t\ m_n$, since this is the value of the unit.

3.7. Runtime Error Handling

Most errors that can occur at runtime are checked for. Array subscript out of bounds, string subscript out of bounds, message not understood by receiver; all result in sensible errors. The unique string table is assembled with the program, so that actual names of messages and classes are given. One of the compile time options causes assembly of a line number table, so that runtime errors give the source line on which they occur.

CHAPTER 4

PROBLEM AND SOLUTION

4.1. The Problem

Consider the message sending sequence. Assuming a procedure is to be called, the pushing of parameters and receiver onto the stack is unavoidable. However, the lookup operation involves at the very least, a procedure call, a hash value computation, a probe and comparison, and a procedure return. If a reprobe or two is necessary into the message hash table, or if the message is not found and the whole operation must be repeated with the message table of the superclass, the overhead is even worse. This is the problem.

4.2. The Objective

All of this can be avoided if the class of the receiver can be inferred at compile time. As an example, consider the following statement:

$y \leftarrow x + 2.$

In the normal case, this results in:

(1) Push 2

(2) Find the class of x.

(3) Look up + in its message table.

(4) Push x.

(5) Call the method returned by the lookup.

However, if it can be inferred that x is an integer before the statement, the steps could be:

(1) Push 2.

(2) Push x.

(3) Call the *Integer_plus* routine.

thus eliminating the expensive lookup. In addition, if the message is a builtin one that maps into a few native instructions, it is a candidate to be replaced by an inline expansion. The objective is to do these simplifications wherever possible.

4.3. The Solution

The basic body of information needed to remove lookups as described above is the set of possible types of each variable at each program point. First, what is a program point? In the type inference algorithms presented in [TEMPO], a program point is placed between every statement and the one(s) immediately following it in the flow of execution. This works in the case of TEMPO, because of the one dimensional nature of TEMPO programs, and because TEMPO statements are defined in such a way that first computation, then assignment takes place. Thus no type changes caused by a statement can affect the execution of that statement.

Unfortunately, since TINYTALK allows nesting of statements, the statement is not a good unit of execution for inference on TINYTALK programs. Instead, the approach taken is to first compile the TINYTALK program into an intermediate form before doing type inference on it. The unit of execution upon which inference takes place is the intermediate code item. The general outline of processing is divided into six phases:

(1) Parse the entire program into a parse tree representation.

(2) Generate intermediate code for each message and for the main program. At the same time construct a program flow graph by joining with program points all pairs of intermediate code items which will execute sequentially.

(3) Set up type expressions for variables affected by each code item in each program point.

(4) Solve these type expressions.

(5) Use the results to replace lookups with direct calls or with inline expansions.

(6) Produce the assembly program, doing some final peephole optimizations, assemble and link.

Since we are dealing with the intermediate form, it is necessary to introduce pseudo-variables which do not appear in the original program. These include the stack variables s_1, s_2..., s_n, the temporary variables, r_n, and the message return value *retval*. The stack pseudo-variables s_1, s_2..., s_n, represent values stored on the stack in the process of passing arguments to methods. The temporary pseudo-variables, r_n, are used to store intermediate results in the evaluation certain complex units. In the VAX implementation, temporary variables (and the pseudo-variable *retval*) are stored in registers.

4.4. Phase 1: Construct Parse Tree

The construction of the parse tree is handled by a standard recursive descent parser based on the BNF for the language (see appendix A). The BNF was designed so that one token lookahead is all that is necessary.

4.5. Phase 2: Generate Intermediate Code and Flow Graph

4.5.1. Intermediate Code

The intermediate code is a set of code operators with variable numbers of arguments, representing the primitives of TINYTALK execution. The arguments of all the code operators (SEND excepted) are of two forms, variables and labels.

A variable is a structure representing one unique user variable in the program. It gives the name of the user variable, and additional qualifying information. For an argument or local, the qualifying information consists of the message in which it is defined. For a field variable, the qualifier is the class of which it is a field. Globals have no qualifiers. This qualification is necessary since distinct variables can have the same name.

A label is a small unique integer used in the final output phase to create a label name for the assembler.

The following are the code operators initially generated:

GENARG *argument*

This operator pushes an argument to a message onto the stack. *Argument* is a variable.

SEND *message*, *receiver*, *argument count*

This operator is used to generate code to send a message. *Message* is the unique string of the message being sent. *Receiver* is the variable which is to receive the message. *Argument count* is the number of arguments the message is being given.

RETURN *value*

This operator produces code to return from a method, yielding *value* (a variable) as the result.

MOVE *from, to*

This operator moves the value of one variable, *from*, to another, *to*. It is used in conjunction with reference count manipulation operators to code assignments, and to manipulate temporary results.

IFTRUE *variable, label*

This operator produces code to jump to *label* if *variable* is equal to the builtin constant *true*.

IFALSE *variable, label*

This operator produces code to jump to *label* if *variable* is equal to the builtin constant *false*. These two operators are used to implement control flow in **if**, **while**, and **for** statements.

JUMP *label*

This operator implements unconditional branching and is used to jump back to the top of a loop.

LABEL *label*

This operator places a label for a destination of a JUMP, IFTRUE, or IFALSE.

In addition, there are two operators which are used to do the reference counting.

INCREF *variable*

Increment the reference count of *variable* by one.

DECREF *variable*

Decrement the reference count of *variable* by one.

Appendix B gives the VAX implementation of the code operators.

4.5.2. Generation of Flow Graph

In order to make type inferences using data flow analysis, the control of flow in the program must be represented. This is done by means of a flow graph. When one code item can execute directly after another, a program point is placed between them. The entire net of code items joined by program points constitutes the flow graph.

A program point contains pointers to the preceding and following code items and a list of variable instances. Each variable instance[1] gives the type information about a variable at the program point. In addition, in order to keep track of the usage of the stack variables s_1, $s_2, \ldots$, s_n, it is necessary to know at each program point the value of n, the height of the stack relative to its height at the beginning of the method. Each program point records this information as well.

A code item contains, as well as its code operator and arguments, pointers to any preceding and following program points. Each program point has no more than one preceding and one following code item, however, code items may have several preceding and following program points. The rules below give the method of generating the connections between program points and code items.

(1) The initial code item has a preceding program point (numbered 1) whose preceding code item is null.

(2) The final code item has a following program point whose following code item is null.

[1]See section 4.6.2

(3) GENARG, SEND, and MOVE code items have one following program point. They have one preceding program point unless preceded by a JUMP or RETURN. If preceded by JUMP or RETURN, they represent unreachable code; a warning is generated and everything up to the next LABEL is flushed.

(4) IFALSE and IFTRUE code items have one preceding program point, but two following program points. One of these is between the IFALSE (IFTRUE) and the code item which would be executed in the case of no jump. The other is between the IFALSE (IFTRUE) and the LABEL code item which is the target of the jump if the condition is met.

(5) JUMP code items have one preceding and one following program point. The following program point is between the JUMP and its target LABEL code item.

(6) LABEL code items have one following program point and one or two preceding program points.

The introduction of the LABEL code item type as the target for all branches guarantees that no code item will have more than a total of three preceding plus following program points.

As an example, the statement

while i<=10 **do** i ← i+1.

produces the intermediate code (leaving out the reference count operations)

```
LABEL    L0
GENARG   10
SEND     '<=', i, 1
IFALSE   retval, L1
GENARG   1
SEND     '+', i, 1
MOVE     retval, i
JUMP     L0
LABEL    L1
```

(L0 and L1 represent small unique integers). When the program points are added, the result is as in figure 4.1.

4.6. Phase 3: Set up Type Expressions

4.6.1. Type Expressions

A type expression is used to represent the information about the type of a variable at some point.

4.6.2. Composition of Type Expressions

A type expression is made up of primaries, representing sets of possible types, combined with the ∪ and ∩ operators and grouped with parentheses. The primaries are referred to hereafter as *nodes*; and are represented as leaves of an expression tree whose internal nodes are ∪ or ∩. The following are the type expression nodes:

type set

This node represents a set of types, or classes, of which the variable might be an instance. For example, the set {*Real*,*Integer*} represents the possibility of being of class *Integer* or of class *Real*. Two type sets are of special interest: *null*, the null set, and *gen*, the set of all defined classes, the universal set.

A(variable)

This node represents the type of an argument variable at the beginning

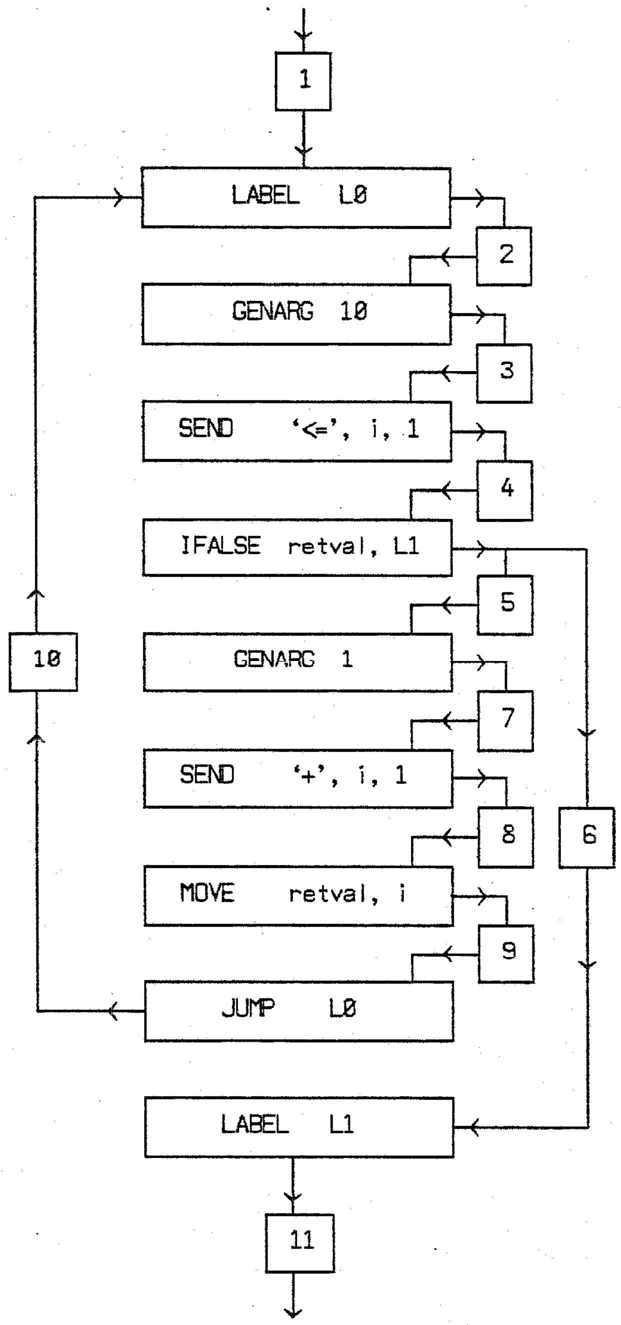

Figure 4.1

Intermediate Code and Flow Graph for
while i <= 10 do i <- i+1

of a method. The *variable* is either an argument to the method, or is *self*.

V(variable, program point)

This node represents the type of a variable at some program point.

R(message, rtype, argtype_1, ..., argtype_n)

This node represents the return type of message *message* when sent to a receiver of type *rtype* and given arguments of types $argtype_i$. *Message* is the name (unique string) of some message; *rtype* and $argtype_1$, ..., $argtype_n$ are other type expressions. Note that this is a non-trivial thing to calculate because, if the receiver is not a type set with a single element then a number of methods are implied by the message, each with its own possible return type expression.

As an example, R('+', {*Integer*}, {*Integer*,*Real*}) represents the return value of '+' when the receiver is *Integer* and the argument is *Integer* or *Real*. Thus the type associated with $2 + x$, might be of such a form.

M(message)

This node represents the set of all types (classes) which understand *message* or have a superclass which does.

One point to note is that **M** primaries do not actually appear in any type expressions. The value of an **M** node can be determined at parse time, and the resulting type set is used directly in the type expressions. **M** primaries are used here only for clarity.

The procedure used to calculate the value of **M**(*message*) is a form of transitive closure on the graph of subclass dependencies. The reason for this is that if some class understands *message*, then for inference purposes, all its descendant classes do also; since sending that message to an instance of one of the descendant classes will not result in a runtime error. Thus the value of $\mathbf{M}(message)$ is the set of all classes that understand *message* ∪ all their subclasses ∪ all *their* subclasses ∪ ...

Following are some simple examples of type expressions:

x: $\{Integer, String\}$

x is of type *Integer* or of type *String*

y: $\mathbf{M}('+')$

y is one of the types which understand '+'. This might initially be $\{Integer, Real, Char, String\}$, but could be extended by user defined methods for '+' in other classes.

z: $\mathbf{V}(x, 10)$

z here is of the same type as x is at program point 10.

4.6.3. Variable Instances

Since the goal of the type inference procedure is to know the type of each variable of interest at each program point, a structure called a "variable instance" is created to represent this information. A program point contains a list of these variable instances.

A variable instance contains the variable of which it is an instance and four type expressions. The first, called the *vi_tformula*, is the type expression giving the type at that program point. The second, called the *vi_dtype*, is the definitive type. The final result of solving the type expressions (which is a type set) is stored here. The third and fourth, called vi_ptype_0 and vi_ptype_1 are used to store intermediate results in the type expression solution phase (see section 4.7).

4.6.4. Significant Variables

In the execution of a piece of code, only certain variables can be accessed or modified. Hence these are the only variables which might need variable instances in the program points of that piece of code. In particular, a method can only access locals, arguments, and fields; the main program can only access globals.

Field variables pose a significant problem. The reason for this is that any message sent might potentially modify the type of a field even if it does not do so explicitly. This is the aliasing problem found also in program proving theory. A field variable refers to that particular field in the object referred to as *self*, the receiver of the present message. However, if this object is also the field of some other accessible object, or is in an array, it has, in effect an alias. Then some message innocently called may change the field in which we are interested by accessing the object via its alias. Whether this is the case depends on the dynamic nature of the program, i.e. we can't tell without running it to find out. Static type inference methods cannot detect such behavior. At present, fields are always assumed to be *gen* (but see section 6.3).

In addition, in order to evaluate **R** type functions (see section 4.6.1), the overall return type of each method must be recorded. This pseudo variable will be called *m_return*. It is a significant variable in methods.

4.6.5. Rules for Propagation of Types over Code Items

If, at some point in the program, a set of variables are known to be of certain types given by their *vi_tformulas*, the code item after that program point will possibly modify the *vi_tformulas* of a number of these variables. Then the program point after the code item will have a variable instance list somewhat different from the program point before.

In the algorithm described in [TEMPO], a variable instance for every significant variable occurs in the list for every program point. If one considers a method of, say, three arguments, five local variables, two temporaries, three stack variables and *retval* this amounts to 16 variables to keep track of. If it has 100 program points, then 1,600 variable instances are needed. Since the type expression solution algorithm must operate on all the methods simultaneously, this is multiplied by, for example, 30 methods in a small to medium program[2], giving 48,000 variable instances. This is clearly outrageous[3]. Since the code items affect the types of zero to at most two variables, it is ridiculous to create a variable instance for all the unaffected variables.

The approach taken here is to only put in the variable instance list of a program point those variables which have *changed* from the immediately preceding program point(s). Thus, while the algorithm of [TEMPO] records the type state at each program point, the algorithm described here records only the *changed* state.

Program point 1 has no preceding code items; it represents the variable type state at the beginning of the method and contains in its variable instance list instances for all significant variables (section 4.6.3). Variable instances for globals are given *vi_tformulas* of *gen*. Locals have the type set {*Object*} as their *vi_tformula*, since they are initialized to *nil* when a method is invoked. The *vi_tformulas* of arguments are initialized to A primaries. *Self* is initialized to a type set whose only member is the class in which this

[2]The towers of hanoi program has 35 methods in it.

[3]In the present implementation, a variable instance takes up 20 bytes, resulting in nearly one megabyte in variable instances alone, not counting the up to four type expressions (see section 4.6.2) which go with each one.

method is defined.

Thereafter, the following rules give the effects of the various code items on the *vi_tformulas* of the variables involved (P refers to the preceding program point, F to the following):

GENARG arg

F's stack counter is one more than P's. Call this new value n. A variable instance for the stack variable s_n is created and put into F's variable instance list, and is given a *vi_tformula* of $\mathrm{V}(arg,P)$. Basically, pushing an argument onto the stack causes the top of the stack to have the same type after the operation as the argument did before.

For example, after "GENARG 10", s_n is of type $\{Integer\}$. If after program point 42 there is "GENARG x", at program point 43 s_n is of type $\mathrm{V}(x,42)$.

SEND message, receiver, argcount

F's stack counter is *argcount* less than P's. Call this new value n. A variable instance for the pseudo-variable *retval* is created and put in F with a *vi_tformula* of $\mathrm{R}(message, receiver, s_n, s_{n-1}, \ldots, s_{n\text{-}argcount+1})$. More intuitively, the type of *retval* after the SEND is the type returned by the message with its arguments typed as before the SEND.

A special case exists when the message is *new* (or *new:to:*) and the receiver is a class constant, such as *Integer*, *Real* or a user defined class. In this case, the type of *retval* is set to the class a new instance of which is being created. This is the only time during the type expression setup where knowledge about the semantics of a particular message is used.

A variable instance in F for *receiver* is created and its *vi_tformula* is set to **M**(*message*) ∩ **V**(*receiver*,*P*). In English, the type of the receiver after the SEND must satisfy simultaneously two constraints: it must not have taken on a type which it couldn't have been before the SEND, (since methods cannot modify their receivers), and it must be of a type which understands[4] the message (since otherwise a runtime error would have occurred in the process of sending).

For example, if following program point 42 there is "SEND '+',*x*,12", then at program point 43 the type of *retval* is **R**('+',**V**(*x*,42),{*Integer*}), and the type of *x* is **M**('+') ∩ **V**(*x*,42).

MOVE fromvar,tovar

A variable instance in *F* for *tovar* is created and its *vi_tformula* is set to **V**(*fromvar*,*P*). That is, *tovar* after has the same type as *fromvar* before.

For example, after "MOVE 'A',*x*", *x* is of type {*Char*}. If "MOVE *x*,*y*" follows program point 42, then at program point 43 *y* has type **V**(*x*,42).

RETURN val

This code item does not modify the variable instance list; instead it modifies *m_return*[5] for this method. Specifically, *m_return* := *m_return* ∪ **V**(*val*,*P*). Essentially, the return type of a method is the union of the types of the arguments of all its RETURN code items.

LABEL label

If the LABEL code item has only one preceding program point, then no variable instances are put into *F*. If it has two preceding program points, call them P_1 and P_2, then *F*'s variable instance list has put in it one variable instance for every significant variable, and one for each stack

[4]"understands" is here used in the sense of section 4.6.1, i.e. understands or has a superclass that does.

[5]see section 4.6.3

variable presently in use. This last can be determined from the stack counter in P_1, (which is always the same as in P_2). The *vi_tformula* of the variable instance of each variable x in F is $V(x,P_1) \cup V(x,P_2)$. What this says is that if a LABEL can be reached from two program points, then directly after the LABEL the type of a variable will be the union of its types at each program point directly before the LABEL. The purpose of this rule is explained in section 4.7.2.

IFTRUE, IFALSE, JUMP, INCREF, DECREF

These code items do not provide any information about the types of their variables, so their following program points always have empty variable instance lists.

4.6.6. A Small Example

This section presents a small example of type equation propagation over a few code items. The method

```
message m: a | b
(
   b ← a + 2.5 .
   ^ b
)
```

produces the intermediate code items (ignoring reference counting), with program points assigned:

```
   [PROGRAM POINT 1]
GENARG    2.5
   [PROGRAM POINT 2]
SEND      '+', a, 1
   [PROGRAM POINT 3]
MOVE      retval, b
   [PROGRAM POINT 4]
RETURN    b
   [PROGRAM POINT 5]
```

The significant variables are identified to be a and b, and equations for the types of these variables, and of the pseudo-variables *retval* and the s_i are as follows:

```
   [PROGRAM POINT 1]
     a:      A(a)                      (a is an argument)
     b:      {Object}                  (b, a local is initialized to nil)

GENARG       2.5

   [PROGRAM POINT 2]
     s_1:    {Real}                    (a stack variable is used)

SEND         '+', a, 1

   [PROGRAM POINT 3]
     retval: R('+', V(a,2), V(s_1,2))  (return value of '+')
     a:      M('+') ∩ V(a,2)           (a must understand '+')

MOVE         retval, b

   [PROGRAM POINT 4]
     b:      V(retval,3)               (b =retval → type of b = type of retval)

RETURN       b

   [PROGRAM POINT 5]
```

The value of *m_return* for this method is, as a result of the RETURN code item, $\mathbf{V}(b,4)$.

Notice that, in line with the comments of section 4.6.5, only type equations for variables whose type has changed occur at each program point.

Notice also that there are **V** nodes referring to variables and to program points not containing equations for those variables. For example, the equation for a in program point 3 contains a **V** node referring to the type of a at program point 2, but there is no equation for the type of a in program point 2. This problem is dealt with in section 4.7.2.

4.6.7. A Full Example

As another example, consider the following message definition which adds the message *sum* to class *Array* to sum the elements of the array.

```
class Array understands
  message sum | i n
  begin
    for (n ← 0. i ← self lb). i <= self ub. i ← i+1 do
      n ← n + (self at: i).
    ^ n
  end
```

Program 4.1

The intermediate code and flow graph is shown in figure 4.2. and variable instances and their type expressions are added (figure 4.3). Note that, in addition to the type expressions for each variable instance, a type expression for the method return value, *m_return*, is produced. In this case, it is simply **V**(n,20) since there is only one RETURN code item.

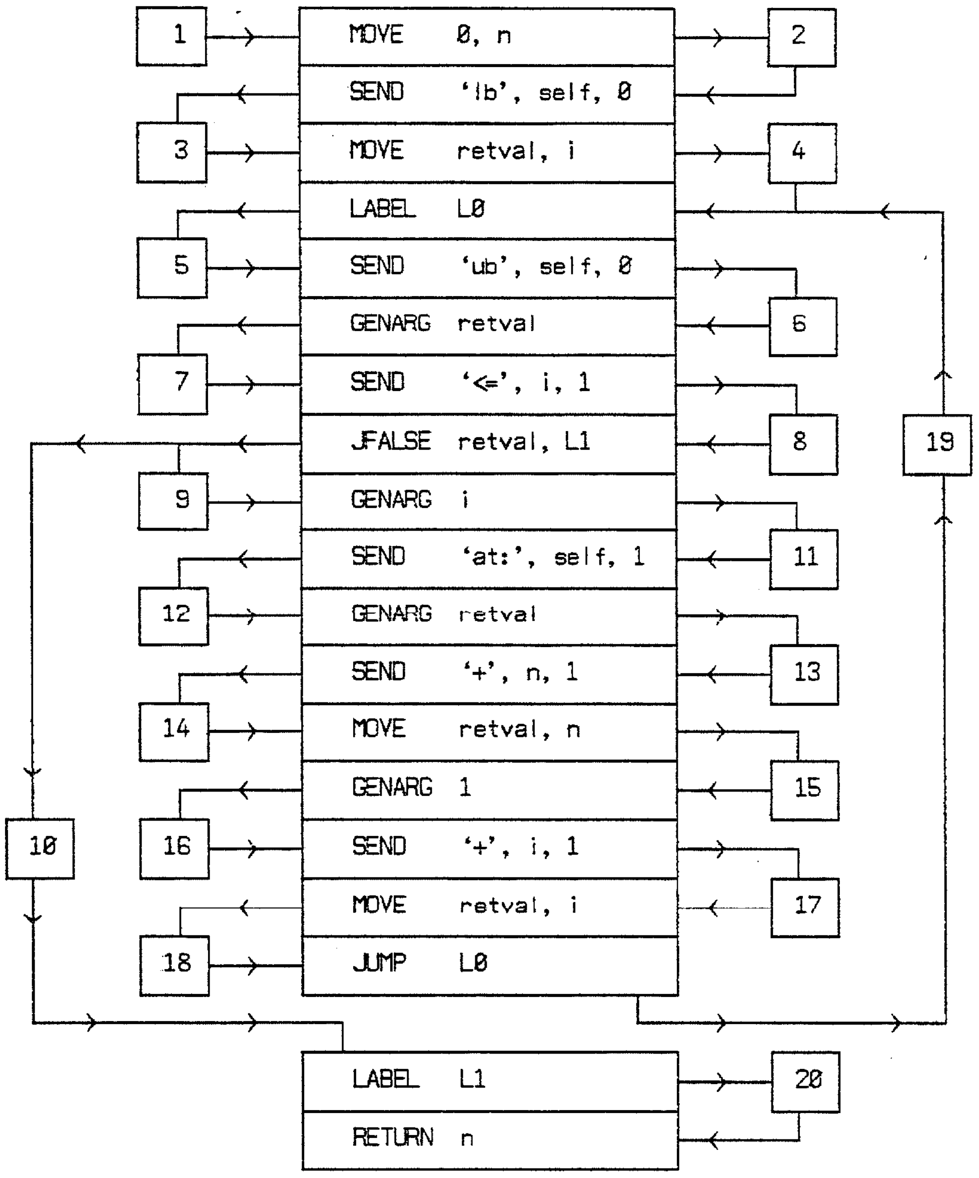

Figure 4.2

Intermediate Code and Flow Graph for Program 4.1

program point	stack top	variable instance list	
1	0	**i**: **n**: **self**:	{Object} {Object} {Array}
2	0	**n**:	{Integer}
3	0	**self**: **retval**:	$V(self,2) \cap M('lb')$ $R('lb', V(self,2))$
4	0	**i**:	$V(retval,3)$
5	0	**i**: **n**: **self**:	$V(i,4) \cup V(i,19)$ $V(n,4) \cup V(n,19)$ $V(self,4) \cup V(self,19)$
6	0	**self**: **retval**:	$V(self,5) \cap M('ub')$ $R('ub', V(self,5))$
7	1	s_1:	$V(retval,6)$
8	0	**i**: **retval**:	$V(i,7) \cap M('<=')$ $R('<=', V(i,7), V(s_1,7))$
9	0		
10	0		
11	1	s_1:	$V(i,10)$
12	0	**self**: **retval**:	$V(self,11) \cap M('at:')$ $R('at:',V(self,11),V(s_1,11))$
13	1	s_1:	$V(retval,12)$
14	0	**n**: **retval**:	$V(n,13) \cap M('+')$ $R('+',V(n,13),V(s_1,13))$
15	0	**n**:	$V(retval,14)$
16	1	s_1:	{Integer}
17	0	**n**: **retval**:	$V(i,16) \cap M('+')$ $R('+',V(i,16),V(s_1,16))$
18	0	**i**:	$V(retval,17)$
19	0		
20	0		

Figure 4.3

Type Equations for Intermediate Code in Figure 4.2

4.7. Phase 4: Solve Type Equations

4.7.1. General Description of Algorithm

The general method used to solve the type expressions is to repeatedly evaluate the *vi_tformulas* for each variable instance in the program using the values from the previous iteration to substitute for any **V** nodes. When an **R** node is encountered, the type of its receiver implies a set of methods which could be invoked. The value of the **R** node is, loosely speaking, the union of the evaluated *m_return* type expressions of each of these methods. The solution procedure operates on the entire program at once, and the main program is treated just like a classless method. The algorithm is roughly as follows:

```
$w$ := 1
while any changes have been made to any vi_ptypes
  $w$ := 1−$w$
  for each method
    for each program point
      for each variable instance, $v$
        evaluate $v$'s vi_tformula and store
          the result into $v$'s $vi\_ptype_w$
      end for
    end for
  end for
end while

for each variable instance, $v$, in each program point in each method
  put $v$'s $vi\_ptype_w$ into $v$'s vi_dtype
end for
```

Algorithm 4.1

The purpose of w is to switch between vi_ptype_0 and vi_ptype_1 at each iteration. The **while** condition is then computed on the basis of comparisons between the two.

4.7.2. Preliminaries

A number of preliminaries must be handled before the above algorithm can be executed. First, the *vi_tformulas* in their original form can contain nodes $\mathbf{V}(x,P)$, which refer to a variable x at a program point P which does not contain a variable instance for x[6]. This is a problem because when a node $\mathbf{V}(x,P)$ is being evaluated the value is found by looking in the vi_ptype_w of the variable instance for x in P.

The solution is to make a preliminary pass over the *vi_tformulas* of all the variable instances in the program looking for nodes $\mathbf{V}(x,P)$ and replacing P with the preceding program point closest to P containing a variable instance for x. This involves tracing backwards along the flow graph searching for a program point with a variable instance for the variable in question.

The rule for setting up the variable instance list after a LABEL code item (section 4.5.3) guarantees that there are never two preceding program points to chose from in searching backwards. This is because the only type of code item that can have two preceding program points is a LABEL, and a LABEL with two preceding program points always contains variable instances for all the significant variables of the method and for all in-use stack variables and temporaries. Therefore, any backwards search would certainly find its variable in the variable instance list of such a LABEL if not in a later one. As an example, figure 4.4 gives the results of applying this process to the type expressions of figure 4.3, with changed program point numbers in bold face.

[6]This does not happen in the algorithm presented in [TEMPO] because each program point contains a variable instance for each variable, most of which have as type expressions $\mathbf{V}$ nodes whose program point is the one immediately preceding.

program point	stack top	variable instance list	
1	0	**i**: **n**: **self**:	{Object} {Object} {Array}
2	0	**n**:	{Integer}
3	0	**self**: **retval**:	$V(\text{self},1) \cap M(\text{'lb'})$ $R(\text{'lb'}, V(\text{self},1))$
4	0	**i**:	$V(\text{retval},3)$
5	0	**i**: **n**: **self**:	$V(i,4) \cup V(i,18)$ $V(n,2) \cup V(n,15)$ $V(\text{self},3) \cup V(\text{self},12)$
6	0	**self**: **retval**:	$V(\text{self},5) \cap M(\text{'ub'})$ $R(\text{'ub'}, V(\text{self},5))$
7	1	$\mathbf{s_1}$:	$V(\text{retval},6)$
8	0	**i**: **retval**:	$V(i,5) \cap M(\text{'<='})$ $R(\text{'<='}, V(i,5), V(s_1,7))$
9	0		
10	0		
11	1	$\mathbf{s_1}$:	$V(i,8)$
12	0	**self**: **retval**:	$V(\text{self},6) \cap M(\text{'at:'})$ $R(\text{'at:'},V(\text{self},6),V(s_1,11))$
13	1	$\mathbf{s_1}$:	$V(\text{retval},12)$
14	0	**n**: **retval**:	$V(n,5) \cap M(\text{'+'})$ $R(\text{'+'},V(n,5),V(s_1,13))$
15	0	**n**:	$V(\text{retval},14)$
16	1	$\mathbf{s_1}$:	{Integer}
17	0	**n**: **retval**:	$V(i,8) \cap M(\text{'+'})$ $R(\text{'+'},V(i,8),V(s_1,16))$
18	0	**i**:	$V(\text{retval},17)$
19	0		
20	0		

Figure 4.4

Corrected Type Equations from Figure 4.3

Another enhancement can be made for efficiency. The evaluation of a *vi_tformula* involves evaluating any **R** or **V** nodes and replacing them with type sets. If a *vi_tformula* contains no **R** or **V** nodes, but only type sets and **A** nodes combined with union and intersection, its evaluation will be the same. Therefore, we make another pass over the variable instances and copy any *vi_tformulas* of this form directly to *vi_dtype*. The algorithm then becomes

```
for each variable instance, $v$, in each program point in each message
   if $v$'s vi_tformula contains no V or R nodes then
      set $v$'s vi_dtype to $v$'s vi_tformula
   end if
end for

$w := 1$
while any changes have been made to any vi_ptypes
   $w := 1-w$
   for each method
      for each program point
         for each variable instance, $v$
            if $v$'s vi_dtype = null then
               evaluate $v$'s vi_tformula and store
                  the result into $v$'s $vi\_ptype_w$
            end if
         end for
      end for
   end for
end while

for each variable instance, $v$, in each program point in each method
   if $v$'s vi_dtype = null then
      put $v$'s $vi\_ptype_w$ into $v$'s dtype
   end if
end for
```

Algorithm 4.2

4.7.3. Evaluation of vi_tformulas

The algorithm for evaluating the *vi_tformulas* is a straightforward recursive traversal of the type expression tree evaluating its subcomponents. For explanation purposes, a function *value* is defined which returns the value of a type expression. It takes one more parameter, a flag, which indicates what to do with **A** nodes. Then the treatment of the various types of nodes is as follows:

- **type set** t

  $value(t,flag) = t$

- **A node** a

  $value(a,flag) = (\text{ if } flag \text{ then } gen \text{ else } a\ )$

- **union**

  $value(t_1 \cup t_2,flag) = value(t_1,flag) \cup value(t_2,flag)$

- **intersection**

  $value(t_1 \cap t_2,flag) = value(t_1,flag) \cap value(t_2,flag)$

- **V node**

  If the variable instance, v, for x in P has its *vi_dtype* non-null, then $value(\mathbf{V}(x,P),flag) = v$'s *vi_dtype*, else $value(\mathbf{V}(x,P),flag) = v$'s vi_ptype_w.

- **R node**

  $value(\mathbf{R}(msg,rtype,a_1,a_2,\ldots,a_n),flag) =$

  $$Rvalue(msg,value(rtype,\text{true}),value(a_1,\text{true}),\ldots,value(a_n,\text{true}))$$

The evaluation of an **R** node involves the evaluation of the arguments, *rtype*, a_1, a_2, ..., a_n, but with *flag* being true, implying the substitution of *gen* for **A** nodes. This is because the types of **A** nodes are unknown, since the method can be called with parameters of any type (but see section 6.4). It also involves the function *Rvalue*, the method of evaluation for which is given in the next section.

4.7.4. Evaluation of R Nodes

In order to explain the evaluation of **R** nodes, a function *Rvalue*(*message*,*rtype*,a_1,a_2,...,a_n) is defined. At this point in the execution, *rtype* and a_1, a_2, ..., a_n are all type sets. *Message* implies a number of methods which may be invoked and return a value of some type. The set of possible methods, *M*, is found as follows. First find the set of classes which have methods for *message*. Note that this is not just M(*message*), it is more restrictive since subclasses of classes containing methods for *message* are not included. Next form the intersection of this with the *rtype* type set. Finally, *M* is the set of methods named *message* in the classes in this intersection. The value of the **R** node is the union of the return types of all the methods in *M*. The procedure for finding this union is:

(1) For each element *m* in *M*, form all possible n-tuples *t* by taking one element from each of a_1, a_2, ..., a_n.

(2) If *m* is a user defined method use the procedure outlined in section 4.7.4.1 to calculate the return type when the argument types are as in the n-tuple *t*. Otherwise, use the procedure in section 4.7.4.2.

(3) Form the union of the return types for all the n-tuples and for all elements a_i of *M*. This is the value of the R node.

If an argument is of type *gen*, special treatment is used: it is not split into its component elements in (1), but remains a single entity. This avoids having to perform many type expression evaluations for arguments with type *gen*, i.e. fields, and arguments whose type results from an **A** node evaluation.

4.7.4.1. Return Type of a User Defined Method

The procedure for calculating the return type of a user defined method is to evaluate its *m_return* type expression in exactly the same way as was described in section 4.7.3 with reference to *vi_tformulas*, with one difference. **A** nodes are replaced with a type set containing as its single element the corresponding element of the n-tuple for which the method return type is being evaluated. Thus, if a node $\mathbf{A}(x)$ is to be evaluated, and x is the second declared formal parameter, then the second element of the n-tuple is used as the only element of a type set which is then the value of $\mathbf{A}(x)$.

4.7.4.2. Return Type of a Builtin Method

The procedure for calculating the return type of a builtin method is entirely different. It is essentially a pattern matching process. Each builtin method has associated with it an ordered list of (*type-n-tuple* → *return-type*) pairs. This list is searched for a pair whose *type-n-tuple* matches the n-tuple for which the method return type is desired, and the value is then the *return-type*. There is also a default return type, a catchall which is used if no matches are found. For example, the pair list associated with the method *Integer_+* is ([*Integer*]→[*Integer*], [*Real*]→[*Real*]) and the default type is {*Integer*,*Real*}. What these really say is that *Integer* + *Integer* yields *Integer*, *Integer* + *Real* yields *Real*, and (for the purposes of type inference) *Integer* + anything yields *Integer* or *Real*.

One might expect that the catchall case should be *null* instead of {*Integer*,*Real*}. The reason it is not is due to the special handling of *gen* described in the last section. If an element of the n-tuple is *gen*, that n-tuple will not match any of the n-tuples in the pair list. In this case the default value will be used, and it must therefore be the union of all the *return-types* of all the pairs in the pair list.

4.7.4.3. Example of Evaluating an R Node

As an example, suppose some methods are defined:

```
class Integer understands
   message ltm: x then: y
   begin
      if 0<=x then
         ^ x
      else
         ^ y
   end

class Real understands
   message ltm: x then: y
   begin
      if self<=x then
         ^ 0
      else
         ^ y
   end

class Char understands
   message ltm: x then: y
   begin
      if self<=x then
         ^ '`@'
      else
         ^ y
   end
```

The method *Integer_ltm:then:* (method *ltm:then:* in class *Integer*) will have as its *m_return* type expression the expression $A(x) \cup A(y)$[7], *Real_ltm:then:* will have as its *m_return* the expression $\{Integer\} \cup A(y)$, and $\{Char\} \cup A(y)$ will be the return type expression of *Char_ltm:then:*.

Suppose now that, given these definitions, it becomes necessary at some point to evaluate

[7]Actually it will be $\mathbf{V}(x,1) \cup \mathbf{V}(y,1)$, but $\mathbf{V}(x,1)$ and $\mathbf{V}(y,1)$ have as their *vi_dtypes* $\mathbf{A}(x)$ and $\mathbf{A}(y)$ respectively, so that the earlier stages in the evaluation of *m_return* will change them to **A** nodes.

R(*ltm:then:*, {*Integer*,*Real*,*Array*}, {*Integer*,*Real*}, {*String*,*Char*})

We first find *M*, the set of methods which might be invoked. As outlined in section 4.7.4, we find out which classes have a method for *ltm:from:*, i.e. {*Integer*,*Real*,*Char*}. Next we take the intersection of this with the receiver type set {*Integer*,*Real*,*Array*}, yielding {*Integer*,*Real*}. Finally, *M* is the set of methods in these classes with the name *ltm:then:*, that is { *Integer_ltm:then:*, *Real_ltm:then:* }. Next we form all 2-tuples possible by taking one element from the first argument's type set, {*Integer*,*Real*} and one from the second's, {*String*,*Char*}. This gives the four 2-tuples [*Integer*,*String*], [*Real*,*String*], [*Integer*,*Char*], and [*Real*,*Char*]. Now we evaluate the *m_return* type expression (**A**(x) ∪ **A**(y)) of *Integer_ltm:then:*, with each of these assignments to the types of the arguments, giving

[*Integer*,*String*]	{*Integer*} ∪ {*String*}
[*Real*,*String*]	{*Real*} ∪ {*String*}
[*Integer*,*Char*]	{*Integer*} ∪ {*Char*}
[*Real*,*Char*]	{*Real*} ∪ {*Char*}

Then we do the same for *Real_ltm:then:*, giving

[*Integer*,*String*]	{*Integer*} ∪ {*String*}
[*Real*,*String*]	{*Integer*} ∪ {*String*}
[*Integer*,*Char*]	{*Integer*} ∪ {*Char*}
[*Real*,*Char*]	{*Integer*} ∪ {*Char*}

Finally, taking the union of all this, we get {*Integer*,*Real*,*String*,*Char*} as the value of the **R** node.

4.7.5. Substitution of A Nodes

The result of the preceding manipulations is that each variable instance of each program point of each message has as its *vi_dtype* a type expression devoid of **R** and **V** nodes. However, it may still contain **A** nodes. These must be removed to finally reduce each expression to a type set. Thus each *vi_dtype* is evaluated once more using the function *value* (section 4.7.3) with *flag* set, thus substituting *gen* for all **A** nodes. The complete algorithm is then:

```
for each variable instance, $v$, in each program point in each method
   if $v$'s $vi\_tformula$ contains no V or R nodes then
      $v$'s $vi\_dtype$ := $v$'s $vi\_tformula$
   end if
end for

$w$ := 1
while any changes have been made to any $vi\_ptypes$
   $w$ := 1−$w$
   for each method
      for each program point
         for each variable instance, $v$
            if $v$'s $vi\_dtype$ = $null$ then
               $v$'s $vi\_ptype_w$ := $value$($v$'s $vi\_tformula$, false)
            end if
         end for
      end for
   end for
end while

for each variable instance, $v$, in each program point in each method
   if $v$'s $vi\_dtype$ = $null$ then
      $v$'s $dtype$ := $value$($v$'s $vi\_ptype_w$, true)
   end if
end for
```

Algorithm 4.3

and the final result is that each variable instance has a type set giving the set of all possible classes of which the variable can be an instance at that point in the program. This is the information needed to simplify and optimize the code.

4.8. Phase 5: Using the Results of the Type Inference

4.8.1. Removing Lookups

The first stage is to replace SEND code items wherever possible with one of two other forms of SEND: ESEND or DSEND. ESEND, or Exact SEND, is the same as SEND, except that the message parameter[8] is replaced with an actual method. ESEND generates code to directly call the method, without any lookup being necessary. DSEND, or Decide and SEND, is the same as SEND, except that it has as an additional parameter a list of [class,method] pairs where the first is a possible class of the receiver which understands the message, and the second is the method to call in that case. DSEND generates code to test inline the class of the receiver on the basis that a few tests are faster than a lookup.

The simplification procedure is to scan through the code items of each message looking for SEND code items. When one is found, the flow graph is scanned backwards for a variable instance for its receiver. The back scanning is as described in section 4.7.2. The *vi_dtype* type set is obtained and checked to see how many members it has.

If there are no members then an error message is issued to the effect that the message is not understood by any possible classes of the receiver.

If there is only one member, it is checked to see if it understands the message. If not, its superclass is checked and so on until there is no superclass. In this case an error message is again issued stating that the message is not understood by any possible classes of the receiver. Otherwise, an ESEND is generated with the method corresponding to the message in the class which was found.

[8]See section 4.5.1

If, on the other hand, there are several members, a DSEND must be used.

Each element c of the receiver's type set is checked to see if it has a method for the message. If not its superclass is checked, and so on as above. If a class is finally found, the method, m, for the message is used, along with c, to form a pair $[c,m]$ which is added to a list, eventually to be used as the additional argument for the DSEND. If no ancestor class of c is found, no pair is formed.

If the list of $[c,m]$ pairs is empty, an error message is again issued explaining that the message is not going to be understood by the receiver. Otherwise, the number of elements is checked against a threshold. This threshold is ideally set at the point where the expected execution time of a sequence of branch on type instructions is equal to the expected execution time of a lookup[9]. If length of the list is greater than the threshold, the SEND is not replaced..

Otherwise, the DSEND will generate a chain of <branch on class; call method> sequences. For example, suppose we have

```
class A
begin
   message xeep:
      ...
end

class B
subclassof A
   ...
```

[9]or somewhat smaller, if code size is an important consideration, since a <branch on type; call method> chain takes up more instructions than a lookup call.

```
class C
begin
   message xeep:
      ...
end

class D
   ...
```

Xeep: is understood[10] by *A*,*B*, and *C*, *B* by virtue of it being a subclass of *A*. Now suppose *xeep:* is being SENT to a receiver known to be of type {*B*,*C*,*D*}. Then the list of [*c*,*m*] pairs would be {[*B*,*A_xeep:*],[*C*,*C_xeep:*]}, and the SEND would be replaced by

DSEND(*xeep:*, *receiver*, 1, {[*B*,*A_xeep:*],[*C*,*C_xeep:*]})

generating the equivalent of

```
if receiver is of classB then
    call A_xeep
elsif receiver of class C then
    call C_xeep
else
    runtime error: "message 'xeep' not understood by receiver"
```

4.8.2. Generating Inline Code

The next optimization which can be done is to replace selected ESEND's with inline code expansions. For this purpose, another code item, INLINE, is introduced. Its arguments are the actual inline function being performed[11], the receiver, and the method actual parameter(s).

In order to replace an ESEND with an INLINE, information about the type of not only the receiver, but also the arguments may, in general, be needed. When, for example, the method *Integer_+* is ESENT, if the argument is known

[10]again, in the sense of section 4.6.1

[11]At present, the builtin arithmetic functions and comparisons between any combination of integers and reals are expanded inline. This deals with the common loop constructs.

to be of class *Integer*, the INLINE operation *add integer to integer* will replace the ESEND. Similarly, if the argument is known to be of class *Real*, the INLINE operation *add integer to real* can be used. The inline expansions presently implemented require that the argument be of only one type, however this need not, in general, be the case (see for example section 6.8).

One final note: the standard return sequence for a method involves the method decrementing the reference counts of the receiver and arguments just prior to returning[12]. In addition, when the argument is simply a variable, an INCREF for it will appear directly before the GENARG[13]. When an inline expansion replaces an ESEND these INCREF's and DECREF's can frequently be removed. In particular, at this stage a peephole optimization is made to convert

```
INCREF      argument
GENARG      argument
INCREF      receiver
INLINE      operation,receiver,s_n
DECREF      receiver
DECREF      argument
```

(recall that s_n is the stack variable created by the GENARG) to

```
GENARG      argument
INLINE      operation,receiver,s_n
```

This yields another frequent case: the sequence

```
GENARG      argument
INLINE      operation,receiver,s_n
```

is replaced with

```
INLINE      operation,receiver,argument
```

thus avoiding the unnecessary push of the argument onto the stack.

[12]section 3.6.2, rule 3

[13]section 3.6.2, rule 1

Similarly, when the argument is the result of some calculation such that no INCREF appears directly before its GENARG,

```
INCREF      receiver
INLINE      operation,receiver,s_n
DECREF      receiver
```

is converted to

```
INLINE      operation,receiver,s_n
```

4.9. Phase 6: Final Assembly

The final phase is to convert the code items of each message to actual assembly language. The VAX representation of the various code items is given in appendix B. After the assembly language is output, it is assembled and linked with the runtime library.

One further peephole optimization is made at this stage: some more unreachable code is removed. In particular the implicit "RETURN self" at the end of each method is removed if all flow paths in the method lead to an explicit RETURN.

(It also turns out that occasionally two INCREF's on the same variable are generated in a row. This is replaced by a single "add 2" form).

4.10. Machine Independence

The use of code items for most manipulations of the program during type inference optimizations provides a good degree of machine independence. There are few assumptions made about the target machine.

It is assumed that a limited number of temporary variables are available. Temporaries are needed to evaluate units of the form $t\ m_1 ; m_2 ; \ldots ; m_n$ to save the value of t so that the various m_i can be sent to it. In the VAX implementation these come from the general registers, but there is no reason that they must.

It may appear that the calling sequence must use a stack. However, it is possible to implement GENARG in a way that does not use a stack. For example, the first GENARG allocates a frame on the heap, subsequent GENARG's add to it. SEND frees it when done. Thus any calling scheme which minimally allows recursion can be used. One thing to note is that the GENARG's are produced for the last argument first.

CHAPTER 5

RESULTS

5.1. Introduction

The primary objective of the type inference optimizations implemented as described above is to remove message lookups and do inline expansions whenever possible. This is the compiler's main goal. To evaluate the the overall effectiveness, it is necessary to address two performance issues: how good the compiler is at achieving its goal, and how much speedup this actually buys.

The first performance issue is, more precisely: how often are SENDs replaced by ESENDs or DSENDs, and how long, on average, are the compare/branch sequences to which DSENDs expand. The second performance issue is: how much is an individual ESEND, DSEND, or inline expansion better than a SEND.

5.2. Comparison to ALGOL-like Languages

Before addressing the performance of the compiler, it would be nice to have some idea of how much the polymorphic object-orientedness of TINYTALK costs relative to an ALGOL-like language such as PASCAL or C. The unit of measure is a "basic instruction", something which is a single short machine code instruction on most machines[1]. Two cases exist.

On the VAX, this is something line ADDL, CMPL, or PUSHL, but not CALLS, which involves all sorts of stack manipulations and is counted as several "basic instructions".

5.2.1. Simple Messages

First, consider a builtin message which would not involve a procedure call in a standard language. For example, integer addition on most machines is one "basic instruction". In TINYTALK, the steps executed even if an inline expansion is done include creation of a new object and placing the sum of the value fields of the receiver and argument into its value field. The addition takes one basic instruction again; the object creation takes seven in the present implementation, assuming that the real allocator is not needed and only the quick allocator is used[6] (the most common case). Thus even in the best of situations a simple operation is somewhere around 8 times slower in a totally polymorphic object oriented language.

5.2.2. Complex Messages

Next consider a message which would be implemented as a procedure in an ALGOL-like language. Here the situation is considerably better in that, in the best situation, where an ESEND is used, the only overhead is that of reference counting. This amounts to one or about seven basic instructions each for decrementing the reference counts of the arguments and receiver after the method terminates. The number of instructions is one if no free is required, seven in the present implementation if decrementing the reference count results only in a quick free. (It is considerably more than that if a real free is needed, but this is a relatively uncommon event). These numbers for overhead appear to be small compared to the lengths of the bodies of most procedures.

[6] It could be reduced to as little as 5 if the process were totally expanded inline.

5.2.3. Summary

Given a factor of around 8 for simple operations and an approximate equivalency for complex operations, it is expected that any polymorphic language will run (very roughly though perhaps somewhat less than) 8 times slower than a language featuring strong typing in applications where the generality of polymorphic variables is not necessary. This gives some idea of the difference between a best-case object-oriented program and a non-object-oriented one. Note that in applications where the full generality of polymorphic variables is necessary, the standard language will run at least as slowly, since the programmer must explicitly code all the manipulations done automatically in TINYTALK, and that, in addition, the program will be considerably harder to write and debug.

If a factor of 8 is the best that can be done, how closely does the compiler approach this, i.e. what fraction of the SEND messages it replaces with DSENDs, ESENDs, and inline expansions; how long are the average DSEND <compare,branch> sequences; and how much faster does all this work really makes things run. To find these things out, two performance tests are conducted.

5.3. Performance Test 1: Iterative Program

The first test of performance is an iterative program simulating the traditional PASCAL-like programming style. The program is simply

```
begin
   for i←1 ,i<=50000. i←i+1 do
end.
```

The compiler can be run with three levels of optimization. The first does no type inference optimizations at all. The second does type inference optimizations but does not replace ESEND's with inline expansions. The third does this as well.

The compilation of the program results in two SEND operations. At level 2, both of these are converted to ESENDs, since i is inferred to be of class *Integer* at all points after the first assignment. At level 3, both ESENDs are converted to inline expansions, since the messages invoked involve integer arithmetic and comparison. The type inference process required 5 iterations to converge to a solution of the type expressions. Since the minimum is two[3], this is acceptable.

First, data on the memory system; this is the same for each level of optimization:

Objects created	100000
Objects deleted	99999
Memory requests	100000
Real Allocations	2
Quick Allocations	99998
Real Frees	0
Quick Frees	99999

This shows that the two level memory management scheme (section 3.6.1) is very effective in eliminating calls to the real allocator; most requests are handled by the quick allocator. The fact that there are only two real allocations even though 10,000 objects are created shows that only two are ever in existence at the same time, a fact readily apparent from the program[4].

The timing and size information for the three levels follows:

One for *vi_ptype_0 and one for vi_ptype_1. See algorithm 4.3 in section 4.7.5.*

[4]the old and new values of i

Level 1	
Execution Time	37.9 seconds
Code Size	14336 bytes
Static Data Size	5336 bytes
Total Size	19672 bytes

Level 2	
Execution Time	27.7 seconds
Code Size	14336 bytes
Static Data Size	5336 bytes
Total Size	19672 bytes

Level 3	
Execution Time	11.0 seconds
Code Size	14336 bytes
Static Data Size	5336 bytes
Total Size	19672 bytes

These results indicate that the speedup of a simple counting loop, one of the most common control constructs, is about 40% at level 2 and 250% at level 3. Basically, at full optimization loops will execute 3½ times faster[5].

5.4. Performance Test 2: Recursive Program

The second performance test, the recursive implementation of the towers of hanoi[6], involves heavy use of user defined classes and messages, and is intended to more nearly exemplify the object-oriented programming style. The program is given in appendix D.

The compilation resulted in 129 SEND operations. At level 2 optimization, 63 of these were converted to ESENDs and 56 to DSENDs leaving only 10

[5]These timing figures can not be used for comparison to other languages, because the compiler at present produces debugging code to check reference counts, and because the runtime system contains considerable code for metering execution and verifying correct operation of the memory system.

[6]The towers of hanoi can be solved using a simple heuristic: basically move from the tower which was not the destination of the previous move, but this isn't a very useful test.

SENDs. The average length of the test sequences for the DSENDs was 1.7 compare/branch pairs. At first glance, it would seem that this number must be greater than two, however, in many cases where a DSEND is required, it is because the receiver has been inferred to be of one of two types, only one of which understands the message. Thus the DSEND produces code equivalent to

```
if receiver is of class1 then
    call class1_message
else { receiver is of some class which doesn't understand the message }
    runtime error
```

which has only one compare/branch. Given the average of 1.7, this happens somewhat less than half the time. The threshold for DSENDs was set to 4. At level 3 optimization, 7 of the ESENDs were converted to inline expansions, leaving only 3 SENDs. The compiler required only 8 iterations to solve the type expressions. The memory statistics for the towers of hanoi are:

Objects created	26203
Objects deleted	26203
Memory requests	26353
Real Allocations	40
Quick Allocations	26313
Real Frees	0
Quick Frees	26353

Once again, the quick allocator shows its worth. Most of the real allocations are probably for strings and arrays. The timing and size information for the three levels follows:

Level 1	
Execution Time	16.3 seconds
Code Size	20480 bytes
Static Data Size	7364 bytes
Total Size	27844 bytes

Level 2	
Execution Time	13.6 seconds
Code Size	19456 bytes
Static Data Size	7364 bytes
Total Size	26820 bytes

Level 3	
Execution Time	13.2 seconds
Code Size	19456 bytes
Static Data Size	7364 bytes
Total Size	26820 bytes

These results show about a 20% speedup from level 1 to level 2. The results differ from the previous set in that there is scarcely any appreciable speedup from level 2 to level 3. This is to be expected since only 7 ESENDs could be replaced by inline expansions.

5.5. Summary of Results

The results of the first test show that in certain instances, the compiler can infer enough about the program to make an optimal simplification; all the messages were replaced by inline expansions. In this ideal case a speedup of 3½ times is possible.

The first test also shows that the difference between SENDs and ESENDs for relatively small methods (integer addition and comparison) is about 40%, that is to say, lookups use up about 40% of the time. This would, of course, decrease in the case of larger methods.

The difference between ESENDs and inline expansions, on the other hand, is about a factor of 2½. The procedure calling overhead is once again a dominant cost when weighed against the relatively short method body involved in simple[7] messages (in this case addition and comparison). It really pays to be able to expand short methods inline[8].

The second test shows that, even with the primitive assumptions about the types of field[9] and argument[10] variables, the compiler managed to infer enough to remove all but three of the lookups from a total of 129, half with direct calls to the correct method. Furthermore, only eight iterations of the inference algorithm were needed. Again remembering that the minimum is two, this is an effectively rapid convergence.

Thus, even in more realistic circumstances than those provided by the first test, a substantial fraction of the lookups can be replaced by compile time sophistication. More disappointing is the relatively small number of inline expansions, which, as the first test showed, yielded the greatest gain.

One final note is that the optimized code winds up being smaller than the unoptimized version, since an ESEND takes less code than a SEND.

[7]see section 5.3.1

[8]Section 6.7 gives some ideas on how to extend inline expansion to selected user defined methods.

[9]section 6.4 deals with this problem.

[10]section 6.3 deals with this problem.

CHAPTER 6

FUTURE WORK

6.1. Use of General Registers

The present version of the compiler stores all local variables on the stack. However, the compiler has enough information to guess at the relative frequency of use of local variables, and could decide to store a number of the most commonly used local variables of each message in general registers. Similarly, if one or more of the arguments are heavily used, they could be copied to registers at the beginning of a method.

6.2. Constant Tracing

As well as tracing the type of a variable through successive code items, the inference algorithm could trace whether a variable is a constant. For example, in

```
x ← 2.
y ← x+1.
```

the normal rules for manipulation of the reference count state (see section 3.6.2) that the appearance of x in the second statement will result in code to increment x's reference count. This is unnecessary, however, because x has just been set to a constant, which has an infinite reference count and is never deallocated.

This optimization is certainly possible, but of doubtful usefulness. The example above is clearly artificial, because the programmer would write $y \leftarrow 3$ instead for the second statement. It is not clear how often these "constant variables" appear in actual programs. One place is in the first assignment to a local variable in a method. The local is initialized to *nil*, which need not be dereferenced. The only obvious common programming usage of a "constant variable" is as some kind of flag. This optimization may turn out not to be worth it.

6.3. Better Inference about Field Variables

Field variables present the most difficult inference problem. The present version of the compiler assumes the type *gen* for field variables at all points. The fact that every SEND code item potentially changes a field variable even though that variable may not appear as an argument or the receiver of the SEND makes impossible any meaningful type propagation based on static data flow techniques.

One possibility which I believe would cause significant improvement in the performance of the type inference algorithm is to examine the usage of the various field variables not on a per-code-item basis, but on a global basis. The idea is to note every appearance throughout the program of a field variable of a certain class on the left-hand-side of an assignment, and to infer that, during the execution of the program, the type of that field variable is the union of the types of all the corresponding right-hand-sides.

Notice that this inference is a global one in contrast to the purely local data-flow based inferences described in chapter 4; it uses information not about the local operations performed on a variable, but instead about global usages of the variable. It is therefore inherently worse than the local inferencing because, although some variable might be of several types in different places in the program, its polymorphism might also be well localized, something which a local type inference method would take advantage of, but which global type inferencing cannot detect.

From the code-item viewpoint, every MOVE to a field variable provides global information about the field's type. In the type expression setup phase[1] a field type expression would be formed for each field variable consisting of the union of the types of all the *from* variables in MOVEs to that field. Then in the solution phase[2] the evaluation of a V node for a field variable would result in the evaluation of the corresponding field type expression.

In most cases the fields of objects of a particular class remain of the same type throughout the execution of the program. For example, an object of a user defined class *Point* is likely to have the fields giving its coordinates remaining *Real* throughout (though note that *Object* is always a possible type for a field, since fields are initialized to *nil*). Thus the above scheme would, in many cases, replace *gen* with a type set containing only two elements (*Object* and one-other), a significant improvement.

It is expected that such a scheme for globally accounting for possible changes to the types of fields of a *general* object of some class, while not as effective as some hypothetical scheme for locally accounting for the types of the fields of *particular* objects, would nevertheless be very effective in many cases.

6.4. Better Inference about Method Arguments

In the last phase of solving the type expressions (section 4.7.5) all **A** nodes, which represent the types of arguments at the start of a method, are replaced by *gen*. A better choice would be the set of all types ever used as

[1]section 4.6
[2]section 4.7

that argument throughout the program.

A method similar to the one in the previous section for field variables can also be used to globally account for the possible types of argument variables. Each distinct argument would have an argument type expression associated with it in a way entirely analogous to the field type expression associated with each distinct field variable as described above. Instead of a MOVE signaling a possible modification of the type of the variable, the signal is a SEND of a message of the same name as the method in which the the variable is an argument. For example, consider

```
class A
begin
   message xeep: x
        ...
end

class B
begin
   message xeep: y
        ...
end

begin    { main program }
   ...
   r xeep: a
   ...
end
```

Here, *r xeep: 5* will result in the intermediate code (leaving out reference counting):

```
GENARG  a
SEND    'xeep:', r, 1
```

Then, after the SEND, something is known about the arguments of all methods which might be invoked by the SEND. In particular, it is now known that x in *A_xeep:* and y in *B_xeep:* could be of whatever type a might be directly before the SEND.

The final value of each argument type expression is just the union of all the types inferred from SENDs as described above. If all such SENDs are accounted for in the type expression setup phase, then in the solution phase, when an A node occurs, the corresponding argument type expression is just evaluated.

This procedure results in a fairly good inference about the types of arguments, but it is possible to do better. In the above example, the SEND contributed general information about the types of the arguments of all methods named *xeep:*, instead of only those which might be called. If, for example, it turned out that r was never an instance of class A, then the type inferred for x would be needlessly less precise because of the unnecessary inclusion of the impossible case. The SEND would never invoke A's *xeep:* and so the type of a could not affect the type of x as a result.

A better value for the type of, say x, is set of all types throughout the program which could actually be used as the argument to the method *A_xeep:* rather than the message *xeep:*. Unfortunately, this requires knowledge of the types of the receivers of messages *xeep:*, and at the expression setup phase no such knowledge is available.

The solution is to defer inference about arguments until after types of receivers are known, after the solution phase. Then it is possible to restrict the type of an argument variable to the set of types throughout the program which could actually be used as that argument. For example, in

```
01: class Integer understands
02:    message << n
03:    begin
04:      ...
05:    end

06: class Real understands
07:    message << k
08:    begin
09:      ...
10:    end

11: begin
12:    x ← 2  << 4 .
13:    y ← 2  << 1.5 .
14:    z ←1.6 << 1.5
15: end
```

within the body of *Integer_<<* (line 4), any references to *n* would presently be optimized assuming its type at method entry to be *gen*. However, it is clear from the rest of the program that the method is only called with its argument being *Integer* (line 12) or *Real* (line 13). Thus the set {*Integer*,*Real*} is a better value to use than *gen*.

Notice that, if the procedure described at the beginning of this section is used, *k* in the body of *Real_<<* (line 9) would be inferred to be {*Integer*,*Real*}, since this is the set of argument types used in all sendings of the *message* << (lines 12,13,14). The new procedure would infer it to be {*Real*} based only on line 14, a better result.

The implementation of the above enhancement is not straightforward, because the determination of the correct type for an **A** node could depend on the types of other **A** nodes, and so on. Loops are possible, as in the following example:

```
01: class Integer understands
02:    message @ x
03:    begin
04:       ^ self & x
05:    end
06:
07: class Integer understands
08:    message & y
09:    begin
10:       if self==0 then
11:          ^ 0
12:       else
13:          ^ (self-1) @ y
14:    end
15:
16: begin
17:    z ← 2 @ 3
18: end
```

Here, the final value to be used for $\mathbf{A}(x)$ in message '@' at line 4 is determined by the types of the argument in all usages of '@', that is in lines 13 and 17. Line 17 is straightforward: the argument is *Integer*. The type of the argument to '@' (i.e. y) in line 13 depends on the value to be used for $\mathbf{A}(y)$ in message '&'. This is determined by the type of argument in all usages of '&', namely line 4. But this is $\mathbf{A}(x)$, which is what we wanted to find out in the first place.

In order to solve this problem, we need some sort of iterative method similar to the one used to solve the type expressions in the first place. The question is, is it worth it. It would seem that the answer is definitely in the affirmative, because nearly all messages are not overloaded in their arguments. (Common exceptions are the arithmetic and comparison operations, but these do not matter since they are builtins). Most user defined messages expect each of their arguments to be of one particular type, and always are so invoked. Thus the above enhancement would, in many cases, replace *gen* with a type set containing only one element, a significant improvement.

6.5. Global Variables in Messages

The present definition of TINYTALK is severely limited in that global variables are not accessible inside messages. This is a desirable feature from the point of view of modularity, however, there are applications for which global information is necessary. In TINYTALK as defined, global variables are really equivalent to local variables of the main program.

When global variables are allowed inside messages, the type inference algorithm becomes substantially more difficult. First of all, the list of significant variables for methods must be expanded to include all global variables. Even those not accessed explicitly in the method must be included.

Furthermore, previously the SEND code item could only affect the pseudo-variable *retval*; in addition, something could be inferred about the receiver. Now, a SEND can also affect the types of all global variables. The process of setting up the type expressions must not only keep track of the return type of each method, but must also form type expressions for the possible final types of each global variable based on their initial types. Thus a new node, **G**(*global_variable*, *message*, *receiver_type*, *arg1_type*, ..., *initial_type*), must be introduced, which is the set of possible final types for *global_variable* (having the initial type *initial_type*) after *message* is sent to a receiver of type *receiver_type* with arguments of types *arg1_type*, ... Needless to say, evaluating a **G** node is not straightforward.

Class variables à la SMALLTALK can be done in the same way.

6.6. Elimination of Unnecessary Code

Methods return *self* when the execution falls out the bottom. In many cases the value returned is simply voided. For example, in the towers of hanoi program (appendix D) practically none of the methods which fall out the bottom ever have the resulting *self* used for anything at the call site.

If a method falls out the bottom, and if the resulting *self* is never used at any of the call sites, the code to return *self* in the method, and the code to void it (decrement its reference count) at the call site, might as well not be produced. This should produce noticeable, though not astronomical, execution time savings.

6.7. Inline Expansion of User Defined Methods

As can be seen from the towers of hanoi program, most classes contain methods to access or directly set the fields of objects of that class (*get* and *set* messages). These messages are exemplified by

```
class A
fields x
begin
    message x   (^x)    { get message }
    message x: v      (x ← v)     { set message }
end
```

These are very common in SMALLTALK programs, since unlike SIMULA, the fields are not directly accessible.

The *get* and *set* messages are excellent candidates for inline expansion, for two reasons. First, they are short and so execution time is dominated by the method calling sequence. Second, these messages tend to be more commonly called, since they are the primitive object manipulation operators for user defined classes.

To produce an inline expansion for a builtin arithmetic or comparison operator (see section 4.6.2), the type of the argument must be known to be a single type in order to select the correct version of the operator. However, in the case of *set* methods, it doesn't matter what the type of the argument is, so a greater proportion of *sets* can be expanded inline (*gets* pose no problem here, having no arguments).

6.8. Type Assertions

In builtin methods such as, for example, *at:* (array subscription) the first thing the method does is to check whether its argument is an integer. At present there is no way to do this in a user defined method. SMALLTALK provides a builtin message understood by class *Object* which returns the class of an object. Then something like

```
if x class ≠ Integer then
   { complain }
```

is used to decide if the arguments are of the right type. Usually the programmer doesn't bother and most user defined messages don't domain check their arguments. Instead they assume that the arguments are of the right type and let some builtin message fail later on if they are not. Using something similar to the test above has two problems. First, { complain } isn't standardized, and much code is duplicated with this kind of argument domain checking. Second, although it is clear to a human that after such a construct x is of class *Integer* (provided { complain } aborts execution), the type inference algorithm would have to know about the semantics of particular messages to figure this out, and it has no such knowledge.

Instead, a statement can be added specifically for the purpose of making assertions about the types of variables. This gives the same result and does not require a more complicated type inference algorithm. The format would be

assert *variable* **is** *class*

The intermediate code generator would produce a single code item

ASSERT *variable*, *class*

for this. Then the setup phase of the type inference algorithm would recognize that, after such a code item, *variable* would be of class *class*. The result of expanding ASSERT would be to produce code to check that *variable* is really of class *class* and to abort with an appropriate error message if not.

This gives a standardized argument domain checking facility and helps the compiler determine types of arguments (decreasing the severity of the **A** node problem described in section 3.4). The **assert** statement is not, of course, restricted to argument variables, but can be used as a form of note to the compiler in places where it might otherwise have trouble making useful inferences, and also as a debugging aid.

6.9. Special Representation For Integers

The present implementation of TINYTALK treats integers exactly the same as all other classes. SMALLTALK 76 as implemented by XEROX dealt with small integers specially. Integers sufficiently small to be less than the smallest pointer were not stored in objects on the heap, but in the variables (pointers) themselves. This encoding allowed integer arithmetic to proceed without using the memory allocator; it basically allows the compiler to take some advantage of the hardware operators transparently to the semantics of the language. This is a possible practical enhancement, but isn't too interesting from a theoretical standpoint.

6.10. External Compilation

The method of type inference optimization used in TINYTALK requires that the entire program be compiled at once. With very large programs, this is clearly impractical. External compilation is an area in which work needs to be done, but does not appear to pose any particularly difficult problems. The method used in SIMULA, namely the creation of attribute files giving type information about externally compiled classes, seems quite adequate to the task.

6.11. Summary

This chapter has outlined a number of areas where future work could be done to enhance the effectiveness of the compiler in inferring the types of variables. The four most profitable areas appear to be the scheme for dealing with fields types, the method for dealing with argument types, the inline expansion of user defined methods, and the **assert** mechanism.

CHAPTER 7

CONCLUSIONS

The results in chapter 5, coupled with the ideas in the last chapter lead to the conclusion that the type inference methods presented in this work are effective in obtaining a significant speedup in a polymorphic language, even one based on the object oriented paradigm with its accompanying high procedure calling overhead.

The analysis in section 5.2 leads to the conclusion that even with optimal type inference on a program which does not use the polymorphic nature of variables a factor of around 8 is lost in speed versus a typed language. This is due to the object-oriented nature of the language; even simple operations require the creation of an object to hold the result. It is also due to the declarationless nature of the language; if it had declarations, then types for which the hardware provides support could be handled as special cases, as is done in SIMULA.

One result of chapter 5 is unexpected; the factor of speedup even when all lookups are removed is only about 40%, which is significant, though not as large as expected. This indicates that lookups are not the primary factor causing object-oriented languages to be slow. The inline expansion, on the other hand, saved an additional factor of 2½. Apparently the main area in which TINYTALK suffers, at least in this implementation, is the overhead of procedure calling. Therefore, the primary goal of enhancements to the methods presented here must be to increase inline expansions, along the lines of section 6.7, for example. Needless to say, the improved methods of type inference described in sections 6.3, 6.4, and 6.8 are necessary to increase the opportunities for such inline expansion.

Another result of chapter 5 is that the choice of memory allocation strategy is the major factor affecting execution speed. This is to be expected since every message sent results in at least one, usually several, calls on the allocator.

The ability of variables to change types gives a very natural approach to overloading and generic procedures, and allows manipulation of complex data structures to be much more simply and easily programmed. The question is, is the corresponding 8 times decrease in performance worth the gain. This work has shown that is it possible to approach the limiting factor of 8 from a starting point of much worse by using data flow type inference methods. Developments in hardware will hopefully close the gap by providing object manipulation as a primitive just as addition and subtraction are on current machines. When this happens, type inference method similar to those described here can be used to produce efficient code for polymorphic, object-oriented languages. The programmer will then be free from arbitrary restrictions imposed to make it easier on the compiler, leaving him to express algorithms in a natural object-oriented way.

REFERENCES

[Birtwistle et al., 1973]
Birtwistle, G. M., O–J Dahl, B. Myhrhaug & K. Nygaard, *Simula BEGIN*, Petrocelli Charter, New York, 1973

[Wilcox et al., 1979]
Wilcox, Clark R., M. Dagenforde, G. Jirak, *Mainsail Language Reference Manual*, XIDAK, 1979

[Ledgard, 1980]
Ledgard, Henry, *ADA, An Introduction*, Springer Verlag, New York, 1980

[DARPA, 1981]
DARPA, *ADA Reference Manual*, Springer Verlag, New York, 1981

[INTEL, 1981]
Introduction to the iAPX 432 Architecture, INTEL, 1981

[Jensen and Wirth, 1974]
Jensen, K. & Niklas Wirth, *Pascal User Manual and Report*, Springer Verlag, Berlin, 1974

[Byte, 1981]
XEROX Learning Research Group, *The Smalltalk-80 System*, Byte Magazine, Vol 6, Number 8, pg. 36, August 1981

[TEMPO]
Jones, Niel D. & Stephen S. Muchnick, *Binding Time Optimization in Programming Languages: Some Thoughts Toward the Design of an Ideal Language* (preliminary)

[Kernighan and Ritchie, 1978]
Kernighan, Brian & Dennis Ritchie, *The C Programming Language*, Prentiss-Hall, New Jersey, 1978

[UNIX]
UNIX Programmer's Manual, Seventh Edition, Virtual VAX-11 Version, Vol 1, University of California at Berkeley, 1981

[Lindsey and van der Meulen, 1980]
Lindsey, C. H. & S. G. van der Meulen, *Informal Introduction to ALGOL 68*, Revised Edition, North-Holland, Amsterdam, 1980

[Knuth 1973]
Knuth, Donald E. *The Art of Computer Programming*, Vol 1, Second Edition, pg. 442, Addison Wesley, Massachusetts, 1973

[Robson, 1981]
Robson, David, *Object-Oriented Software Systems*, Byte Magazine, Vol 6, Number 8, pg. 74, August 1981

[Althoff, 1981]
Building Data Structures in the Smalltalk-80 System, Byte Magazine, Vol 6, Number 8, pg. 230, August 1981

[Ingalis, 1978]
Ingalis, D.H.H. *The Smalltalk-76 Programming System Design and Implementation*, Conf. Rec. 5th ACM Symp. on Principles of Programming Languages, pp 128-145, Tuscon, Arizona, 1978

[Deutch, 1981]
Deutch, L. Peter, *Building Control Structures in the Smalltalk-80 System*, Byte Magazine, Vol 6, Number 8, pg. 322, August 1981

[Suzuki, 1981]
Suzuki, Norihisa, *Inferring Types in Smalltalk, Xerox PARC, 1981*

APPENDIX A

Language Syntax

A.1 Notation

( )	grouping
x | *y*	*x* or *y*
{ *x* }	0 or more *x*
[*x*]	optional *x*
< *x* >	1 or more *x*
word	nonterminal
text	literal text
text	commentary

A.2 Lexical Categories

```
eof               ::=    end of file

keyword           ::=    id:

id                ::=    letter { digit | letter }

op                ::=    < op_character >
                         except that <- and = are reserved.

op_character      ::=    !|$|%|&|*|+|,|—|/|:|;|<|=|>|?|@|\|^|_|`|||~

int               ::=    [+|—] (
                         1|2|3|4|5|6|7|8|9 { digit }
                         | ( 0x | 0X ) < hexit >          hex number
                         | ( 0o | 0O | 0) < octit >       octal number
                         | ( 0b | 0B ) < 0 | 1 >          binary number
                         | 0
                         )

real              ::=    [+|—] <digit> [. { digit }] [ e | E  [+|—] <digit>]

string            ::=    characters or metacharacters in double quotes
                         two double quotes are used to denote a double
                         quote in the string

character         ::=    character or metacharacter in single quotes

comment           ::=    { text possibly containing nested { and }
                         }
```

```
begin           ::=   ( | BEGIN

end             ::=   ) | END

null            ::=

metacharacter   ::=   Two character sequence designating some non-
                      printing character.  The first character is a back-
                      prime or backslash.  The sequences recognized are:
                      ` <upper case letter or one of @ \ ^ ] [>
                      representing the corresponding control character
                      ` <1 to 3 octal digits>
                      representing the corresponding ascii character
                      `#            delete
                      `t            tab
                      `n            newline (linefeed)
                      `b            backspace
                      `f            formfeed
                      `r            return
                      `$            escape
                      Standard C escape sequences.
```

A.3 Syntax

```
program         ::=   { class_spec } block eof

class_spec      ::=   class id ( class_def | class_add )

class_add       ::=   understands mess_def

class_def       ::=   [ subclassof id ]
                      [ fields idlist1 ]
                      begin  { mess_def } end

idlist          ::=   { id }

mess_def        ::=   message mess_hdr [ | idlist ] block

mess_hdr        ::=   id1 | op id2 | < keyword id >

block           ::=   begin stmt_list end

stmt_list       ::=   [ statement { . statement } ]

statement       ::=   unit | if_stmt | while_stmt | for_stmt

unit            ::=   [^] { id ← } term [ k_selector { ; k_selector } ]
                      | null

if_stmt         ::=   if unit then statement [ else statement ]

while_stmt      ::=   while unit do statement
```

```
for_stmt        ::=    for unit . unit . unit do statement

term            ::=    factor { b_selector }

factor          ::=    primary { u_selector }

u_selector      ::=    id

b_selector      ::=    op factor

k_selector      ::=    < keyword term >

primary         ::=    id | integer | real | character | string | block
```

APPENDIX B

VAX Implementation of Intermediate Code

B.1 Register Usage

In the VAX implementation, several of the 16 general registers are reserved by the hardware and certain others by convention. The following table gives the usages:

R0	Reserved for method return values.
R1	Reserved for interface to C routines returning double results and not presently used.
R2	Messed up by C unsigned divide routine and therefore not used.
R3-R6	Free for intermediate results and saved by messages that use them.
R7	Contains -1 for indexing an object's class.
R8	Contains -2 for indexing an object's reference count.
R9	*retval*.
R10	Not used, reserved for pointer to class variables of *self*.
R11	*self*.
R12	hardware maintained argument pointer.
R13	hardware maintained call frame pointer.
R14	hardware maintained stack pointer.
R15	hardware maintained program counter.

Reserving registers 7 and 8 to always contain -1 and -2 respectively makes it possible to access the class and the reference count of an object directly using the VAX addressing modes.

B.2 Accessing Variables

All of the various kinds of variables are accessible using VAX addressing modes. The implementation is as follows (using UNIX *as* syntax):

global *varname*	varname
local number *n*	(-n*4)(fp)
argument number *n*	((n+1)*4)(ap)
field number *n*	((n-1)*4)(r11)
self	r11
class *Classname*	#Classname

To access the reference count of an object, we simply append **[r8]** to each of these. Similarly, appending **[r7]** accesses the object's class.

B.3 Intermediate Code Items

The implementation of the various types of intermediate code items in terms of VAX machine instructions is given below

INCREF variable

```
          incl      variable[r8]
```

DECREF variable

```
          sobgtr    variable[r8],label
          pushl     variable
          calls     $1,OS_free
     label:
```

GENARG variable

```
          pushl     variable
```

SEND message, receiver, argcount

```
          pushl     receiver[r8]          # OS_lookup looks up the message.
          pushl     message               # It takes as parameters the class and
          calls     $2,OS_lookup          # the message unique string number
          pushl     receiver              # and returns the method address
          calls     $argcount,(r0)        # in register 0.
```

MOVE from,to

```
          movl      from,to
```

LABEL label

```
     label:
```

JUMP label

```
          jbr       label
```

IFTRUE variable,label

```
        cmpl    $true,variable
        jeqlu   label
```

IFALSE variable,label

```
        cmpl    $false,variable
        jeqlu   label
```

RETURN variable

```
        movl    variable,r9
        jbr     returnlabel
```

In addition, the beginning of a method contains code to create and initialize locals: one

```
        pushal  nil
```

per local variable, and code to set up *self*:

```
        movl    4(ap),r11
```

Similarly, the method ending:

```
        movl    r11,r9              # return self
    retlab:
        movl    n,r11               # pass number of locals, n, to procedure
        jsb     OS_freelocals       # to decrement reference counts on locals
        jsb     OS_freeargs         # Also decref args and self
        movl    r9,r0               # and load return value into return register
        ret                         # Finally return
```

decrements locals and args, and, in accordance with VAX convention, returns the method result value via register 0. If there are no locals, the second and third instructions are not generated.

APPENDIX C

Running the Compiler Under UNIX

The general format of the command to invoke the compiler is:

st [-O[O]] [-g] [-c] [-S] [-v] *file*.(q|s|o)

where the meanings of (), [] and | are, as usual, grouping, optional item, and alternation, and *file* is any filename. If *file* has the suffix *.q*, it is compiled, assembled, and linked. If the suffix is *.s*, only assembly and loading are performed, and an suffix of *.o* causes loading alone. These conventions are the same as in the other UNIX compilers. The meaning of the various switches is as follows:

-O Do type inference optimization and use ESENDs and DSENDs wherever possible.

-OO In addition to -O, replace ESENDs with inline expansions wherever possible.

-g Compile debugging information into the program. This results in runtime errors giving the source line of the error, and provides some support for the UNIX debugger *sdb*.

-c Compile and assemble only, do not link.

-S Compile only, do not assemble or link.

-v Produce statistics about the compiler execution. Output includes the phase of compilation, data about the number of variables, parse tree nodes, type equation nodes, program points, variable instances, iterations through the type equation solver, and so on.

The result of compilation of *file.q* is three other files: *file.s*, *file.o*, and *file*. These are the assembly source, the unlinked object module, and the executable file respectively. The program can be run simply by typing

```
file
```

to the UNIX command interpreter, the shell.

APPENDIX D

Towers of Hanoi Program

```
{
     Towers of hanoi program designed to run on vt52 style video terminals.
     Plays the game 50 times with a 4 disk pile.
     Liberally translated by Niklas Traub from anonymous example in SIMULA.
}

{ Screen handling }

class Userview.
begin
     message clear.
     ( "`$H`$J" print )

     message home.
     ( "`$H" print )
end

{ Class representing a point on the screen }

class Point.
fields x y.
begin
     message x.      (↑x)
     message y.      (↑y)
     message x: v.   (x←v)
     message y: v.   (y←v)

     message moveto.
     (
          "`$Y" print.
          (59−y) asChar print.
          (32+x) asChar print
     )

     message deltay: dy.
     (
          "`$Y" print.
          (59−y−dy) asChar print.
          (32+x) asChar print
     )

     message deltax: dx.
     (
          "`$Y" print.
```

```
        (59−y) asChar print.
        (32+x+dx) asChar print
    )
end

{ One of the Towers of Hanoi }

class Tower.
fields contents position top.
begin
    message contents.     (^contents)
    message position.     (^position)
    message top.          (^top)
    message height.       (^top+1)

    { Initialization stuff }
    message at: p.     ( position ← p )
    message init | i.
    (
        contents ← Array new: 0 to: 3.        { 4 high stack }
        top ← 0.
        for i←0. i<=3. i←i+1 do
            contents at: i put: "  |  "
    )

    { Draw each of the elements of the tower in a vertical line at position }
    message draw | p i.
    (
        position moveto.
        for i←0. i<=3. i←i+1 do (
            (contents at: i) print.
            position deltay: i+1
        )
    )

    { Stack a disk on top of the tower }
    message stack: d.
    (
        contents at: top put: d.
        position deltay: top.
        if d==false then ^false.
        d print.
        top ← top+1
    )

    { Remove the disk on top of the pile }
    message remove | d.
    (
        if top<=0 then ^false.
        top ← top−1.
        d ← contents at: top.
        contents at: top put: "  |  ".
        position deltay: top.
```

```
            (contents at: top) print.
            ^d
        )

        { Return the disk on top of the pile }
        message topdisk.   (^contents at: top)
end

{ This Class defines the disks that are moved around on the towers }

class Disk.
fields size shape.
begin
        message size.          ( ^size )
        message size: v.       ( size ← v )
        message shape: v.      ( shape ← v )
        message print.         ( shape print )
end

class Game.
fields t1 t2 t3 pos.
begin
        message t1.     (^t1)
        message t2.     (^t2)
        message t3.     (^t3)
        message pos.    (^pos)
        message t1: t.  (t1 ← t)
        message t2: t.  (t2 ← t)
        message t3: t.  (t3 ← t)
        message at: p.  (pos ← p)

        { Initialization }
        message init.
        (
            t1 ← Tower new init at: (Point new x: (pos x+3);  y:(pos y+1)).
            t2 ← Tower new init at: (Point new x: (pos x+13); y:(pos y+1)).
            t3 ← Tower new init at: (Point new x: (pos x+23); y:(pos y+1)).
            t1 stack: (Disk new size: 7; shape: "=======").
            t1 stack: (Disk new size: 5; shape: " ===== ").
            t1 stack: (Disk new size: 3; shape: "  ===  ").
            t1 stack: (Disk new size: 1; shape: "   =   ").
            self draw
        )

        message draw.
        (
            pos moveto.
            "------------------------------------" print.
            t2 draw.
            t3 draw
        )

        message play.
```

```
    (
        self move: t1 height from: t1 to: t2 via: t3.
        pos moveto.
        "\n" print
    )

    message move: n from: a1 to: a2 via: a3.
    (
        if n <> 1 then self move: n−1 from: a1 to: a3 via: a2.
        a2 stack: (a1 remove).
        if n <> 1 then self move: n−1 from: a3 to: a2 via: a1
    )
end

{ main program }

begin
    g ← Game new.
    user ← Userview new.
    g at: ( Point new x:10; y:10 ).
    user clear.
    user home.
    { Play it 50 times to get some reasonable timing statistics }
    for i←1. i<=50. i←i+1 do begin
        g init.
        g play.
        "%d\n" print: i
    end.
    { Now test out the memory allocator. }
    { After this statement, the number of objects created }
    { and the number of objects freed should be the same. }
    user ← g ← i ← nil
end
```