CMU-CS-83-150

Generating Compact Code for Generic Subprograms

Jonathan Rosenberg

August 1983

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

Generating Compact Code for Generic Subprograms

Jonathan Rosenberg

August 1983

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Jonathan Rosenberg was also supported by a Fannie and John Hertz Foundation fellowship.

Abstract

This dissertation explores the generation of compact code for generic subprograms in strongly-typed, compiled programming languages. The abstraction power of a conventional subprogram, which may be parameterized by objects, can be increased by making the subprogram *generic*. A generic subprogram is parameterized by types, and may specify its operations independently of the types of its arguments. Unfortunately, the addition of type parameters presents implementation difficulties because a subprogram now seems to operate on objects whose types are unknown until run time.

Generic expansion is an implementation technique for generic subprograms that creates a copy of the subprogram for each set of parameter types. Used in isolation, this method sometimes produces unnecessarily large object programs. To enable a fair test of the utility of generic subprograms, programmers must perceive the implementation of these subprograms as efficient. Otherwise the use of a language will be contorted as programmers attempt to avoid suspected inefficiencies.

By focusing on generic subprograms this dissertation takes the first step towards the generation of compact code for generic facilities, which allow definitions other than just subprograms to be parameterized by types. This research shows that a compiler can do better by using techniques beyond generic expansion as translation methods for generic subprograms. Smaller object code is obtained with a new translation technique, *polymorphic routine translation*. An experimental compiler built to compare the translation techniques, demonstrates that there are programs for which polymorphic routine translation yields a significant reduction in object code size and in compilation time.

The dissertation describes the translation schemes in detail and discusses the implementation of a compiler. The research also led to the discovery of a statistic that predicts the efficacy of each translation method on a test program. This finding is important because there are programs for which generic expansion requires less compilation time than polymorphic routine translation.

Acknowledgments

Department of Computer Science
for being the best place in the world to do computer science for the six years I was there;

Mary Shaw for guidance through all six of those years and especially for her help during the two years I spent working on this dissertation; she not only provided technical advice but, perhaps more importantly, helped me deal with the myriad non-technical considerations required to complete a Ph.D.;

Bill Wulf for being on my committee and taking the time to delve into the technical details of the work; Bill had a large influence on much of the work, but especially on the introduction to the dissertation and the treatment of parameter passing in chapters 5 and 6; Bill has also been somewhat of a spiritual leader for me in the field of compiler construction;

Jon Bentley for being on my committee; Jon greatly aided me in the finer points of my writing style and also took a close look at appendix A and the details of the proof therein;

Dave Fisher for being my outside reader and making sure that my view of things was not stilted by the academic environment in which this work was done;

Bob "Lundar" Hon & Pete "Burned-Man" Schwarz
for being close friends and putting up with me for six years; without these guys, life would have been unbearable;

Mark "the BF-ing German" Sherman, Andy "US out of El Salvador" Hisgen &Mahadev "Banana" Satyanarayanan
for friendship and many discussions about technical issues; Andy actually took the time to help me debug the BLISS-36 code output by the experimental compiler (using DDT)—a true friend;

Mom for giving me her professional editing advice on the final draft of this dissertation;

Bob Seilhamer for being my counselor and good friend; with Bob's help, I've grown a lot;

Sharon Mento, Marrisa "Queen Bee" Mento & Melanie "Tiny Highny" Mento
for being part of my life and teaching me about love, caring and fun (especially hide-and-go-seek); I could never repay these three, but it sure is going to be fun trying;

The Cramps, Dead Kennedys & MDC
for raucousness, rebellion and a general escape from the world and authority.

Gonna get my Ph.D.,
I'm a teenage lobotomy

— The Ramones

Table of Contents

Part One 1

Preliminaries 3

1. Introduction 5

1.1. Generic Facilities 5
1.1.1. A Simple Example 6
1.1.2. Generic Instantiation 7
1.1.3. Kinds of Generic Facilities 9
1.2. Goals of the Dissertation 11
1.3. The Organization of this Dissertation 12
1.4. The Intermediate Language: IL 12
1.4.1. IL 15
1.4.2. Descriptors 17
1.4.3. Objects 18
1.4.4. An Example of Translation to IL 19
1.4.4.1. The Source Language Program: `Sort` 20
1.4.4.2. IL Translation of `Sort` 21

2. Previous Work 21

2.1. Implementation of Generic Facilities 22
2.1.1. The Work of Standish 23
2.1.2. EL1 25
2.1.3. Alphard 26
2.1.4. CLU 27
2.1.5. Miscellaneous Implementations 27
2.1.5.1. Euclid 28
2.1.5.2. Model 28
2.1.5.3. The Work of Gehani 28
2.1.5.4. C 28
2.2. Space-Compacting Compilers 28
2.2.1. PQCC 29
2.2.2. ECS 29
2.3. Inline Substitution 31
2.3.1. The Work of Scheifler 31
2.3.2. The Work of Ball 32
2.3.2.1. The Time-Space Tradeoff 33
2.3.2.2. Transformation Selection 33

3. The Source Language 33
3.1. The Base Language 33
3.2. General Semantics 36
3.2.1. Packages 36
3.2.2. Subprograms 37
3.2.2.1. Selectors 38
3.2.2.2. Overloading 38
3.2.2.3. User-Definable Operators 39
3.2.3. Types 40
3.2.3.1. Discrete Types 40
3.2.3.2. Array Types 40
3.2.4. Attributes: FIRST, LAST, SUCC and PRED 41
3.2.5. Separate Compilation 42
3.3. The Generic Facility 43
3.3.1. Generic Subprograms 43
3.3.2. Generic Instantiation 43
3.3.3. Generic Parameters 44
3.3.3.1. Generic Type Parameters 44
3.3.3.2. Generic Subprogram Parameters 46
3.3.3.3. Generic Object Parameters 47

Part Two
Translating Generic Subprograms 49

4. Generic Expansion: Description and Implementation 51
4.1. Implementing GE 52
4.2. The Advantages of GE 54
4.3. The Disadvantages of GE 55

5. Polymorphic Routine Translation: Description 57
5.1. Instances as Subprogram Invocations 59
5.1.1. The Translation of Subscripting: Selectors 61
5.1.2. Variable Declarations 62
5.1.3. Parameter Passing 62
5.1.4. Function Return 63
5.2. Composite Actions are Generic 64
5.3. Primeval Subprograms 66
5.4. Instantiating Composite Actions 66

6. Polymorphic Routine Translation: Implementation 69
6.1. Translating Variable Declarations 70
6.2. Translating the Generic Body 72
6.3. Translating the Generic Instantiation 74
6.4. Translating Parameter Passing 75
6.4.1. Passing Parameters of a Generic Type 77
6.4.1.1. Interface Points 79
6.4.1.2. Interface Points in a Primeval Subprogram 81
6.5. Translating Function Return 82
6.6. Type Parameters 82
6.7. Subprogram Parameters 83

6.8. Object Parameters 85
6.9. An Example: The PR Translation of Concat 86
6.10. Advantages of PR 89
6.11. Disadvantages of PR 90
7. Optimizations **91**
7.1. Constant Folding 93
7.2. Routine Deletion 94
7.3. Inline Substitution 94
7.3.1. Determining the Invocations to Expand 95
7.3.2. Inline Expansion: The Basic Schema 98
7.3.3. Optimizations to the Schema 100
7.4. Improvement from Optimizations: An Example 102
7.4.1. The Optimization of Concat 104
8. The Experimental Compiler **107**
8.1. The Front End 109
8.1.1. The FE Phase 109
8.1.2. The COMPILE Phase 110
8.2. The Translator 111
8.2.1. The INFO Phase 112
8.2.2. The TRANSLATE Phase 115
8.2.2.1. GE 115
8.2.2.2. PR 116
8.2.3. The OPT Phase 116
8.2.4. Determining the Invocations to Expand 118
8.2.5. The CODE Phase 120
8.3. The Back End 121
8.4. The Run-time System 121

Part Three
Evaluation and Conclusions **123**

9. Test Programs, Statistics and Evaluation **125**
9.1. Test Programs 126
9.2. Notation 128
9.3. Object Code Size 129
9.4. Compilation Time 131
9.5. A Hybrid Translation Strategy 132
9.6. The Statistic ρ 133
9.7. Execution Time 137
9.8. Inline Substitution 139
9.9. Summary 141
10. Conclusions **143**
10.1. Using PR and GE 145
10.2. Directions for Further Research 146
10.2.1. Generic Packages 146
10.2.2. Operational Equivalence 148
10.2.3. Range Constraints and Constraint Checking 149
10.2.4. Unused Parameter Removal 151

10.3. Summary 152

References 158

Appendix A. Complexity of Inline Substitution 161

A.1. The Language L 162
A.2. The Transformations 163
A.2.1. Inline Expansion 165
A.2.2. Constant Folding 165
A.2.3. Routine Deletion 166
A.3. Statement of the Problem 166
A.4. The Proof 16
A.4.1. 3-SAT 16
A.4.2. The General Idea 16
A.4.3. Literals 16
A.4.4. Clauses 17
A.4.5. Putting it All Together 17
A.5. Inline Substitution and NP-completeness 1

Appendix B. Package STANDARD 17

Appendix C. The Run-time System 18

Appendix D. The Size Estimation Table 1

Appendix E. Compiler Phase Timings 1

List of Figures

Figure 5-1: The Generic Function Concat 57
Figure 5-2: Translation of Concat into Composite Instances, I 60
Figure 5-3: Translation of Concat into Composite Instances, II 65
Figure 5-4: Primeval Subprograms, Part I 67
Figure 5-5: Primeval Subprograms, Part II 68
Figure 7-1: Inline Expansion Schema 99
Figure 7-2: Inline Expansion Loop 102
Figure 8-1: The Parts of the Experimental Compiler 107
Figure 8-2: The Front End 110
Figure 8-3: The Translator 112
Figure 8-4: Primary Optimization Loop 117
Figure 8-5: Inline Substitution Loop 117
Figure 8-6: Routine Deletion Loop 118
Figure 8-7: Size Estimation Routine 119
Figure 9-1: $\rho(P)$ versus $sr(P)$ 136
Figure 9-2: $\rho(P)$ versus $cr(P)$ 137
Figure 9-3: $\rho(P)$ versus $er(P)$ 138
Figure A-1: Attribute Grammar for L, Part I 163
Figure A-2: Attribute Grammar for L, Part II 164
Figure A-3: Mapping from Variables in U to Functions 169

List of Tables

Table 1-1:	IL Operators (decreasing precedence)	13
Table 1-2:	IL Operators Represented by Invocation	15
Table 1-3:	Class of Objects Specified by Descriptors	16
Table 1-4:	Kind Dependent Descriptor Fields	16
Table 1-5:	Operators Applicable to Objects	16
Table 3-1:	Generic Type Parameters	46
Table 5-1:	Composite Actions for Types	68
Table 6-1:	Bindings for Non-generic Types	78
Table 6-2:	Bindings for All Types	79
Table 6-3:	Parameter Conversions	81
Table 7-1:	Substitutions for Inline Expansion Schema	99
Table 8-1:	Run-time Modules	122
Table 9-1:	Descriptions of Test Programs	127
Table 9-2:	Statistics of Test Programs	128
Table 9-3:	Notations for Measurements	128
Table 9-4:	Object Code Size of Test Programs (words)	129
Table 9-5:	Compilation Times of Test Programs (seconds)	131
Table 9-6:	$\hat{ge}$, $\hat{pr}$ and ρ for Test Programs	135
Table 9-7:	Execution Times of Test Programs (milliseconds)	138
Table 9-8:	Improvement of GE Object Code Size Due to IS	140
Table 9-9:	Improvement of PR Object Code Size Due to IS	140
Table D-1:	Size Estimation Table	186
Table E-1:	Phase Timings for GE	188
Table E-2:	Phase Timings for PR	188

Part One

Preliminaries

Chapter 1
Introduction

This dissertation explores the generation of compact code for generic subprograms in strongly-typed, compiled programming languages. Generic subprograms are an important part of a generic facility, which is one of the most recent and most touted abstraction tools for programming. Over the years programming language designers increasingly have been concerned with the importance of *abstraction*, a term that the cosmologist J. Callahan defines succinctly in a paper about the curvature of the universe [Callahan 77]:

> *In mathematics the job of isolating what is relevant is called abstraction.*

Programming languages have always contained some notions of abstraction—subprograms have been around since the dawn of high-level languages. Though invented to save space, subprograms since have been recognized as an abstraction tool. A subprogram defines an action, which subsequently may be invoked by mention of the subprogram's name. For example, a subprogram named `Sort` that sorts the elements of the globally defined array `A` would have the specification

```
procedure Sort;
```

A parameterless subprogram, like `Sort`, is a weak abstraction because the action it defines is specific to particular objects mentioned in its body. A subprogram can be abstracted further by allowing it to be *parameterized* with objects. The subprogram can now specify its action independently of objects on which it will operate. The subprogram `Sort` can be parameterized so that it will sort any array of integers:

```
procedure Sort(A : array of Integer);
```

A subprogram with parameters is less specific than a parameterless subprogram. This ability to avoid specificity and isolate what is relevant—the algorithm—makes parameterization a crucial aspect of abstraction. But parameterizing subprograms with objects was just the first step in exploring the power of parameterization. An advance was made with the discovery of the *generic facility*, which allows definitions to have parameters other than objects.

A common example of the parameterization allowed by a generic facility, and the one of concern in this dissertation, is the *generic subprogram*. A generic subprogram, which is parameterized by types as well as objects, has greater abstraction capability than a non-generic subprogram. This is because a generic subprogram defines its action independently of the types of the objects it manipulates. This can be done by parameterizing a subprogram with types to obtain a generic subprogram. The procedure `Sort` can now be defined to operate on any array by making it a generic procedure, as follows:

```
generic
   type T;
procedure Sort(A : array of T);
```

Unfortunately, the addition of type parameters presents implementation difficulties because the body of `Sort` manipulates objects in ways that depend on their types. Assignment, for example, is a manipulation whose operation depends on the type of the objects involved. The problem is that some of the objects may be of type `T`, which is a parameter and not known until `Sort` is invoked. This makes it tricky to generate code for a generic subprogram.

There is an obvious implementation technique for generic subprograms, known in this dissertation as *generic expansion*. This method creates a customized copy of a generic subprogram for each possible set of type parameters. This solves the problem, because each subprogram now operates on objects with fixed types. Generic expansion, however, may lead to unnecessarily large object programs.

Although generic facilities have been in vogue for some time, little work has been done on implementation techniques. There are, thus, few compilers for languages with generic facilities and little experience to support the current fascination with this abstraction tool. To judge their usefulness it will be necessary to build compilers for languages with generic facilities. It is crucial that the implementations have no obvious inefficiencies, because programmers quickly learn a compiler's weak points and avoid the language features that exercise these flaws.

By focusing on generic subprograms, this dissertation begins the exploration into the efficient implementation of generic facilities. The research described in this work began with the detailed description of *polymorphic routine translation*, which avoids the profuse replication of code that generic expansion entails. To determine the usefulness of this scheme a compiler was constructed that implements both this technique and the generic expansion method. The two techniques were executed on a set of test programs, and statistics were collected to allow a comparison of the two methods. The statistics measure the object code size, compilation time and execution speed of the

test programs. The results demonstrate that for some programs polymorphic routine translation produces smaller object code and requires less compilation time than generic expansion. In addition, a source program measure was developed that enabled the separation of those test programs that should be compiled by the new method from those programs that will do better with generic expansion. This measure presents hope that a simple, widely applicable measure can be found to determine the appropriate translation method for a program.

The remainder of this chapter presents an introduction to generic facilities (section 1.1), a discussion of the goals of the dissertation (section 1.2), suggestions on how to read this dissertation (section 1.3) and a primer on the intermediate language used in the experimental compiler (section 1.4).

1.1. Generic Facilities

This section introduces generic subprograms by giving an example of the definition and use of a generic subprogram (section 1.1.1) and defining the semantics of these subprograms (section 1.1.2). A discussion of the kinds of generic facilities that are of interest to this dissertation appears in section 1.1.3.

The examples in this chapter and chapter 2 are written in a notation much like the language Ada [DOD 82]. The examples should be easily understood by the reader, who should consider the notation to be descriptive. The source language on which this work is based is derived from Ada and is described in chapter 3.

1.1.1. A Simple Example

This section develops a definition for a generic subprogram that exchanges the values of two variables of identical type. A procedure to perform this action on two variables of type T is

```
procedure Exchange(X, Y : T) is
   Temp : T;
begin
   Temp := X;
   X := Y;
   Y := Temp;
end Exchange;
```

This procedure will exchange any two variables of type T. A small change allows the subprogram to be further parameterized so that it will exchange two variables of any type:

```
generic
   type T;
procedure Exchange(X, Y : T) is
   Temp : T;
begin
   Temp := X;
   X := Y;
   Y := Temp;
end Exchange;
```

Preceding the procedure definition is a generic header, consisting of the keyword **generic**, followed by a list of *generic formal parameters*, which indicate how the subprogram is parameterized. In this case, there is a single generic formal parameter

```
type T;
```

that indicates that T is a *generic formal type parameter.*

1.1.2. Generic Instantiation

The definition of `Exchange` above is a template for declaring procedures. It may not be invoked directly; rather a *generic instantiation* must be used to create a procedure from this template. For example, a procedure named `Int_Swap` is created by the generic instantiation

```
procedure Int_Swap is new Exchange(Integer);
```

The generic instantiation specifies a parenthesized list of *generic actual parameters*, which appears after the name of the template. There is a single generic actual parameter, `Integer`, which corresponds to the generic formal parameter `T`.

The effect of a generic instantiation is determined by performing *generic expansion.*[1] As a first approximation, this technique can be applied by making a copy of the generic subprogram that is the target of the instantiation. This copy is then modified by replacing each occurrence of a generic formal parameter by its corresponding generic actual parameter. The semantics of the generic instantiation is the semantics of the subprogram obtained by this process.

For example, applying generic expansion to the instantiation of `Int_Swap` above yields

---

[1] For a different view of the semantics of a generic facility see the paper by the ADJ group [Goguen 78].

```
procedure Int_Swap(X, Y : Integer) is
   Temp : Integer;
begin
   Temp := X;
   X := Y;
   Y := Temp;
end Int_Swap;
```

The simple textual substitution suggested above is not sufficient in general. One difficulty is that name conflicts may invalidate the transformation. This problem is solved easily by appropriate use of renaming, and it is assumed that the reader can perform the necessary substitutions. A second problem occurs if a generic actual parameter has a side effect and the corresponding generic formal parameter is referenced more than once within the generic subprogram. Naive expansion would lead to multiple evaluations of the parameter. But the semantics of GE calls for a generic actual parameter to be evaluated once, at instantiation time. The side-effect problem is common in code generation; a solution is discussed in section 4.1.

1.1.3. Kinds of Generic Facilities

For this dissertation a *generic facility* is any scheme that allows a subprogram to be parameterized by types. The existence of type parameters is crucial because types are a powerful, useful form of parameterization and type parameters present interesting implementation problems.

Many languages with generic facilities provide additional kinds of parameterization besides the minimum requirements imposed by the definition above. A language may allow the parameterization of definitions other than subprograms. For example, it may be possible to parameterize type definitions, object definitions or encapsulation definitions. Encapsulation definitions, which provide a convenient means of packaging related definitions, are especially interesting entities for parameterization. This is because the parameterization of an encapsulation can define an abstract data type, an idea that has been praised as highly as generic facilities. Generic encapsulation definitions are interesting, but are beyond the scope of this dissertation.[2]

Another way in which languages modify a basic generic facility is to allow kinds of parameters other than types. It is possible to allow definitions to have object and subprogram parameters, for example. Object and subprogram parameters for subprograms seem uninteresting because non-generic subprograms in many languages allow these kinds of parameters. But parameterization by objects and

[2] But see section 10.2.1 on generic packages.

subprograms is useful in conjunction with encapsulations, which normally have no parameters. To get some insight into the problems of implementing generic packages, the source language used in this dissertation allows parameterization of generic subprograms with objects and subprograms.

In addition to the parameterization provided, generic facilities can be classified according to the earliest time at which the types within generic instantiations can be determined. Types may be determined at compile time, link time or run time. In general, if the semantics of a language requires types to be determined at run time, then the language must be interpreted. This is because it is difficult to generate reasonable code to manipulate objects whose types are unknown until run time. If type determination may be performed at compile time or link time, then the language may be compiled since the types of all objects are known prior to code generation. (This generalization should not be applied too literally, but it provides a useful rule of thumb.)

There are several ways that a generic facility may introduce the need for run-time type binding. Allowing a generic instantiation to contain an instantiation of itself is one way; the following example requires run-time binding of a type for this reason.

```
generic
   type T;
procedure P(N : Integer) is
   type T_New is array(1..5) of T;
begin
   if N > 0 then
      declare
         procedure P_New is new P(T_New);
      begin
         P_New(N-1);
      end;
   end if;
end P;

procedure Loop is new P(Character);
```

The generic procedure P contains an instantiation of itself. At run time, T_New must be bound to the sequence of types

```
array(1..5) of Character
array(1..5) of array(1..5) of Character
array(1..5) of array(1..5) of array(1..5) of Character
. . .
```

Such recursive use of generic instantiations may force the binding of some types at run time.[3] This

[3] Gehani has shown [Gehani 80] that it is undecidable whether a recursive generic instantiation will require run-time binding of types.

dissertation is concerned with generic facilities whose bindings may occur before run time and may, thus, be compiled comfortably. The source language in this dissertation forbids a generic subprogram to instantiate itself and thus avoids recursive generic instantiations.[4]

1.2. Goals of the Dissertation

Generic facilities show great promise for easing the program development and maintenance cycle, but usable implementations are required to see if the potential will be fulfilled. A useful generic facility would provide for the definition of a wide variety of generic entities: abstract data types, subprograms and iterators,[5] at the very least.

An implementation of a full generic facility is a tall order, so this dissertation chips off a manageable piece of the problem: to demonstrate a technique for generating compact code for generic subprograms. The generic expansion method for translating generic subprograms is well known and, therefore, provides a useful benchmark. To be successful the new technique must produce code that is significantly more compact than the code produced by generic expansion.

The efficiency of object code is judged generally by some function of the space occupied by the code and data and the average or worst-case time the code requires to execute. A problem with execution time is its sensitivity to many things: paging behavior, input data and machine load, for example.

For this dissertation the measure of efficiency is space, primarily because it can be treated as a static measure insensitive to its environment. In addition, Knuth has shown [Knuth 71] that a typical program spends over 50% of its execution time in about 4% of its code. This being the case, it makes sense to use as little space as possible for the 96% of a program that is executed rarely. Even if the speed of this space-minimized code is decreased, there will be little effect on overall execution time.

Because space is the optimization criterion for this work, the kinds of space to measure must be determined. A compiled language generally has the following domains that require space:

- code;
- static data (literals, branch tables);
- stack-based data (objects local to a block or subprogram);

---

[4] The language Ada [DOD 82] has also chosen this approach.

[5] An iterator is a control abstraction that defines a general looping mechanism. For a discussion of iterators see chapter 3 of the book by Hibbard et al. [Hibbard 83].

- heap-based data.

The code and static data spaces should be measured, as the sum of their sizes represents the minimum space that a program requires. It is not obvious whether to include stack-based and heap-based data spaces. The generation of an extra local variable by the generic translation mechanism may require no additional stack space.[6] The same may hold for heap objects. Thus, the space measure for this work consists of the sum of the sizes of the code and static data spaces.

The technique suggested in this dissertation must do more than just produce less code than generic expansion. The new method also must be applicable to the whole class of compiled languages with generic subprograms. I have restricted myself to this class because I believe it contains those languages that are the most useful for building large systems and these systems are likely to benefit greatly from a generic facility.

In addition to wide source language applicability, the translation method must be amenable to produce code for multiple target machines. To accomplish this, the implementation should consult external files or tables for machine-dependent information. It should also be possible to change implementation decisions—such as run-time representations for objects or parameter binding methods—without serious repercussions to the translation scheme. This is an elusive goal, depending largely on intuition.

A translation scheme may do a nice job on a particular class of programs by forcing programs outside of that class to suffer. Since not every program will make extensive use of a generic facility, it would be unacceptable for the translation technique to cause such a distributed penalty. The technique must make good on the generic subprograms in a program without affecting the quality of code for the rest of the program.

This dissertation is not concerned with the semantic analysis of generic facilities. This does not imply that there are no interesting issues in semantic analysis; it is simply outside the purview of this research.

---

[6] A standard stack maintenance technique is to overlay the stack space used by blocks that occur at the same static nesting level. This technique is described in chapter 14 of the book *Compiler Construction for Digital Computers* [Gries 71].

1.3. The Organization of this Dissertation

This dissertation is divided into three parts. Part one is composed of chapters 1, 2 and 3 and provides introductory information. Chapter 1 introduces generic subprograms and discusses the goals of the dissertation. This chapter also contains a description of the intermediate language used in the experimental compiler. The language, IL, is used throughout part two of the dissertation to demonstrate translation techniques.

Chapter 2, which is not crucial to understanding the dissertation, describes research that influenced the work on implementation of generic subprograms. The source language in which generic subprograms are written is described in chapter 3. This language must be understood if the reader is to make sense of the bulk of this work. The reader should consult the introduction to that chapter for advice on the sections to read.

Part two contains the heart of the research: a presentation of the techniques used to produce compact code for generic subprograms. Chapters 4 through 6 describe the basic translation techniques. The optimizations applied to the results of the basic methods are described in chapter 7, and some details about the experimental compiler are given in chapter 8. The casual reader is advised to at least read chapters 4 and 5 to obtain some understanding of the two translation techniques.

Part three should be read by anyone who opens this dissertation. Chapter 9 presents statistics about the translation methods. These statistics, accompanied by a discussion of their import, were obtained by executing the experimental compiler on a group of test programs. Chapter 10 presents the conclusions of the dissertation and a discussion of suggested topics for further research.

There are five appendixes. Appendix A contains a proof that performing inline substitution for minimal space is NP-hard and, hence, probably impractical for use in a compiler. Appendix B contains the listing of the predefined package STANDARD that defines the standard environment in which programs are compiled. Appendix C gives a listing of most of the routines that make up the run-time system for executing programs. Appendix D contains the table used in the experimental compiler to estimate the size of expressions and statements during inline substitution. Finally, appendix E provides timings for each phase of the experimental compiler.

1.4. The Intermediate Language: IL

A compiler's task is to translate a source language to a target language. There may be a large difference between the abstraction levels of the source language and target language as, for example, when the source language is Ada and the target language is machine language. Translation may be difficult in this case due to the gap between the complexity of source language operations and target language operations. Translation can be eased by the introduction of an *intermediate language*, which has an abstraction level between the abstraction levels of the source and target languages. With an intermediate language the translation is performed in two steps:

1. Translate the source language program to an intermediate language program.
2. Translate the intermediate language program to a target language program.

The intermediate language allows the translation to be divided into two subtasks, each of which is itself a translation. But the difference between the abstraction levels of successive language pairs has been reduced. If the technique is used properly, then the design effort required for the two translation tasks is less than that required for the original task.

Use of an intermediate language is not crucial to the techniques in this dissertation. In the experimental compiler, however, it was useful to define an intermediate language. The intermediate language, IL, is used extensively to illustrate translation techniques. The remainder of this section provides an introduction to IL, so the reader will be able to understand the examples.

Within the compiler, IL programs are represented by a graph. There is also an external representation for IL so that it may be transmitted between compiler phases via text files. Because this external representation is ugly and difficult to read, IL will be presented in a programming language notation within this dissertation.

1.4.1. IL

The intermediate language IL is a low level language similar to the BLISS family of programming languages [DEC 78a]. Much of IL is standard programming language fare: for example, **if** statements and **while** loops. The intuition and experience of the reader are relied upon for such constructs.

But IL also contains a number of features that may be unfamiliar to many readers:[7]

- IL is expression oriented: most constructs yield a value, and any expression can serve as a statement.
- IL has a typing mechanism that employs *descriptors* (described in section 1.4.2).
- IL always distinguishes explicitly between the *address* of an object and the *value* of an object.[8]
- IL provides the single parameter passing mechanism of call-by-value (described in section 6.4). Other parameter passing methods must be simulated.

Most of the expressions and operators in IL can be understood without explanation. For reference, table 1-1 lists each IL operator and a brief description of its purpose. This table lists only the IL operators that the reader will encounter. The operators are presented in decreasing order of precedence, from the top of the table to the bottom.

Operator	Purpose
.	dereferencing
*, /	integer multiplication, division
+, -	integer addition, subtraction
=, ≠, >, ≥, <, ≤	comparison
and	boolean conjunction
or	boolean disjunction
←	assignment
return	routine return

Table 1-1: IL Operators (decreasing precedence)

The most distinguishing feature of IL is the distinction between the *address* of an object and its *value*. Consider the IL declaration

```
local X, Y : Integer_Desc;
```

This declaration can be thought of as declaring two integer variables X and Y (the precise meaning of this declaration is described in the next two sections). The address of a variable is denoted by its identifier, so the address of the variable X is represented by the expression

---

[7] All of these features, except for data typing, are found in the BLISS family of languages [DEC 78a].

[8] This is in contrast to most programming languages, in which the distinction between address and value is implicit. For example, in Pascal [Jensen 78], the statement

```
A := B;
```

means "copy the value in variable B to the address of variable A."

X

To assign the value of variable Y to variable X the statement following might seem correct:

```
X ← Y;
```

But this statement copies the address of the variable Y to the variable X.[9]

To perform the assignment correctly it is necessary to obtain the value of Y. In IL, the value of a variable is obtained by application of the dereferencing operator, which is represented by ".", pronounced "dot."[10] The proper way to perform the assignment of Y to X is

```
X ← .Y;
```

It is possible to apply the dereferencing operator more than once. For example, after execution of the assignment statements

```
Ref_X ← X;
Ref_Y ← Y;
```

the variable Ref_Y contains the address of Y, and Ref_X contains the address of X. The statement following copies the value of Y to X:

```
.Ref_X ← ..Ref_Y;
```

Explicit dereferencing is also used to simulate parameter passing methods other than call-by-value, the only method provided by IL. This is explained in detail in section 6.4.1 and illustrated in the example at the end of this chapter.

Besides the prefix and infix operators illustrated in table 1-1, IL contains operators that are written as routine invocations. Routine invocation is indicated by the juxtaposition of an expression and a parenthesized list of expressions, as in

```
OutNum(.N/2, 10, 0);
```

Routine invocation has a higher precedence than all of the operators in table 1-1. Table 1-2 provides a list of the IL operators that are represented as invocations.

---

[9] In fact, this statement is illegal because the type of X is integer and not address of integer.

[10] The dot operator is taken from BLISS-10 [Wulf 71].

Notation	Purpose
ref(*value*)	create an object containing *value*; yields address of object
subscript(*array*, *index*)	yields address of $index^{th}$ element of *array*
kind(*descriptor*)	yields kind of *descriptor* (section 1.4.2)

Table 1-2: IL Operators Represented by Invocation

1.4.2. Descriptors

Descriptors serve the same purpose in IL that types serve in most high-level languages: a machine-independent representation for the structure of data. The difference between descriptors and types is that a descriptor is an IL value and may be manipulated by operators.

A descriptor specifies a set of requirements for the object to which it is attached. These requirements describe the kinds of values that the object must be capable of holding. It is the code generator's responsibility to create an object that satisfies these requirements. In addition, a set of operators is provided for manipulating objects in a manner dependent only on the object's associated descriptor.

A descriptor is denoted by the notation

desc(*kind*, *kind-dependent expressions* ...)

The *kind* field, which specifies the *kind* of the descriptor, determines the class of objects the descriptor defines: integer objects, boolean objects or routine objects, for example. The *kind-dependent expressions*, which differ for each descriptor kind, specify auxiliary information about the objects.

For example, the descriptor

```
desc(Integer);
```

has no kind-dependent expressions and specifies an integer object, which may hold a value from an implementation-defined subset of the integers. Integer objects may be manipulated by the ←, + and < operators, as well as others.

Tables 1-3, 1-4 and 1-5 provide a list of descriptor kinds, the class of objects they specify, their kind-dependent fields and the operators applicable to objects whose associated descriptor is that indicated.

An eighth descriptor kind, Union, is used for generic type parameters. The Union descriptor is not

Kind	Class of Objects
Integer	integers
Boolean	boolean values
Enumeration	enumeration values
Array	1-dimensional arrays
Pointer	addresses of objects
Routine	routines
Descriptor	descriptors

Table 1-3: Class of Objects Specified by Descriptors

Kind	Representation
Integer	`desc(Integer)`
Boolean	`desc(Boolean)`
Enumeration	`desc(Enumeration,` *number of values*`)`
Array	`desc(Array,` *index descriptor*`,` *component descriptor*`,` *lower bound*`,` *upper bound*`)`
Pointer	`desc(Pointer,` *descriptor of referenced objects*`)`
Routine	`desc(Routine)`
Descriptor	`desc(Descriptor)`

Table 1-4: Kind Dependent Descriptor Fields

Kind	Operators
Integer	*, /, +, -, =, ≠, >, ≥, <, ≤, ←, **return, ref**
Boolean	=, ≠, >, ≥, <, ≤, **and, or,** ←, **return, ref**
Enumeration	=, ≠, >, ≥, <, ≤, ←, **return, ref**
Array	**subscript, return, ref**
Pointer	., ←, **return, ref**
Routine	←, **ref,** *invocation*
Descriptor	←, **kind, ref**

Table 1-5: Operators Applicable to Objects

shown in the tables but is described in section 6.1. Also not shown in the tables is a second form for an Array descriptor:

`desc(Array,` *index descriptor*`,` *component descriptor*`)`

This form, which omits the *upper bound* and *lower bound* fields, may be associated only with a formal parameter. It specifies that the parameter will be bound to an array object with the indicated index and component descriptors. Because no bounds are specified, this allows the parameter to be bound to any array of the appropriate type, regardless of its bounds.

There is an operator, **kind**, that is applicable to any descriptor value. The **kind** operator yields the value of the *kind* field of its argument. The expression

```
kind(desc(Array, Integer_Desc, Character_Desc))
```

yields the value `Array`. The value yielded by the **kind** operator has a type that has no name in IL, but there is a literal of this type for each descriptor kind. The literals are denoted `Integer`, `Boolean`, `Enumeration`, `Array`, `Pointer`, `Routine` and `Descriptor`. The **kind** operator may be used only as an operand of the = operator:

```
if kind(.D1) = kind(.D2) then ...
```

or as the selection expression in a **case** expression:

```
case kind(.Arr) of
   . . .
```

A kind literal may be used as an operand of the = operator:

```
return kind(.D) = Integer
```

or as a case arm value in a **case** expression:

```
case kind(.T) of
   when Integer => ...;
   when Array => ...;
esac;
```

Besides being used to describe an object at compile time, an IL descriptor may be used as the run-time representation of a generic type parameter. This is described in section 6.6.

1.4.3. Objects

As most programming languages, IL allows an identifier to be bound to a typed object, but in IL descriptors serve the purpose of types. An object may contain a value if the descriptors associated with the object and the value are identical. A common way to bind an identifier to an object is by a local declaration:

```
local X : desc(Integer);
```

This declaration defines an identifier `X` that is local to the immediately surrounding block or routine. This identifier will be bound to the address of a new object that satisfies the requirements specified by the descriptor **desc(Integer)**. The expression

```
X
```

denotes the address of the object, while the expression

```
.X
```

denotes the value of this object.

A local declaration creates an object that is modifiable; its value may be changed. It is possible to bind an identifier to a constant object that may not be modified. This is done by a constant declaration:

```
constant Factor : desc(Integer) = .Base/2;
```

When executed, this declaration causes the identifier `Factor` to be bound to an object containing the value of the expression `.Base/2`. In this case, the identifier denotes the object directly so that the expression

```
Factor
```

denotes `.Base/2`

In addition to local and constant declarations, IL provides formal parameter declarations. For example, in the routine declaration

```
routine BLT(From, To : desc(Pointer, Integer)) = ...
```

the identifiers `From` and `To` are defined to be formal parameters. Unlike a local or constant declaration, no new object is created for a formal parameter declaration. Instead, the declaration is used to define the type of object to which the formal parameter will be bound during execution of the routine. Call-by-value is the only parameter binding method supported in IL, so other parameter binding methods must be simulated. The call-by-value method and the simulation of other methods are discussed in section 6.4.

When it is convenient and no confusion may result, the descriptors are omitted from constant declarations. For example, in the declaration

```
constant LB = 2;
```

it is clear that the descriptor for `LB` is an integer descriptor.

1.4.4. An Example of Translation to IL

This section gives an example of the translation to IL of a source language fragment. Although the source language is not described until chapter 3, the reader should understand enough of the program to make the example useful.

Of special interest in this example are the simulation of call-by-reference for parameters A and Min, the variable declarations for I, J and Temp, and the use invocations of the routine Subscript on the left-hand-side of assignment statements.

1.4.4.1. The Source Language Program: Sort

```
type Table is array(Integer range <>) of Integer;

--
-- Sort array A, from component Start to A'LAST.  Place minimum
-- component in parameter Min.
--

procedure Sort(A : in out Table;
               Start : in Integer;
               Min : out Integer) is

   I, J : Integer;

begin

   I := Start;
   Min := A(Start);

   while I < A'LAST loop
      J := I + 1;
      while J <= A'LAST loop
         if A(I) > A(J) then
            declare
               Temp : Integer;
            begin
               Temp := A(I);
               A(I) := A(J);
               A(J) := Temp;
            end;
         end if;
         J := J + 1;
      end loop;

      I := I + 1;
      if A(I) < Min then
         Min := A(I);
      end if;
   end loop;

end Sort;
```

1.4.4.2. IL Translation of Sort

```
constant Integer_Desc = desc(Integer);

constant Table_Desc = desc(Array, Integer_Desc, Integer_Desc);

                              ! Pointers simulate call-by-reference
routine Sort(A : desc(Pointer, Table_Desc),
             Start : Integer_Desc,
             Min : desc(Pointer, Integer_Desc)) =

begin
   local I, J : Integer_Desc;

   I ← .Start;
   .Min ← .subscript(.A, .Start);

   while .I < .(.A+UB_Offset) do     ! This yields the upper
                                     ! bound stored in the
                                     ! array dope vector at
                                     ! UB_Offset

      J ← .I + 1;
      while .J ≤ .(.A+UB_Offset) do
         if .subscript(.A, .I) > .subscript(.A, .J) then
            begin
               local Temp : Integer_Desc;
               Temp ← .subscript(.A, .I);
               subscript(.A, .I) ← .subscript(.A, .J);
               subscript(.A, .J) ← .Temp
            end
         fi;
         J ← .J + 1
      od;

      I ← .I + 1;
      if .subscript(.A, .I) < ..Min then
         .Min ← .subscript(.A, .I)
      fi

   od

end Sort;
```

Chapter 2
Previous Work

This chapter discusses research that has influenced the development of this dissertation. The projects are drawn from three areas of compiler research: implementation of generic facilities, code space optimization and inline substitution.

Section 2.1 discusses work on the implementation of generic facilities. Although some of the facilities presented are beyond the scope of this dissertation, they are relevant to a general discussion because many of the techniques used are analogous to techniques in the dissertation.

The PQCC (Production Quality Compiler Compiler) and ECS (Experimental Compiler Systems) projects are concerned with producing code-space efficient compilers. Because this dissertation investigates generation of compact code, it is important to see the approaches of these two projects. PQCC and ECS are discussed in section 2.2.

The use of inline substitution as a code-space reduction technique is crucial to the success of implementing generic subprograms. Section 2.3 provides a synopsis of two important investigations into inline substitution.

Each language discussed in this chapter has its own syntax, which is irrelevant for our purposes. The program fragments in this chapter, therefore, are presented in the notation used in the previous chapter.

2.1. Implementation of Generic Facilities

This section discusses several implementations of generic facilities that have contributed ideas to this dissertation. The discussion covers the facets of each implementation relevant to this work—the section does not provide a comparison of similar features across the languages involved.

2.1.1. The Work of Standish

In his Ph.D. dissertation [Standish 67], Standish investigated the implementation of a generic facility that was grafted on to the language Algol 60 [Naur 63]. Standish modified Algol 60 by promoting types to a position of equal status with other Algol values (for example, integers, reals and arrays). This extension is extremely powerful since it allows the programmer to define and manipulate types at run time. But this power imposes a burden on the execution of the language because objects can be manipulated whose types are unknown until run time.

Standish coped with run-time types by using a run-time *descriptor* to represent a type. The descriptor contained enough information to determine storage allocation and manipulation of an object. This prescription would describe adequately the descriptors used in this work (section 1.4.2).

The primary method suggested in the current dissertation to save space is to share a single code body among several generic instantiations. The shared code body manipulates objects whose types are not known until run time—the same problem faced by Standish. And as in Standish's work, the technique suggested in the current work uses descriptors to cope with this problem.

In Standish's language the ability to define and manipulate types at run time was provided by primitive operations over descriptors. The programmer defines a type by constructing its descriptor, which is used by the compiler and run-time system to perform type-dependent actions.

For example, a (descriptor for a) type representing complex numbers is constructed by the following program fragment:

```
Complex : descriptor;                    -- Complex is a variable
                                         -- of type descriptor

Complex := [RealPart : Real | ImagPart : Real];

C1, C2 : Complex;     -- Storage allocated according to descriptor
```

The expression

```
[RealPart : Real | ImagPart : Real]
```

yields a descriptor describing a type with values comprising two components: `RealPart`, which is of type `Real`, and `ImagPart`, also of type `Real`. `Real` is a predefined type descriptor that defines a type whose values form a subset of the real numbers.

Such a type facility provides, at least functionally, the necessary elements of a generic facility. For example, one could define a generic sort procedure as follows:

```
procedure Sort(Elt : descriptor; A : Sequence(Elt);
               Compare : function(X, Y : Elt) return Boolean) is
begin
   for I from A'FIRST to A'LAST-1 loop
      for J from I+1 to A'LAST loop
         if not Compare(A(I), A(J)) then
            Swap(Elt, A(I), A(J));
         end if;
      end loop;
   end loop;
end Sort;
```

(The procedure Swap, invoked within the inner **for** loop, is a generic procedure that is assumed to be defined elsewhere.) A call of `Sort` might be

```
Names : Sequence(String);

. . .                -- Initialize Names
Sort(String, Names, String_Less_Than);
```

2.1.2. EL1

The EL1 language [Wegbreit 72, Wegbreit 74] extended Standish's work by according full language status to types. The EL1 designers defined a run-time type mechanism with greater expressive power than Standish's type system and used then current software-engineering techniques to aid in the language design.

The increased power of the EL1 type facility means that an EL1 program must be interpreted. Nonetheless, the EL1 designers developed a compiler to make it possible to obtain efficient code for parts of an EL1 program. The compiler is only capable of generating code for program segments whose type information can be determined at compile time. It is up to the user to inform the compiler of the program parts to be compiled.

As with Standish's language, the power of the type mechanism in EL1 obviates the need for a special generic facility. For example, the following generic function adds two values of types `Integer`, `Real` or `Complex`:

```
function Plus(X, Y : any) return any is
begin
   if Type(X) = Integer then
      if Type(Y) = Integer then
         return Fix_Add(X, Y);
      elsif Type(Y) = Real then
         return Float_Add(Float(X), Y);
      else
         return Complex(Plus(X, Y.Re), Y.Im);
      end if;
   elsif Type(X) = Real then
      . . .
   end if;
end Plus;
```

Subprograms written using this paradigm are difficult to write and to understand due to the nested control structure. In addition, it is tedious for the compiler to exploit knowledge about the types of actual arguments in a call to `Plus`. For example, the compiler could transform the call

```
Plus(A, 3)
```

into the call

```
Fix_Add(A, 3)
```

if `A` is of type `Integer`, but this would require compile-time interpretation of the body of `Plus`.

For these reasons, EL1 provides a construct for defining generic subprograms, called the **generic** statement. Using the **generic** statement, `Plus` would be rewritten as follows:

```
function Plus(X, Y : any) return any is
begin
   generic (X, Y) of
      when [Integer, Integer] =>
         return Fix_Add(X, Y);
      when [Integer, Real] =>
         return Float_Add(Float(X), Y);
      when [Integer, Complex] =>
         return Complex(Plus(X, Y.Re), Y.Im);
      when [Real, Integer] =>
      . . .
   end generic;
end Plus;
```

This allows the compiler to take explicit advantage of the generic structure of the subprogram.

The **generic** statement has influenced the construction of the primitive actions in this dissertation (section 6.2). As in EL1, the motivation was to enable the compiler to take advantage of type information.

2.1.3. Alphard

Although the Alphard language [Shaw 81] was not implemented, it inspired much of the work on the design of generic facilities. In addition, the approach taken by the Alphard language definition [Hilfinger 81] inspired the general translation process used in this dissertation.

Alphard was one of the first languages to include a powerful compile-time generic facility that provides the ability to specify complex run-time relationships among formal parameters. This power is due largely to Alphard's insistence on making no distinction between generic and non-generic parameters and the dynamic evaluation semantics among these parameters. Unfortunately, the language is not defined completely and the typing mechanism has deficiencies.

Although incomplete, the language definition took an interesting approach by defining the semantics of the language in terms of a small set of primitive actions (for example, copying a value and binding a variable to an object). The semantics of a program was determined by applying a set of rewrite rules to the program, so reducing it to an equivalent program expressed entirely in primitive actions.

Crucial to this definition technique was the availability of a new kind of subprogram called the *selector*. A selector is like a function but returns a modifiable object. (A function returns a constant object, which is not modifiable.) A selector that provides access to the elements of a symmetric matrix with integer elements could be defined as follows:

```
selector Subscript(A : Symmetric_Matrix;
                   I, J : Integer) return Integer is
        --
        -- Matrix is actually stored like a linear array,
        -- with elements in lexicographic order of
        -- indices (see Knuth, Vol. I [Knuth 68a], page 297).
        --
begin
    return A[I*(I-1)/2 + J];      -- returns the address of element
end Subscript;
```

An invocation of `Subscript` may be used as a variable of type `Integer`:

```
N := Subscript(Mat, 2, 5);
```

or, more interestingly:

```
Subscript(Mat, I, J) := Subscript(Mat, I, J) + 1;
```

The presence of selectors allowed a consistent approach to the rewrite rules. For example, subscripting was defined in terms of a rewrite rule that involved a selector. The construct

```
A[I1, I2, ..., In]
```

was merely a convenient abbreviation for

```
Subscript(A, I1, I2, ..., In)
```

This would invoke the predefined `Subscript` selector, or a user-defined selector named `Subscript` if one existed. Subprogram overload resolution determines the proper selector.

If selectors had not been available, subscripting and assignment would have required special treatment. The statement

```
Mat[I, J] := Mat[I, J] + 1;
```

might be represented semantically as follows:

```
Array_Element_Assign(Mat, I, J,
                     Array_Element_Value(Mat, I, J) + 1);
```

This is a more cumbersome and less consistent translation than that obtained using a selector.

The ideas embodied in the Alphard definition are similar to those used in the approach to translation advocated in this dissertation. A source language program is translated into an equivalent program expressed in a small set of primitives. Further processing (optimization and code generation) takes place on this intermediate program form. The advantage of this paradigm is that by concentrating on optimization methods for a small set of primitives a simple, but general, optimization technique is obtained. Section 2.2.2 contains a further discussion of this idea.

2.1.4. CLU

CLU [Liskov 77] superficially is similar to Alphard and Ada, but has chosen a LISP-like, object-oriented semantics as opposed to the Algol-like, value-oriented semantics of Alphard and Ada. In a language with a value-oriented semantics the creation and destruction of objects follows a stack discipline: the last object created is the first object destroyed. The implementation of such of a language can use a stack for object management. Storage management in an object-oriented system requires the use of a heap, because objects can be created and destroyed in an unpredictable order. This difference in storage manipulation has a significant effect on the implementation of generics, since, if code sharing is done, storage allocation and deallocation becomes a tricky matter (section 6.1).

Originally, CLU had a slightly different semantics for generic instantiation than Alphard and Ada. CLU allowed the definition of recursive generic units whose meaning could not be determined until run time (section 1.1.3). To accommodate this semantics the CLU compiler used a rather complicated run-time binding scheme, described in an early paper about the implementation of CLU [Atkinson 78].

The CLU implementors have since decided that this scheme was unnecessarily complex, and the current CLU compiler uses a form of polymorphic routine translation to implement generics.[11]

2.1.5. Miscellaneous Implementations

This section contains brief descriptions of additional research on the implementation of generic facilities. These efforts are collected here more for completeness than for their immediate relevance to this work.

2.1.5.1. Euclid

Euclid [Lampson 77] was designed as a programming language for writing verifiable systems programs. Where necessary, the designers sacrificed expressive power to ease of verification and generation of efficient code. In particular, the generic facility in Euclid is rather meager since generic parameters are restricted to objects; no generic type parameters are allowed.

Holt's paper [Holt 79] about the implementation of Euclid parameterized types discusses two implementation methods for such types: the *macro model*, and the *subroutine model*. The macro model is nothing more than generic expansion and the subroutine model is, essentially, polymorphic routine translation. The paper discusses the use of type templates to enable generic subprograms to share code.[12] Due to differences in the semantics of Euclid, the actual implementation technique is different from the routine sharing described in this dissertation.

---

[11] I was informed of the current implementation scheme in a private communication with R. W. Scheifler of M.I.T.

[12] Although Euclid has no generic subprograms, a subprogram may be imbedded within a parameterized type definition, yielding the equivalent of a generic subprogram.

2.1.5.2. Model

The language Model [Morris 79] is a Pascal derivative offering simple data abstraction and generic facilities. The generic facility has been designed so that it is easy to implement using generic expansion. The language, like Pascal, is static and all binding can be performed at compile time, making the task of generic expansion even easier.

2.1.5.3. The Work of Gehani

Gehani [Gehani 80] provides an implementation technique for and a proof about generic subprograms. The implementation technique is generic expansion. Gehani also suggests that while performing this expansion, the compiler should note identical generic instantiations and use a single instantiation. The proof involves a subtle question about performing semantic analysis of generic subprograms: Is it possible to determine at compile time whether a recursive generic instantiation will lead to an infinite recursion? As might be expected, the problem is undecidable, even when the language is restricted severely.

The proof is irrelevant because the source language in this dissertation outlaws recursive generic instantiations. In a language like EL1, however, determining the instantiations that could be processed at compile time might allow useful optimizations.

2.1.5.4. C

A data abstraction facility of sorts has been added to the language C [Stroustrup 81]. The abstraction facility is straightforward, but the designers suggest a technique by which generic abstract data types may be obtained. The method requires that the programmer use the macro definition facility of the compiler to perform macro expansion on a type definition module. This is an extreme example of generic expansion, and the use of this technique can lead to the semantic discrepancies described in section 1.1.2.

2.2. Space-Compacting Compilers

The work in code space compacting is dominated by the PQCC and ECS compiler projects. These ambitious efforts were aimed at producing machine-independent production-quality code generators. Although their goals were similar, the philosophies of these projects were different.

2.2.1. PQCC

The PQCC project at Carnegie-Mellon University [Leverett 80, Wulf 80] is a continuation of ideas formulated in the development of the BLISS-11 compiler. The BLISS-11 compiler, which is described in the book by Wulf et al. [Wulf 75], was highly machine-dependent and produced extremely high-quality code. Experiments showed that the code produced by BLISS-11 was only about 7% larger than assembly language code written by an experienced programmer. In addition, the BLISS-11 code was found to execute as fast as the hand-written code.[13]

The PQCC project built on the experience gained in the development of the BLISS-11 compiler and pursued the goal of building a truly automatic compiler-writing system. Such a compiler-writing system would take a language description and a machine description and produce a *production quality compiler* (a *PQC*) for the given language and target machine.

In order to produce a PQC that generates high-quality code, the designers of PQCC thought it desirable to follow the general structure of the BLISS-11 compiler. Thus, a PQC is composed of many phases,[14] each of which does a small, well-defined job. This structure has the advantage of modularity and allows code generation decisions to be delayed. Delaying a decision means to put off making the decision until as late in compilation as possible. Ideally, each decision would be made after all decisions that could affect it.

The delaying is crucial since a PQC contains many ad hoc, machine-dependent optimization techniques that rely on heuristics to combat inherent combinatorial explosion. The delaying makes more knowledge available for a decision, and the decision is, thus, more likely to be correct.

2.2.2. ECS

The ECS project at IBM [Allen 77] investigated language and machine independence in optimizing compilers. Unlike the PQCC effort, the ECS researchers were not concerned with producing a generator for such compilers. Nonetheless, their goal of machine-independence and high-quality code is similar to the goals of PQCC.

The ECS philosophy is to translate the source program to a machine-independent intermediate

---

[13] These experiments were performed at the Digital Equipment Corporation. The results stated in this dissertation were provided to me in a private communication with Wm. Wulf.

[14] Currently, a PQC consists of more than 40 phases.

language. Standard, machine-independent optimizations are then applied to this representation. Optimization is followed by a phase that transforms this intermediate code into a lower, machine-dependent level, from which machine language is generated.

In many ways, the approach in this dissertation is closer to the ECS philosophy than to the philosophy of PQCC. The translation methodology used in ECS and this work is as follows:

- machine-independent translation of the source program to an intermediate language;
- machine-independent optimization of the intermediate program;
- machine-dependent translation of the intermediate code into machine language.

With this technique the application of general optimization techniques is intended to subsume many ad hoc optimization techniques and thus avoid special case analysis. The most important machine-independent optimization technique suggested by ECS and this dissertation is inline substitution. Inline substitution is important because of the nature of the machine-independent translation process. Machine independence is obtained by treating source language operations like subprogram invocations. This allows a consistent methodology regardless of the target machine instruction set (explained further in chapter 7).

For example, consider the source language expression

```
I * J
```

On machines without a hardware multiply instruction, it may be desirable to translate this expression into an invocation of a software multiplication routine. But inline code is preferable for target machines with a multiply instruction. This can be obtained if the expression is translated into

```
Integer_Multiply(I, J)
```

rather than into any particular sequence of machine instructions.[15] The application of inline substitution will result in inline machine code or an invocation, whichever results in less code.

[15] Of course, `Integer_Multiply` may be represented as a sequence of machine instructions.

2.3. Inline Substitution

Inline substitution is an optimizing technique that has been known for a long time[16] [Allen 77, CLI 75], but little work has been done on determining the desirability of the transformation. The lack of knowledge about the desirability of inline substitution is evidenced by most compilers that employ the method because they require the programmer to indicate those subprograms that are to be substituted inline at each invocation [ANSI 66, DOD 82, Mitchell 79].

Section 2.3.1 discusses an interesting investigation into the desirability of inline substitution to reduce expected execution time. Recently Ball has investigated techniques for predicting the desirability of inline substitution; this work is discussed in section 2.3.2.

2.3.1. The Work of Scheifler

Scheifler was interested in the use of inline substitution to decrease program execution time [Scheifler 77]. Although this dissertation is concerned with decreasing space, the work of Scheifler is relevant since it is one of the few investigations into automating inline substitution.

Scheifler set out to investigate the following problem: Given a CLU program, a maximum program size and a maximum subprogram size, determine a sequence of inline substitutions (over a specified subset of all invocations) that minimizes the expected program execution time. His paper contains a proof that this problem is NP-hard and, therefore, believed to be intractable.

Scheifler, therefore, turned to the use of heuristics. The algorithm he used consisted of three steps:

1. All invocations that may be expanded inline without increasing the total program size are replaced. This is worthwhile, since removing subprogram call overhead cannot increase expected execution time.
2. The invocation with the highest ratio of expected number of executions to estimated size change is expanded inline.[17] This step is repeated until the maximum program size is reached.
3. All subprograms that are invoked once only are expanded inline if the substitution does not violate the subprogram size constraint.

---

[16] References to the method may be found under the terms *inline expansion, procedure integration, statement functions, inline procedures* and *open procedures*.

[17] As a preliminary to this algorithm, each program was executed with various sets of input data and statistics on subprogram invocations collected.

In estimating the size change resulting from a substitution in step 2, Scheifler used a simple approximation of the final code size of an expression. The size of an expression with N operands is (N + 1) plus the size of each operand that is an expression. Certain operators, such as **if-then-else**, were given special treatment to achieve a better estimate.

Scheifler points out that this approximation measure is not very accurate, although it does seem to provide a good indication of relative size. If one is very concerned with the increase in size caused by the application of inline substitution, it is likely that a more accurate size measure would be necessary.

2.3.2. The Work of Ball

In his Ph.D. dissertation [Ball 82], Ball investigated the automation of two related optimizations: inline substitution and *subprogram specialization*. Of these two techniques, only subprogram specialization is unfamiliar to the reader. Subprogram specialization is the creation of a customized version of a subprogram. For example, consider the procedure `FWrite`:

```
procedure FWrite(S : String; Block : Boolean) is
begin
   Write(Out_File, S);
   if Block then          -- Do line blocking
      for I in 1..80-S'LENGTH loop
         Write(Out_File, " ");
      end loop;
   end if;
end FWrite;
```

When invoked, `FWrite` writes the string `S` to the file `Out_File`. If the boolean parameter `Block` is true, then `FWrite` performs line blocking.

If a program has calls of the form

```
FWrite(Mesg, False);       -- Mesg is a variable
```

a specialized version of `FWrite`, say `FWrite2`, could be constructed:

```
procedure FWrite2(S : String) is
begin
   Write(Out_File, S);
end FWrite2;
```

Calls of the form

```
FWrite(Mesg, False);
```

are transformed to

```
FWrite2(Mesg);
```

These invocations execute faster than the untransformed versions—there are one fewer parameters to pass, and there is no test of `Block` (in the `if` statement) to perform.

2.3.2.1. The Time-Space Tradeoff

The application of either inline substitution or specialization always decreases execution time.[18] But this decrease in time is frequently at the expense of an increase in space. Ball dealt with this tradeoff by allowing the user to specify a *time/space ratio:* the ratio of the expected savings in time to the increase in code space. This ratio is used as a threshold for accepting transformations so that a transformation is rejected if its time/space ratio is smaller than the specified ratio.

2.3.2.2. Transformation Selection

Ball's algorithm for selecting the transformations to perform is as follows:

1. enumerate a list of possible transformations;
2. calculate the time/space ratio of each transformation;
3. perform the transformation with the highest time/space ratio;
4. re-evaluate the remaining transformations (step 3 has modified the program);
5. repeat from step 3 until the possible transformations are exhausted.

As for many optimization problems, parts of the process are combinatorially intractable. A particularly sticky problem in this regard is the enumeration of possible candidates for specialization. The set of possible specializations for a procedure may not be independent. For example, the choice of a particular specialization of a subprogram may affect the desirability of another specialization of that subprogram. To determine the optimum transformations, all possible combinations of specializations for a procedure must be considered. To avoid this combinatorial explosion, Ball developed a heuristic (which will not be discussed) that limits the number of specialization candidates.

A crucial part of this process is the estimation of the size of the code that will result from a transformation. Actual compilation of all possible transformations is impractical, so a method of estimating the result of a transformation is needed.

---

[18] Execution time will not be improved if the transformed invocation is never executed, but this is a pathological case.

Ball uses a form of flow analysis to predict the size of a transformation. For each expression in a subprogram the expression's dependency on the subprogram's parameters is determined. The dependency information is used to predict the effect that constant actual parameters will have on the compile-time determination of the expression.

For example, in procedure `FWrite`:

```
procedure FWrite(S : String; Block : Boolean) is
begin
   Write(Out_File, S);
   if Block then          -- Do line blocking
      for I in 1..80-S'LENGTH loop
         Write(Out_File, " ");
      end loop;
   end if;
end FWrite;
```

the analysis would find that if the actual parameter passed to `Block` is constant, then the test of the if statement will be determinable at compile time in the inline expansion.

This information is used to estimate the effect of inline expanding an invocation of `FWrite`. In particular, if the parameter to `Block` is constant, then Ball predicts that the size of the inline expansion by

the size of the invocation `Write(Out_File, S)` +
(*the size of the body of the* `if`) * (*probability that body is executed*)

This method depends on estimates of execution frequency for the program. These estimates are also needed to compute the time/space ratios for a transformation. Ball assumes that execution frequency statistics are available for a program.

Chapter 3
The Source Language

To test the feasibility of the compilation process it was necessary to choose a source language for test programs. The particular choice of language is unimportant, but the language must have certain characteristics:

- The language is intended to be compiled.
- The language permits compile-time type checking.
- The language has a compile-time generic facility.
- The language has a value semantics, not an object semantics.

A language with these characteristics can provide a powerful generic facility and still be compiled into efficient code.

This chapter provides the reader with enough understanding of the source language to grasp the bulk of this work. The derivation of the language from Ada is discussed in the section 3.1. The description of the source language is divided into two parts: the general semantics of the language (section 3.2) and the generic facility (section 3.3). The reader familiar with Ada and its generic facility may wish to read only sections 3.2.2 (subprograms) and 3.2.2.3 (user-definable operations). Readers familiar with Ada, but not with its generic facility, should read section 3.3.

3.1. The Base Language

The language Ada [DOD 82] is the basis for the source language since Ada has the necessary characteristics and also the following benefits over other languages:[19]

- much effort has gone into the design of Ada and its generic facility;
- Ada is the most widely known language with a generic facility;
- the definition of the Ada language is accessible [DOD 82];

---

[19] For information on the Ada language the reader is referred to the Language Reference Manual [DOD 82] or one of the many Ada textbooks [Habermann 83, Hibbard 83].

- Ada will be implemented in its entirety, so work on the generic facility of Ada is sure to be beneficial to compiler writers.

The designers of Alphard felt that selectors (a kind of function, described in section 3.2.2.1) and a full complement of user-definable operators were crucial to good data abstraction [Shaw 81]. I agreed wholeheartedly and so included selectors and user-definable operators to determine their effect on the implementation of generic subprograms. In addition, these features enabled a consistent approach to translating *actions*. Actions, which include the predefined operators and operations like subscripting, are explained in section 5.

I also took liberties in eliminating parts of the Ada language not relevant to the generic facility. Thus, the source language includes a **while** loop, but not a **for** loop. Implementing a **for** loop adds complexity to the compiler without affording insight into the implementation of generic subprograms.

3.2. General Semantics

This section describes the general (non-generic) semantics of Ada. Important features of Ada are its scoping mechanisms, which are described in sections 3.2.1 (packages) and 3.2.2 (subprograms). The type system is discussed in section 3.2.3, attributes in section 3.2.4 and separate compilation in section 3.2.5.

3.2.1. Packages

The unit of data encapsulation in the source language is the *package*, which provides a scope for a set of related entities. In addition, a package forces the separation of specification and implementation because the definition of a package consists of a *package specification* and a *package body*.

The package specification defines the entities that are visible outside the package definition. For example, the package `Screen` following is a collection of types, variables and subprograms useful for video screen manipulation.

```
package Screen is
   type Coordinate is ...;
   X, Y : Coordinate;                -- Type of screen coordinates
   procedure Move(NewX, NewY : in Coordinate);
                                     -- Current cursor position
end Screen;                          -- Move cursor
```

The procedure declared in the specification of package `Screen` is defined without a body; the body must appear in the corresponding package body (section 3.2.2 describes subprogram specifications and bodies). A package body must provide bodies for subprograms declared in the corresponding package specification. For example, the body for package `Screen` might look as follows:

```
package body Screen is
   procedure Move(NewX, NewY : in Coordinate) is
   begin
      ...
   end Move;
end Screen;
```

3.2.2. Subprograms

There are three varieties of source language subprograms: *procedures*, *functions* and *selectors*. A procedure provides an abstraction that encapsulates an action, while a function encapsulates a computation that yields a value. A selector is similar to a function, but yields an object instead of a value.

A subprogram may be defined in two parts: a *subprogram specification* and a *subprogram body*. The subprogram specification provides the kind of subprogram (procedure, function or selector), the subprogram name and the formal parameters. In addition, a function or selector specification provides a *return type*, which is the type of the value (or object, for a selector) that must be returned via a **return** statement.

A subprogram body consists of a subprogram specification followed by a sequence of statements. Every subprogram must have a body, but, in general, a separate subprogram specification may be omitted because it will be obtained from the subprogram body. A separate subprogram specification is necessary only if either the subprogram is declared in a package specification or the subprogram is generic (section 3.3.1). If a separate subprogram specification is given, it must be identical to the specification in the corresponding subprogram body.

The three kinds of source language subprograms have identical syntax and semantics for their *subprogram parameters*, which are illustrated in the subprogram specification following:

```
procedure Search(Key, Alternate : in String;
                Found : out Boolean;
                Count : in out Integer);
```

In this example, `Key` and `Alternate` are declared as formal parameters of type `String`, `Found` of type `Boolean` and `Count` of type `Integer`.

There are three *binding modes* that may be used to specify the binding of an actual parameter: **in**, **out** and **in out**. The meaning of the binding modes is as follows:[20]

- **in**: At subprogram entry one of two actions occurs (at the option of the compiler): the formal parameter is initialized with the value of the corresponding actual parameter or the formal parameter becomes an alias for the actual parameter. The formal parameter may not be modified—it is read-only.
- **out**: The formal parameter becomes an alias for the actual parameter, so a modification to a formal parameter immediately affects the actual parameter. The formal parameter may not be read—it is write-only.
- **in out**: The formal parameter becomes an alias for the actual parameter, so a modification to a formal parameter immediately affects the actual parameter. The formal parameter may be read or written.

3.2.2.1. Selectors

A *selector* is identical to a function except that a selector returns a modifiable object instead of a value. This means that a selector invocation may be used like a variable; a function invocation may be used only like a value. Section 2.1.3 gives an example of the definition of a selector that performs subscripting on a symmetric matrix.

Within the body of a selector a **return** statement returns an object. For example, the statement

```
return A;
```

causes execution to return from the enclosing selector and yield the object bound to **A**. If this statement were in a function, then it would cause a return with the value contained in **A**.

3.2.2.2. Overloading

A program may contain distinct operations that perform the same logical action on objects of different types. An example of such an operation is the determination of the length of a value of type `String`, `List` or `Set`. Using a single name for these related actions aids abstraction:

```
function Length(S : in String) return Integer;
function Length(L : in List) return Integer;
function Length(S : in Set) return Integer;
```

For this reason the source language allows multiple subprograms with the same name, known as

---

[20] The definition of parameter passing is a touchy matter. The binding modes in the source language were defined with two goals in mind: to provide a deterministic semantics in the presence of aliasing of actual parameters and to allow small values of binding mode **in** to be passed efficiently. The first goal is satisfied for actual parameters of binding mode **out** or **in out**. For parameters of binding mode **in**, the first goal was sacrificed in favor of efficiency.

overloading. In case the subprogram name in an invocation is not unique, the compiler is responsible for determining the target subprogram. To enable the compiler to do this, the types of the arguments in the invocation must be sufficient to determine a single subprogram, or the invocation is illegal.

3.2.2.3. User-Definable Operators

Most programming languages provide infix notation for frequently-used operators. This is convenient and corresponds to mathematical usage because the arithmetic operators (+, -, /, *) and the relational operators (<, <=, >, >=, =) are commonly written in this manner. The source language allows the programmer to define operators and obtain the same convenience of notation.

User-defined operators are written as function definitions distinguished by the appearance of a string for the function name, which must be one of the following strings: `":="`, `"="`, `"/="`, `"<"`, `"<="`, `">"`, `">="`, `"+"`, `"-"`, `"*"`, `"and"` and `"or"`. Operators are invoked in infix notation

```
4 + A
```

or standard subprogram invocation notation

```
"+"(4, A)
```

The assignment, equality and inequality operations are defined automatically for all types in the language.[21] A user definition of assignment, equality or inequality may interact with these predefined operators. The semantics of these interactions need to be stated explicitly, as there several possible interpretations.

To see this, let assignment be redefined for a type T. If an array type is declared with components of type T, the semantics of assignment for objects of this array type is indeterminate. There are two obvious possibilities: Each element of the array is assigned using the redefined assignment operator, or assignment for the array type is as predefined, so the effect is that of copying array components. Either definition is acceptable, but the source language adopts the first.[22] There are identical considerations for equality and inequality, and these are handled in a similar manner: redefined operators are used when they exist.

---

[21] This is not true for limited private generic type parameters, which are described in section 3.3.3.1.

[22] The definition of assignment interacts with the semantics of parameter passing and function return since these actions involve the transmission of values. This creates sticky semantic problems that are beyond the scope of this dissertation.

3.2.3. Types

A *type* defines a set of values, which are said to be *of the type*. A type is also attached to every object, and the object is also said to be *of the type*. The compiler uses this type information to insure that an object of type T will only contain values of type T (or no value). The two classes of source language types are *discrete types* and *array types*. Discrete types are described in the next section and array types in section 3.2.3.2.

3.2.3.1. Discrete Types

The primary characteristic of the class of discrete types, which comprises the *integer* and *enumeration* types, is that a value of a discrete type has no substructure. The values of a particular discrete type are, however, ordered and this allows the programmer to iterate through the values of the type (section 3.2.4).

There is a single integer type, `Integer`, whose values form an implementation-defined subset of the integers. The usual arithmetic and relational operators are defined on values of type `Integer`.

An *enumeration* type is defined by a list of identifiers, which defines the set of values for the type. The predefined type `Boolean` is an example of an enumeration type. Its definition is

```
type Boolean is (False, True);
```

The two values of type `Boolean` are denoted `False` and `True`.

3.2.3.2. Array Types

A value of an *array type* is an ordered sequence of objects of a specified type, known as the *component type*. The sequence of objects is indexed by values of a specified discrete type, the *index type*. For example, the array type `String` is predefined as[23]

```
type String is array(Integer range <>) of Character;
```

A value of type `String` is a sequence of `Character` objects indexed by `Integer` values.

An array type definition provides no actual bounds for array values. The definition only specifies an index type for the bounds—in this case, `Integer`. An object of an array type, however, must have specific bounds and these are provided by appending an *index constraint* to the type name. For example, in the object declaration

---

[23] `String` is predefined in the package STANDARD (section 3.2.5).

```
Name : String(0..9);
```

the index constraint `(0..9)` specifies a lower bound of 0 and an upper bound of 9. This constraint restricts `Name`'s values to values of type `String` with lower bound 0 and upper bound 9.

3.2.4. Attributes: FIRST, LAST, SUCC and PRED

Ada contains a class of expressions known as *attributes*, which are distinguished by a peculiar syntactic form. The source language contains several expressions derived from the Ada attributes, although much of the distinctive syntax has been eliminated.

The `FIRST` and `LAST` attributes yield information about the bounds of an array object. The expression

```
A'FIRST
```

yields the value of the lower bound of array `A`. To obtain the value of the upper bound of array `A` one uses the expression

```
A'LAST
```

The type of a `FIRST` or `LAST` attribute is the index type of the specified array.

The `PRED` and `SUCC` attributes are applicable only to expressions of a discrete type. `PRED` applied to a discrete expression yields the predecessor of the expression; `SUCC` yields the successor of an expression.[24] The predecessor and successor of an integer value are determined in the obvious manner:

```
PRED(N) ≜ N - 1
SUCC(N) ≜ N + 1
```

For values of an enumeration type the ordering is that specified in the type definition. For example, for type `Color`

```
type Color is (Red, White, Blue);
```

the ordering is

```
Red < White < Blue
```

The predecessor of an enumeration value is its predecessor in the ordering; the successor of a value is its successor in the ordering.

---

[24] It is illegal if `PRED` is applied to a value with no predecessor or if `SUCC` is applied to a value with no successor.

3.2.5. Separate Compilation

The unit of compilation in the source language is a subprogram specification or body, or a package specification or body. Additionally, the subprogram may be generic, and any number of subprograms and packages may be collected into a single unit for compilation.

A unit is compiled in a standard predefined environment. This environment is defined by the package STANDARD, which contains the definitions of all predefined language entities; the type `Integer` and the arithmetic operators, for example. To make the predefined entities visible the compilation of a unit takes place as if the unit were declared at the end of the body of package STANDARD. For example, the generic procedure `Exchange` is compiled as if the following were done:

```
package STANDARD is
   . . .          -- Specifications of predefined entities
                  -- (appendix B)
end STANDARD;

package body STANDARD is
                  -- Implementations of predefined entities
   . . .
   generic
      type T is private;
   procedure Exchange(X, Y : in out T);

   procedure Exchange(X, Y : in out T) is ...;
end STANDARD;
```

It is not intended that package STANDARD be definable in the source language. The package is a definitional aid and the compiler is free to implement package STANDARD in any semantically consistent manner. The specification of package STANDARD used in the experimental compiler is given in appendix B.

The result of the compilation of a unit is stored so that the unit may be used in a later compilation. A compiled unit is made usable by a *with clause*. The **with clause**

```
with A, B;
```

must appear directly before a unit to be compiled; it extracts the results of the compilation of `A` and `B` from the library. The unit following the **with** clause is compiled as if `A` and `B` were declared directly before it in package STANDARD.

It is not necessary for the body of a subprogram or package to include a **with** clause for the corresponding specification. The appropriate specification is obtained automatically by the compiler.

3.3. The Generic Facility

The remainder of this chapter describes the generic facility of the source language. Section 3.3.1 discusses the form of a generic subprogram, section 3.3.2 describes generic instantiation and section 3.3.3 describes generic parameters.

3.3.1. Generic Subprograms

A generic subprogram consists of a subprogram specification, which is preceded by a *generic header*, and a subprogram body. Although the subprogram body is written without a header, the body may refer to the entities declared in the generic header. This is illustrated in the following example:

```
generic                              ------------------------------
   type T is private;                -- The generic header.
                                     ------------------------------

procedure Exchange(X, Y : in out T);
                                     ------------------------------------
                                     -- The procedure specification.
                                     -- It may refer to T.
                                     ------------------------------------

-- Note no generic header
procedure Exchange(X, Y : in out T) is
                                     ------------------------------------
   Temp : T;                         -- This is the procedure body.
begin                                -- It also may refer to T.
   Temp := X;                        ------------------------------------
   X := Y;
   Y := Temp;
end Exchange;
```

3.3.2. Generic Instantiation

A generic subprogram is a template and may be used only as the target of a *generic instantiation*, which defines a specialized copy of the generic unit.[25] A generic instantiation must provide an actual parameter for each generic formal parameter specified in the generic header. For example, a version of procedure `Exchange` appropriate for variables of type `Character` is defined as follows:

```
procedure Char_Swap is new Exchange(Character);
```

In this case, the formal parameter `T` is bound to `Character`.

---

[25] The requirement that a generic instantiation be used to obtain a subprogram that can be invoked is known as *explicit instantiation*. It is possible to allow a generic subprogram to be invoked as if it were a subprogram that accepts parameters of different types. In this formulation, called *implicit instantiation*, it is the compiler's responsibility to determine the appropriate instantiations of the generic subprogram. There are good arguments for both kinds of instantiation, but the source language retains the explicit instantiation used in its base language, Ada.

Each actual parameter must *match* the corresponding formal parameter. The matching rules are provided in the following sections, which describe the three kinds of generic parameters.

3.3.3. Generic Parameters

There are three kinds of generic parameters: *generic type parameters*, *generic subprogram parameters* and *generic object parameters*. A generic parameter may be used within a generic unit in any manner consistent with its kind. Therefore, a generic type parameter may be used like a type, a generic object parameter may be used like an object and a generic subprogram parameter may be used like a subprogram.

3.3.3.1. Generic Type Parameters

Within a generic subprogram the actual type of a generic type parameter is unknown. For the compiler to determine what operations are available for the generic type, the programmer must state his demands on the actual type. For example, he may require that the actual type be an array type with discrete elements, or that assignment, equality and inequality are defined for objects of the type. The use of a generic type parameter is restricted to a set of operations determined by the programmer's stated requirements.

The four kinds of generic type parameters are *limited private*, *private*, *discrete* and *array*. Each kind of parameter makes a different set of demands of the actual type. These sets define the operations that may be performed with the type parameter within the generic subprogram.

The minimum demand is made by a *limited private* generic type parameter, which is declared as

```
generic
   type T is limited private;
```

A limited private type parameter requires nothing of its actual type, so any type matches this kind of parameter. Consequently, nothing is known about the generic type parameter within the generic unit. The legal operations for a limited private type parameter are

- As the type in an object declaration:

  ```
  Temp : T;
  ```

- As the component type in an array type definition. For example,

  ```
  type New is array(Integer range <>) of T;
  ```

  (The type may not be used as the index type in an array type definition (section 3.2.3.2) because an array must be indexed by a discrete type and a limited private type is not known to be discrete.)

- As an actual generic type parameter in a generic instantiation. For example,

```
procedure Write_Elt is new Write(T);
```

A *private* generic type parameter demands that its actual type have assignment, equality and inequality operations. Of course, a private type parameter also has the operations of a limited private type parameter.

For example, within the body of procedure `Exchange`, whose specification is

```
generic
   type T is private;
procedure Exchange(X, Y : in out T);
```

values of type `T` may be assigned and compared for equality and inequality.

A *discrete* generic type parameter is declared as

```
generic
   type Index is (<>);
```

A discrete generic type parameter matches an actual parameter that is a discrete type: `Integer` or an enumeration type. As the type is known to be discrete, the following operations are available (in addition to those for a private type):

- The SUCC and PRED attributes (section 3.2.4) are applicable to values of the type.
- The relational operators "<", "<=", ">" and ">=" are applicable to values of the type.
- The type may be used as the index type in an array type definition, for example,

```
type IC is array(Index range <>) of Character;
```

An *array* generic type parameter is illustrated in the following program fragment:

```
generic
   type Index is (<>);
   type Element is private;
   type IE is array(Index range <>) of Element;
function Binary_Search(A : in IE;
                       Key : in Element) return Boolean;
```

Note that the definition of `IE`, which declares an array generic type parameter, is allowed to reference preceding generic type parameters.

An array generic type parameter matches an array type that has the same index and element types as the generic parameter. For example:

```
function Char_Search is
      new Binary_Search(Integer, Character, String);

                -- This is legal: Index -> Integer
                --                Element -> Character
                --                IE -> String[26]
                --
```

For an array generic type parameter:

- values of the type may be subscripted;
- the FIRST and LAST attributes (section 3.2.4) may be applied to values of the type.

For convenience, the four kinds of generic type parameters are summarized in table 3-1.

Kind	Representation	Operations
Limited private	**limited private**	*object declaration, generic actual parameter,*
Private	**private**	*object declaration, generic actual parameter,* ":=", "=", "/="
Discrete	(◇)	*object declaration, generic actual parameter,* ":=", "=", "/=", "<", "<=",">", ">=", SUCC, PRED, *array index type*
Array	**array**(*index type* ◇) **of** *component type*	*object declaration, generic actual parameter,* ":=", "=", "/=", FIRST, LAST, *subscripting*

Table 3-1: Generic Type Parameters

3.3.3.2. Generic Subprogram Parameters

A generic parameter may be a subprogram, like "<" in the following specification:

```
generic
   type T is limited private;
   with function "<"(X, Y : in T) return Boolean;
                  -- Generic subprogram parameter,
                  -- preceded by with

   function Min(X, Y : in T) return T;
```

A generic subprogram parameter, F, is matched by an actual subprogram parameter, A, if and only if

- A is the same kind of subprogram (procedure, function or selector) as F, and
- A has the same number of parameters as F, and corresponding parameters have identical types (the parameter names are irrelevant), and

[26] String is predefined in package STANDARD as

```
array(Integer range <>) of Character;
```

- if F is a function or selector, then A and F have identical return types.

Within the generic subprogram, a generic subprogram parameter may be used as a subprogram with the given specification and may be invoked, or it may be used as a generic actual parameter.

3.3.3.3. Generic Object Parameters

It is also possible to parameterize a generic subprogram with a generic object parameter:

```
generic
   type T is private;
   Size : in Integer;          -- Generic object parameter

subprogram Stack() is
   . . .
```

A generic object formal parameter matches an actual parameter that is a value (or object) of the same type as the formal.

The semantics of binding for generic object parameters is different from that for subprogram parameters. The effect of generic instantiation is as if the generic actual parameters were substituted for the generic formal parameters throughout the generic subprogram. The binding of an actual object parameter to a generic object parameter is enduring and is in effect for the lifetime of the generic subprogram. This is in contrast to the binding of an object parameter to a subprogram, which exists only during an invocation of the subprogram.

Part Two

Translating Generic Subprograms

Chapter 4
Generic Expansion: Description and Implementation

Generic expansion (GE) is the technique suggested most often for translating generic subprograms [Gehani 80, Morris 79]. GE translates a generic instantiation by creating a specialized copy of the generic subprogram. The copy is specialized by replacing references to generic formal parameters by the corresponding actual parameters.[27] As a first approximation, the specialization can be done by copying the generic subprogram and substituting a reference to a generic actual parameter for each reference to a generic formal parameter.

This simple procedure is not adequate in general because of the possibility of name conflicts and side effects. A name conflict occurs when a name that appears in a generic actual parameter is re-bound locally in the subprogram copy. This is incorrect because the name must be bound in the context of the generic instantiation. Generic actual parameters may contain expressions that have side effects, and these expressions must be evaluated once, at generic instantiation time. But simple replacement may produce multiple copies of such expressions, which will then be evaluated (incorrectly) more than once.

Because the experimental compiler performs GE on the intermediate program form, side effects are the only problems that need to be handled. Section 4.1 discusses one solution to this problem.

---

[27] This technique is similar to the subprogram specialization transformation suggested by Ball [Ball 82] and described in section 2.3.2.

To see how GE works, consider the generic procedure `Print`:

```
generic
   type Index is (<>);
   type Elt is limited private;
   type Arr is array(Index range <>) of Elt;
   with procedure Write(E : in Elt);
procedure Print(A : in Arr);

procedure Print(A : in Arr) is
   I : Index;
begin
   I := A'FIRST;
   while I < A'LAST loop
      Write(A(I));
      Put(", ");
      I := SUCC(I);
   end loop;
   Write(A(A'LAST));
end Print;
```

and the instantiation

```
procedure Sprint is
   new Print(Integer, Character, String, Char_Put);
```

Performing GE by substituting `Integer` for `Index`, `Character` for `Elt`, `String` for `Arr` and `Char_Put` for `Write`, yields

```
procedure Sprint(A : in String) is
   I : Integer;
begin
   I := A'FIRST;
   while I < A'LAST loop
      Char_Put(A(I));
      Put(", ");
      I := SUCC(I);
   end loop;
   Char_Put(A(A'LAST));
end Sprint;
```

This specialized copy is then processed by the compiler like a non-generic procedure.

4.1. Implementing GE

GE can be performed at the source level or within the compiler on the intermediate form of the program. This section discusses manipulations performed on the intermediate program representation. (The extrapolations necessary to implement GE as a source-to-source transformation technique are simple.) For the most part, the GE translation of an instantiation can be implemented as described in the previous section: copy the generic subprogram and replace all references to a generic formal parameter by its corresponding actual parameter.

Unfortunately, naive copying and replacing will sometimes fail, and more care must be taken in performing GE. As mentioned in the introduction to this chapter, the reason for this failure is side effects in generic actual parameters. Only generic actual type parameters and generic actual object parameters can contain expressions, so only these parameters can have side effects. There is no problem with subprogram parameters, as a generic actual subprogram parameter may only be the name of a subprogram, which has no side effects.

A generic actual type parameter that is an array type may contain expressions with side effects. Consider the following declarations:

```
generic
   type T is private;
procedure Buffer();

procedure Buffer() is
   Line1 : T;
   Line2 : T;
begin
   . . .
end Buffer;

function Bound() return Integer is
begin
   Next_Bound := Next_Bound + 1;
   return Next_Bound;
end Bound;

procedure New_P is new Buffer(String(Bound()..Bound()));
```

During evaluation, the type parameter `String(Bound()..Bound())` has a side effect because `Next_Bound` is modified. The semantics of generic instantiation calls for binding the generic formal parameter `T` at the point of instantiation. This would cause the evaluations of two invocations of `Bound`. But naive replacement yields

```
procedure New_P() is
   Line1 : String(Bound()..Bound());
   Line2 : String(Bound()..Bound());
begin
   . . .
end New_P;
```

which contains four invocations of `Bound`, and is incorrect.

One solution is to evaluate the expressions in `String(Bound()..Bound())` prior to the expansion of the subprogram. These results are stored in compiler-generated temporary variables and used in place of the invocations of `Bound`:

```
LB : Integer := Bound();
UB : Integer := Bound();
procedure New_P is new Buffer(String(LB..UB));
```

Replacement now yields

```
procedure New_P() is
   Line1 : String(LB..UB);
   Line2 : String(LB..UB);
   . . .
```

As with generic actual type parameters, a generic actual object parameter may involve side effects. For example, in

```
generic
   NewLine : in Character;
function Read() return String;

function Read() return String is . . .;

Break_Chars : String(0..127);

function Next() return Integer is
begin
   Cur_Char := Cur_Char + 1;
   return Cur_Char;
end Next;

function Command_Read is new Read(Break_Chars(Next()));
```

the generic actual object parameter `Break_Chars(Next())` involves a side effect. If the generic formal parameter `NewLine` is referenced more than once within the generic function `Read`, then simple substitution will lead to multiple invocations of `Next`. This problem may be solved like the problem for type parameters: The actual parameter `Break_Chars(Next())` is evaluated prior to the expansion of `Read` and the result stored in a compiler-generated variable. The value of this variable is then used in place of the generic actual parameter. Of course, the compiler may want to perform this preevaluation only in those cases where it cannot determine that an actual parameter is free of side effects.

4.2. The Advantages of GE

One nice feature of GE is that it can be utilized in source-to-source program transformations. Of course, this may not produce acceptable results for large programs, but it does allow a compiler that does not implement generic subprograms to be used in translating programs containing generic subprograms. This is done by feeding the source program through a preprocessor that produces an

equivalent source program containing no generic subprograms. This dissertation does not discuss the details necessary to transform a program with generic subprograms. The transformations are simple, but care is required to handle name conflicts and generic actual parameters with side effects.

Another nice characteristic of GE is that it produces code that is especially amenable to further optimizations, such as constant folding, constant propagation and variable lifetime analysis. This is because the copying performed by GE produces subprograms with a lot of specific information. The optimizations customize each subprogram copy to its own needs, so that GE also produces a translation of a generic instantiation that executes as fast as possible (in the absence of anomalous paging behavior).

In addition, GE is a fast translation technique for programs that make light use of generic subprograms. The quantification of this is discussed in chapter 9, where the speed of GE translation is compared to the speed of the alternative translation technique, polymorphic routine translation.

4.3. The Disadvantages of GE

The biggest problem with GE is that it may generate an unnecessarily large translation for a program with a heavy use of generic subprograms. This is discussed in chapter 9, which shows that the alternative translation technique produces significantly smaller translations than GE does for some programs. For these programs, the GE translation also requires more time than the PR translation.

Semantic analysis of a generic instantiation requires only the specification of a separately-compiled generic subprogram.[28] To perform GE it is necessary to have the body of the target generic subprogram available to copy. This can be a disadvantage in a compiler since it must arrange that the bodies of separately compiled generic subprograms be accessible and in a convenient form for use by GE.

---

[28] This was one of the original design goals of Ada. (This point, and many other design issues, can be found in the *Ada Rationale* [Ichbiah 79].) Unfortunately, the Ada language now requires that some semantic analysis be performed on the body of a generic subprogram for each generic instantiation.

Chapter 5
Polymorphic Routine Translation: Description

The GE technique is extreme: it creates a copy of a generic subprogram for each instantiation. *Polymorphic routine translation* (*PR*) is extreme also, but in the opposite sense. PR shares a single routine among all generic instantiations of a generic subprogram.

To see how PR works, consider the generic function `Concat` illustrated in figure 5-1.

```
generic
   type Elt is private;
   type Arr is array(Integer range <>) of Elt;
function Concat(A, B : in Arr) return Arr;

function Concat(A, B : in Arr) return Arr is
   I, J : Integer;
   Result : Arr(1..A'LAST+B'LAST);     -- Assume A'FIRST=B'FIRST=1
begin
   I := 1;
   while I <= A'LAST loop
      Result(I) := A(I);
      I := I + 1;
   end loop;

   J := 1;
   while J <= B'LAST loop
      Result(I) := B(J);
      I := I + 1;
      J := J + 1;
   end loop;
   return Result;
end Concat;
```

Figure 5-1: The Generic Function `Concat`

Execution of an invocation of a generic instantiation of `Concat` consists of a sequence of *actions* that

include assignment (`I:=1`, `Result(I):=A(I)`), attribute evaluation (`A'LAST`, `B'LAST`), subscripting (`Result(I)`, `B(J)`), while loop test (`J <= B'LAST`) and object creation (for `I`, `J` and `Result`).

Some actions always operate on values of the same type, and each of these actions can be implemented the same way regardless of context.[29] Examples of such actions, which are known as *independent actions*, are **while** loop test, which always yields a boolean result, and addition, which operates on, and yields, integers.

The body of `Concat` also contains actions that are dependent on the types of the objects they deal with. These actions are called *dependent actions* and include assignment, subscripting and the relational operators. As a concrete example of a dependent action, consider assignment. If the the type of the operands to assignment is `Integer`, then this action can probably be implemented as a memory to memory move. But if the operands are of some array type, assignment will be more complicated, requiring a block transfer or a loop.

There are several *instances* of the assignment action within `Concat`. Some of the instances involve objects with fixed types:

```
I := 1;
I := I + 1;
J := 1;
I := I + 1;
J := J + 1;
```

These instances are known as *non-generic instances* because they can be implemented in a manner independent of the generic formal parameters.

On the other hand, there are instances of the assignment action that do involve objects with a generic type:

```
Result(I) := A(I);
Result(I) := B(J);
```

The implementations of these *generic instances* depend on the values of the generic formal parameters.

Recall that the idea of PR is to share a single subprogram among several generic instantiations. This

---

[29] This refers to the implementation in IL of these actions. The final code generated for an action may, of course, vary depending on its context.

means that the actions within the subprogram must be shared. Sharing instances of independent actions and instances of non-generic dependent actions is no problem because these instances operate with fixed types. The key to PR is to discover a way to share the generic instances of dependent actions. The basic strategy, described in the remainder of this chapter, is to treat generic instances as invocations of generic subprograms. This yields a consistent translation scheme: user-defined generic subprograms, which operate on objects whose types are unknown until run time, are treated identically to predefined operations that operate on objects whose types are also unknown until run time. This approach allows the compiler to use the same mechanisms to translate generic subprograms and predefined operations.

5.1. Instances as Subprogram Invocations

As mentioned above, generic instances are treated as subprogram invocations. Instances viewed in this manner, as subprogram invocations, are known as *composite instances*. The remaining instances, *primitive instances*, are implemented as primitive operations and have distinct syntactic representations in IL.

If the composite instances within the body of `Concat` were represented as subprogram invocations, `Concat` would appear as in figure 5-2. This translation may appear arbitrary: Assignment instances are treated like subprograms, but so so are the instances of addition, which is an independent action. The remainder of this section describes how an instance is determined to be composite or primitive.

If an instance is generic, it is always treated as composite. This is a convenient way to handle the type dependency of such an action. For example, the assignment

```
Result(I) := B(J);
```

is represented by the invocation

```
":="(Subscript(Result, I), Subscript(B, J));
```

This allows the compiler to represent the generic instance of assignment as an invocation of a subprogram named `":="`. All generic instances of assignment that are dependent on a specific generic type, in this case `Elt`, invoke a single subprogram.[30] The details of the type dependency are contained in a single locale.

Another reason for treating an instance as composite is the desire to use the single method *inline*

---

[30] The description of such subprograms appears in section 6.2.

```
function Concat(A, B : in Arr) return Arr is
   I, J : Integer;
   Result : Arr(1.."+"(LAST(A), LAST(B)));
begin
   ":="(I, 1);
   while "<="(I, LAST(A)) loop
      ":="(Subscript(Result, I), Subscript(A, I));
      ":="(I, "+"(I, 1));
   end loop;

   ":="(J, 1);
   while "<="(J, LAST(B)) loop
      ":="(Subscript(Result, I), SubScript(B, J));
      ":="(I, "+"(I, 1));
      ":="(J, "+"(J, 1));
   end loop;
   return Result;
end Concat;
```

Figure 5-2: Translation of `Concat` into Composite Instances, I

substitution to perform the bulk of the optimization in the compiler. For example, non-generic instances of assignment could be treated as primitive actions (they are composite actions in figure 5-2). Then it would be necessary to determine the final code for these assignment statements early in the translation. A decision would be made as to whether each assignment is better translated like a primitive assignment operation or like a routine invocation. Treating all instances of assignment as composite allows this decision to be delayed until the optimization phase. More information is known about the program then, and it is probable that better optimizations will be performed.

Of course, it is possible to treat assignment specially. Assignments would be represented by unique IL operators and decisions about the code for such operations delayed. But this requires extra mechanism in the compiler. Thus, for consistency, all instances of a dependent action are treated as composite. This allows a uniform handling of instances of these actions regardless of their context. An action that has the property that all of its instances are treated as composite is termed a *composite action.*

The composite actions may be made as small as desired, allowing any level of analysis. For example, it is possible to treat subprogram invocation, which is a primitive action in figure 5-1, as a sequence of simpler actions: for example, actual parameter evaluation, parameter binding (which might be fur-

ther broken down into type checking and value transmission) and subprogram entry.[31] This breakdown allows a more detailed analysis of invocations during optimization. Whether this is desirable depends on the particular implementation and target machine. For this work, treating subprogram invocation like a primitive action is sufficient (but see section 5.1.3 about parameter passing).

5.1.1. The Translation of Subscripting: Selectors

In figure 5-2 the assignment statement

```
Result(I) := A(I);
```

is represented as

```
":="(Subscript(Result, I), Subscript(A, I));
```

Assignment requires that its left-hand operand be an object, not a value, so the `Subscript` subprogram must return an object. A function returns a value, and so is inappropriate for the definition of `Subscript`. This is where selectors (section 3.2.2.1) are important. They allow object-yielding actions like `Subscript` to be treated like subprograms. For example, a subscript subprogram for arrays of type `String` could be defined as

```
selector Subscript(A : in out String; I : in Integer)
                   return Character;
```

Selectors can also be used in the definition of other operations that must yield an object: record component selection and array slicing,[32] for example. Without selectors it would be necessary to treat these actions specially. The use of selectors is an example of utilizing a small number of mechanisms throughout the compiler.

---

[31] For nice examples of the division of subprogram invocation into simpler actions see the paper by Hanson and Griswold [Hanson 78] and the paper by Lampson [Lampson 82].

[32] An array slice is an expression that yields a reference to a contiguous segment of an array value. This segment may be used as a variable:

```
A(1..2) := B(N..N+2);
```

The languages Ada and ALGOL68 [DOD 82, van Wijngaarden 77] allow array slices.

5.1.2. Variable Declarations

The generic function `Concat` contains several actions that create and destroy variables. Upon entry to the function, `Integer` objects for `I` and `J` are created, and `Result` is bound to a new object of type `Arr`. At exit from the function, the objects bound to `I` and `J` are destroyed. The object bound to `Result` cannot be destroyed because the value it contains must be returned from the function. (Function return is discussed in sections 5.1.4 and 6.5.)

The actions involved in the creation of `Result` are dependent on the generic type `Arr`. Thus, it would seem that variable declaration should consist of dependent actions. These actions are, in fact, dependent (and hence composite), but object manipulation is more complicated than the other composite actions. A discussion of the translation technique for variable declarations is delayed until section 6.1, and variable declarations are displayed in all examples without modification.

5.1.3. Parameter Passing

Most compilers generate code for several parameter binding methods. Having several methods allows parameters of different types to be passed efficiently. For example, a parameter with mode in and type `Integer` usually is passed most efficiently by value. On the other hand a parameter of an array type with mode **in** is likely to be passed by reference to avoid copying the array components.

In the procedure with specification

```
generic
   type T is limited private;
procedure Write(X : in T);
```

the binding method for the formal parameter `X` is dependent on the value of `T`. For example, the generic instantiation

```
procedure Int_Write is new Write(Integer);
```

has the specification

```
procedure Int_Write(X : in Integer);
```

and this subprogram will expect `X` to be passed by value. But the generic instantiation

```
procedure Str_Write is new Write(String);
```

has the specification

```
procedure Str_Write(X : in String);
```

and `X` will to be passed by reference.

Parameter passing is thus a dependent action. Like variable declaration, however, parameter passing is not treated as a normal composite action. If parameter passing were defined by a subprogram like other composite actions, the definition would be recursive because invoking the parameter passing subprogram would involve another parameter binding. The trick for avoiding this infinite regress is described in section 6.4.1.

5.1.4. Function Return

The classical way to return a value from a function is to place the value in a distinguished register [Gries 71]. This method is consistent and efficient, but works only for values that fit in a register. For larger values the easiest solution is to return the address of the value in the register. There are other techniques, but the manner in which a value is returned from a function is dependent on the type of the value. To see this, consider the generic function following:

```
generic
   type T is limited private;
   type Arr is array(Character range <>) of T;
   with function "<"(X, Y : in T) return Boolean;
function Min(A : in Arr) return T;

function Min(A : in Arr) return T is
   C : Character;
   Small : T;
begin
   Small := Arr(A'FIRST);
   C := SUCC(A'FIRST);
   while C <= A'LAST loop
      if A(C) < Small then
         Small := A(C);
      end if;
      C := SUCC(C);
   end loop;
   return Small;
end Min;
```

If function `Min` is instantiated as

```
type Map is array(Character range <>) of Character;

function Smin is new Min(Character, Map, Char_LT);
```

then `Small` will be of type `Character` and its value can be returned in a register. But `Min` may also be instantiated as

```
type Names is array(Character range <>) of String(1..10);

function Lex is new Min(String(1..10), Names, String_LT);
```

Now the local variable `Small` is of type `String(1..10)` and its value will not fit in a register.

Function return is a dependent action that is similar to parameter passing, which was discussed in the previous section. The solution to the implementation of function return, which is much like the implementation of parameter passing, is deferred until section 6.5.

5.2. Composite Actions are Generic

Figure 5-2 presents a view of the generic function `Concat` as composed of instances; some are composite, some are primitive. Composite instances are treated like subprogram invocations. This section discusses where and how these subprograms are defined.

For non-generic instances of dependent actions, such as the assignment operation in

```
":="(I, 1);
```

the appropriate subprogram could be declared in package `STANDARD`:

```
procedure ":="(Dest : out Integer; Source : in Integer) is
...
end ":=";
```

Appropriate subprograms for all composite actions over predefined types could be defined in a similar manner.

For non-generic instances of dependent actions on user-defined types, subprograms could be introduced implicitly by the compiler following the type definition like, for example:

```
type MyType is ...;

-- Inserted by compiler
procedure ":="(Dest : out MyType; Source : in MyType) is
...
end ":=";
```

But this does not work for generic instances of dependent actions, such as the assignment operation in

```
":="(Subscript(Result, I), Subscript(A, I));
```

from figure 5-2. It is not possible to provide a single definition for this assignment because the assignment must work for a number of types; it is generic. So the generic instances of assignment in `Concat` are treated as instantiations of a generic procedure, `ASSIGN`. The body of `Concat` (page 57) is viewed as

```
function Concat(A, B : in Arr) return Arr is
   -- Inserted by compiler
   procedure ":=" is new ASSIGN(Elt);
...
```

To be consistent, all assignment subprograms are treated as instantiations of the generic procedure `ASSIGN`. The assignment procedure for `Integer` now appears (implicitly) as

```
procedure ":=" is new ASSIGN(Integer);
```

Consistency demands further that all composite actions be treated as instantiations of generic subprograms. The translation of the body of `Concat` shown in figure 5-2 can now be rewritten as in figure 5-3. The specifications of compiler-introduced subprograms are displayed in *italics*.

```
function Concat(A, B : in Arr) return Arr is
   procedure ":=" is new ASSIGN(Elt);
   selector Subscript is new SUBSCRIPT(Integer, Elt, Arr);
   function LAST is new LAST(Integer, Elt, Arr);
   I, J : Integer;
   Result : Arr(1.."+"(LAST(A), LAST(B)));
begin
   ":="(I, 1);
   while "<="(I, LAST(A)) loop
      ":="(Subscript(Result, I), Subscript(A, I));
      ":="(I, "+"(I, 1));
   end loop;

   ":="(J, 1);
   while "<="(J, LAST(B)) loop
      ":="(Subscript(Result, I), Subscript(B, J));
      ":="(I, "+"(I, 1));
      ":="(J, "+"(J, 1));
   end loop;
   return Result;
end Concat;
```

Figure 5-3: Translation of `Concat` into Composite Instances, II

5.3. Primeval Subprograms

The generic subprograms that define the composite actions are known as *primeval subprograms*. The specifications of the primeval subprograms are given in package STANDARD. For example, the following declaration of the primeval subprogram ASSIGN appears in package STANDARD:

```
generic
   type T is limited private;      -- Nothing is demanded
procedure ASSIGN(Dest : out T; Source : in T);
```

It is not possible to write a body for ASSIGN in the source language. But this is no problem; as shown in section 3.2.5, package STANDARD is only a conceptual convenience. The bodies of the primeval subprograms are constructed by the compiler in the intermediate language IL (section 6.2).

Figures 5-4 and 5-5 provide the specifications of the predefined primeval subprograms.

5.4. Instantiating Composite Actions

When a type definition is encountered, the compiler creates implicitly a set of composite actions. Each action is defined as an instantiation of a primeval subprogram. For a non-generic type, the implicit definitions of the actions are placed after the type definition, but for a generic type the declarations are placed at the beginning of the declarations in the generic subprogram.

The actions that are created depend on the kind of the type: integer, enumeration, generic discrete, generic or non-generic array, private or limited private. Table 5-1 lists the actions created for each kind of type. Note that the actions created for integer, enumeration and discrete types are identical, as are the actions for non-generic and generic array types.

In figure 5-3, the implicit declarations of the composite actions have been limited to those actions used within the body of Concat. In reality, the compiler creates all of the actions for the types listed in table 5-1. During the optimization phase (section 8.2.3), those subprograms not invoked are deleted.

```
-- To instantiate ":="
generic
   type T is limited private;
procedure ASSIGN(Dest : out T; Source : in T);

-- To instantiate "="
generic
   type T is limited private;
function EQUAL(X, Y : in T) return Boolean;

-- To instantiate "/="
generic
   type T is limited private;
function NOT_EQUAL(X, Y : in T) return Boolean;

-- To instantiate "<"
generic
   type T is (<>);
function LESS_THAN(X, Y : in T) return Boolean;

-- To instantiate "<="
generic
   type T is (<>);
function LESS_THAN_OR_EQUAL(X, Y : in T) return Boolean;

-- To instantiate ">"
generic
   type T is (<>);
function GREATER_THAN(X, Y : in T) return Boolean;

-- To instantiate ">="
generic
   type T is (<>);
function GREATER_THAN_OR_EQUAL(X, Y : in T) return Boolean;
```

Figure 5-4: Primeval Subprograms, Part I

```
-- To instantiate PRED
generic                              -- Only for discrete types
   type T is (<>);
function PRED(X : in T) return T;

-- To instantiate SUCC
generic                              -- Only for discrete types
   type T is (<>);
function SUCC(X : in T) return T;

-- To instantiate FIRST
generic                              -- Discrete index type
   type Index is (<>);
   type Element is limited private;
   type Arr is array(Index range <>) of Element;
function FIRST(A : in Arr) return Index;

-- To instantiate LAST
generic                              -- Discrete index type
   type Index is (<>);
   type Element is limited private;
   type Arr is array(Index range <>) of Element;
function LAST(A : in Arr) return Index;

-- To instantiate Subscript
generic                              -- Discrete index type
   type Index is (<>);
   type Element is limited private;
   type Arr is array(Index range <>) of Element;
selector SUBSCRIPT(A : in out Arr; I : in Index)
                  return Element;
```

Figure 5-5: Primeval Subprograms, Part II

Type	Actions
Integer	":=", "=", "/=", "<", "<=", ">", ">=", PRED, SUCC
Enumeration	":=", "=", "/=", "<", "<=", ">", ">=", PRED, SUCC
Generic discrete	":=", "=", "/=", "<", "<=", ">", ">=", PRED, SUCC
Non-generic array	":=", "=", "/=", FIRST, LAST, SUBSCRIPT
Generic array	":=", "=", "/=", FIRST, LAST, SUBSCRIPT
Private	":=", "=", "/="
Limited private	*none*

Table 5-1: Composite Actions for Types

Chapter 6
Polymorphic Routine Translation: Implementation

An implementation of PR must handle consistently both user-defined generic subprograms and predefined generic subprograms. In addition, the implementation technique should require few specialized intermediate language (IL) constructs, allowing the compiler's standard optimizations to be used.

One way to achieve this goal is to translate a generic subprogram into an IL routine by the addition of an *implicit formal parameter* for each generic formal parameter. When the routine is invoked, each implicit formal parameter is bound to an *implicit actual parameter*. The implicit actual parameters, which are passed from the generic instantiation, are run-time representations of the generic actual parameters in the instantiation. Within the body of the generic subprogram, generic instances of actions are translated into IL code that references the implicit formal parameters.

For example, the generic procedure `Sort`

```
generic
   type Index is (<>);
   type Elt is private;
   type IE is array(Index range <>) of Elt;
   with function LT(A, B : in Elt) return Boolean;
procedure Sort(A : in out IE);
```

is translated into an IL routine with the following heading:[33]

```
routine Sort(A, Index, Elt, IE, LT) = . . .
```

The implicit formal parameters `Index`, `Elt`, `IE` and `LT` have been added to the list of explicit formal parameters. `Index`, `Elt` and `IE` represent types; `LT` will be bound to a subprogram.

---

[33] The descriptors for the formal parameters have been omitted because the kind of descriptor for the parameter `A` is not introduced until section 6.1.

The PR translation of a program part that is not within a generic subprogram (and is not a generic instantiation) is straightforward and is not discussed in this dissertation. Chapter 5 described the constructs that need to be translated specially by PR, and the remainder of the current chapter discusses the translation of these constructs. Section 6.1 describes the translation of variable declarations. Their translation is tricky because it involves storage manipulation, which is difficult to integrate smoothly into a subprogram-oriented translation scheme. The translation of generic instances of dependent actions is discussed in section 6.2, which is concerned with translating the body of a generic subprogram. Section 6.3 discusses the translation of a generic instantiation, which requires an IL representation for generic actual parameters. The implementations of parameter passing and function return are described in sections 6.4 and 6.5. The translation and manipulation of the three kinds of generic parameters—types, subprograms and objects—is covered in sections 6.6, 6.7 and 6.8.

6.1. Translating Variable Declarations

Variable declaration consists of dependent (and hence composite) actions. It is not practical, however, to represent object creation and destruction like other composite actions—by subprogram invocations—due to the nature of storage management in a stack-oriented system. Assume that the actions comprising object manipulation were treated as normal composite actions. There would be two actions, say `Create` and `Destroy`. Within the body of `Concat` (figure 5-1), the local variables `I`, `J` and `Result` would be manipulated as follows:

```
   . . .
begin
   Create(I);     Create(J);
   Create(Result);
   ":="(I, 1);
   . . .
   Destroy(J);    Destroy(I);
   return Result;              -- Result can't be destroyed
end Concat;
```

This is an unsatisfactory solution. The routines `Create` and `Destroy` are subprograms and each must have a subprogram frame.[34] But in a stack-oriented system subprogram frames are allocated from the same stack as local storage. An execution of `Create` allocates storage for `I`, `J` or `Result`. Exit from a subprogram causes all storage used by the subprogram to be relinquished, and this would deallocate the storage acquired by `Create`, destroying the local variables before their use. Of course, there are ways around this problem: separate stacks can be used for subprogram frames and

[34] For a discussion of subprogram frames, see chapter 8 of Gries' compiler book [Gries 71].

local storage, or local storage can be allocated from a heap. But these solutions are inefficient and violate the stack-oriented discipline.

In addition to the local storage lifetime problem, treating object manipulation actions like normal composite actions leads to further complications. One would hope that an object with a size determinable at compile time would be allocated in the static part of the subprogram frame. But consider the declarations of I and J in function `Concat`. To allocate I and J in the static part of the subprogram frame, the compiler must first note that the sizes of these variables are compile-time determinable (this is no problem). The compiler must then remove the corresponding invocations of `Create` and `Destroy` from the subprogram body. This would require special knowledge of the compiler about the `Create` and `Destroy` procedures.

To avoid this extra mechanism, the implementation in this thesis treats object creation and destruction in a special manner. As discussed in section 1.4.3, each variable in IL has an associated descriptor. For example, in the declaration

```
local I : desc(Integer);
```

the descriptor

```
desc(Integer)
```

specifies that I is to be bound to the address of an object capable of holding an integer value. The compiler treats this declaration as if it indicated calls of `Create` and `Destroy` at appropriate program points. This representation keeps the allocation and deallocation information connected tightly to the variable declaration and, thus, allows the compiler to easily manipulate the actions.

The creation of local variables of a generic type requires some additional mechanism. To describe the possible types of a variable of a generic type, the Union descriptor specifies a list of descriptors for an object. For example, in the following:

```
generic
   type T is private;
procedure Exchange(X, Y : in out T);

procedure Exchange(X, Y : in out T) is
   Temp : T;
begin
   Temp := X;
   X := Y;
   Y := Temp;
end Exchange;
```

```
procedure Int_Swap is new Exchange(Integer);
procedure String_Swap is new Exchange(String(1..10));
```

the translation to IL of the declaration of Temp is

```
local Temp : desc(Union, Integer_Desc,
                  desc(Array,
                       Integer_Desc, Character_Desc,
                       1, 10));
```

This descriptor indicates that at creation time the variable Temp will be bound to the address of an object with one of the specified descriptors. During its lifetime (an invocation of Exchange) Temp remains bound to this object.[35]

To inform the compiler of the descriptor to use for creating a variable with a Union descriptor the declaration is annotated with an *allocation expression*:

```
local Temp : desc(Union, Integer_Desc,
                  desc(Array,
                       Integer_Desc, Character_Desc,
                       1, 10))
             alloc <expression>;
```

The *<expression>* must yield a descriptor equal to one specified in the union descriptor. This descriptor will be used to create the object for Temp. Storage allocation for such locals is discussed in section 8.2.5.

6.2. Translating the Generic Body

The body of a non-primeval generic subprogram is translated like the body of a non-generic subprogram.[36] Only constructs referencing directly a generic parameter necessitate special handling in a generic subprogram. The representation and manipulation of generic parameters is discussed in sections 6.6 (type parameters), 6.7 (subprogram parameters) and 6.8 (object parameters).

Constructs involving a generic parameter indirectly require no special handling. For example, the following assignment statement, from Concat:

```
Result(I) := B(J);
```

---

[35] The Union descriptor in IL is not the same as the union mode in ALGOL68 [van Wijngaarden 77]. A variable in ALGOL68 with a union mode may change the type of its associated object during its lifetime.

[36] Primeval subprograms require special treatment because their bodies are not expressed in the source language. This is explained later in this section.

involves the generic type `Elt` indirectly because it is the type of the expressions in the assignment. No regard need be taken of this dependence during translation because the assignment is translated into

```
":="(Subscript(Result, I), Subscript(B, J));
```

Overload resolution resolves the reference to `":="` to be the local implicit instantiation of `":="` (figure 5-3). The dependence on the value of `Elt` is contained entirely in this instantiation of `":="`.

The body of a primeval subprogram is not expressed in the source language. For this reason, and also because the body is implementation-dependent and machine-dependent, the body of a primeval subprogram is constructed by the compiler during the INFO phase (section 8.2.1). During this phase the instantiations of each primeval subprogram are found. For each primeval subprogram the compiler determines the set of all non-generic types bound to the generic type parameters.[37] This set of types is used to direct construction of the body of each primeval subprogram.

For example, consider the primeval subprogram `Assign` with specification

```
generic
   type T is limited private;
procedure ASSIGN(X : out T; Y : in T);
```

and assume that a program contains the following (implicit) instantiations of `ASSIGN`

```
procedure ":=" is new ASSIGN(Integer);
procedure ":=" is new ASSIGN(Character);
procedure ":=" is new ASSIGN(String);
```

The compiler constructs the body of `ASSIGN` so that it will perform assignment for objects of type `Integer`, `Character` or `String`. The body of a primeval subprogram is a case statement, with one arm for each possible descriptor kind. In the example for `ASSIGN`, there are three possible descriptor kinds: Integer, Enumeration[38] and Array. Thus, `ASSIGN` is translated as

---

[37] Section 8.2.1 describes the method used to collect this information.

[38] The predefined type `Character` is an enumeration type (appendix B).

```
routine ASSIGN(Dest, Source, T) =    ! T is implicit parameter39
begin
   case kind(.T) of  ! See what kind of descriptor
      when Integer =>
         code to perform integer assignment;
      when Enumeration =>
         code to perform enumeration assignment;
      when Array =>
         code to perform array assignment;

   esac
end ASSIGN;
```

The code in each arm is implementation-dependent and machine-dependent. During the INFO phase the compiler reads the IL code for all possible case arms from a file. To retarget the compiler for a different machine, it is necessary to modify the IL file of actions. No further modifications are required, because all manipulations of the IL are performed by the standard compiler mechanisms.

In section 2.1.2, the EL1 generic statement was credited as the inspiration for the structure of the body of a primeval subprogram. In EL1, the programmer is responsible for using the **generic** statement to define a generic subprogram. The EL1 compiler uses the information contained in the arms of the generic statement to optimize calls of a generic subprogram. In this dissertation, the compiler creates the bodies of primeval subprograms automatically. The compiler attempts to use the information in the arms of the case statement to optimize instantiations of the primeval subprogram. The method used to accomplish this is described in section 7.3.

6.3. Translating the Generic Instantiation

A generic instantiation is translated into a routine, the body of which is a call of the target generic subprogram. This call passes two sets of parameters: the explicit parameters (those specified in the subprogram specification) and the implicit parameters (those corresponding to the generic parameters). For example, the instantiation

```
function Cat is new Concat(Character, String);
                        -- Concat in figure 5-1
```

is translated to

```
routine Cat(A, B) =
begin
   Concat(.A, .B, Character_Desc, String_Desc)
end Cat;
```

---

[39] It will be seen in section 6.6 that T is bound to a descriptor at each invocation of **Assign**.

The descriptor `Character_Desc` corresponds to the generic actual parameter `Character` and `String_Desc` corresponds to `String`. Note that the routine has a formal parameter corresponding to each formal parameter of `Cat`'s specification, which is

```
function Cat(A, B : in String) return String;
```

Unfortunately, the translation of an instantiation is complicated by parameter passing—a complex issue in PR. The next section discusses the translation of parameter passing. The passing of implicit parameters is described in sections 6.6 through 6.8.

6.4. Translating Parameter Passing

The semantics of a language may necessitate the use of several parameter passing methods. For example, a language may provide a binding mode that guarantees that a formal parameter act as a local copy of the actual parameter. In some cases, the compiler will be able to pass the address of the actual parameter and avoid an expensive copying operation. But in the presence of aliasing among actual parameters the compiler may be forced to copy the parameter.

Apart from semantics, execution efficiency considerations may compel a compiler to generate code that employs multiple parameter passing methods. A common example, mentioned above, is to pass the address of a large object to avoid the time and space overhead required by copying.

The most common parameter passing methods are *call-by-value* and *call-by-reference*.

In *call-by-value*, the value of the actual parameter is copied to the formal parameter, which then acts as a local variable within the subprogram. Call-by-value saves execution time when the value of the parameter occupies a conveniently sized storage unit and passing the parameter value is as inexpensive as passing the address of the parameter. On most machines a word is such a unit.

In *call-by-reference*, the address of the actual parameter is copied to the formal parameter. This method is useful when the formal parameter must be an alias for the actual parameter, so modifications to the formal parameter affect the actual parameter. Call-by-reference is also useful when copying the value of the actual parameter, as required for call-by-value, would be expensive.[40]

The semantics of parameter passing in the source language (section 3.2.2) allows call-by-reference to

[40] This technique for avoiding a copying operation can be used only if the language does not allow the value of the formal parameter to be modified.

be used for all parameters. For efficiency, however, the experimental compiler uses call-by-value for discrete parameters of binding mode in.

To explain the intricacies of parameter passing in PR translations it is convenient to consider a concrete example of multiple parameter passing methods. For this purpose, the parameter passing methods of the experimental compiler are used in this chapter. Other parameter passing methods may be desirable or required. The techniques in this chapter can be modified to apply to any parameter passing method that can be represented in the intermediate language. To demonstrate this, the remainder of this section discusses the implementation of the methods *call-by-result*, *call-by-value-result* and *call-by-name*.

In call-by-result the value of the formal parameter is copied to the actual parameter at routine exit. Call-by-value-result is similar to call-by-result but, in addition, the value of the actual parameter is copied to the formal parameter at routine entry. These methods are useful when there are different representations or different semantics for the types of the actual and formal parameters. This situation occurs in Ada, and call-by-result and call-by-value-result are the methods of choice in such cases.

Call-by-result and call-by-value-result can be implemented by creating a temporary variable for the actual parameter. This variable is passed by reference, in place of the actual parameter, and at routine exit the value of this temporary is copied to the actual parameter. (Call-by-value-result requires an additional copying operation prior to routine entry). Because these two methods can be simulated using call-by-reference, the methods in this chapter apply without modification.

The language ALGOL 60 [Naur 63] required the use of call-by-name. In call-by-name, each reference to a formal parameter requires that the corresponding actual parameter be reevaluated in its original context. Call-by-name is expensive, but it is simple to implement by representing an actual parameter by a small routine that evaluates the actual parameter.[41] A formal parameter is referenced by invoking the routine passed to the parameter. The method can thus be implemented by creating routines that are passed by value, and the methods in this chapter apply without modification.

[41] A nice discussion of the implementation of call-by-name can be found in chapter 8 of the book *Compiler Construction for Digital Computers* [Gries 71].

6.4.1. Passing Parameters of a Generic Type

Call-by-reference is a general method because the value of an object can always be obtained from its address. The experimental compiler could, thus, use call-by-reference for all parameters. This buys some simplification in the compiler but may exact stiff code space and execution time penalties. Programs generally have a large number of discrete parameters of mode in, and these parameters could be passed by value. Should call-by-reference be used, each reference to a formal parameter would require an additional level of dereferencing. This extra dereferencing requires additional code to implement and more time to execute.

For a formal parameter of a non-generic type, the compiler can choose the most efficient parameter passing method. For a parameter of a generic type, however, the compiler does not know the type of the parameter prior to an invocation of the subprogram. Since the parameter passing method for a formal parameter must be identical for all invocations, the compiler must choose a method that is consistent for all types the parameter may have. In the worst case, the method used is call-by-reference.

Assuring a consistent parameter passing method for a formal parameter of a generic type guarantees that the body of the generic subprogram can be translated. It is now necessary to transmit an actual parameter to a generic subprogram using the expected method, which may be different than the method desired for that parameter. This becomes a problem within the body of a generic instantiation when a parameter of a non-generic type must be bound to a parameter of a generic type.

To understand the problems involved in passing parameters of a generic type, it is necessary to understand the implementations of call-by-value and call-by-reference. As an example, consider the following two IL routines that yield the sum of the values of two integer parameters:

```
! X & Y passed by value

routine Plus1(X, Y : desc(Integer)) =
begin
   return .X + .Y
end Plus1;

! X & Y passed by reference

routine Plus2(X, Y : desc(Pointer, desc(Integer))) =
begin
   return ..X + ..Y
end Plus2;
```

Routine `Plus1` assumes that `X` and `Y` will be passed by value, so only one level of dereferencing is

needed to obtain a parameter's value. A call of `Plus1` to add the values of variables `A` and `B` would be

```
Plus1(.A, .B);
```

In `Plus2`, `X` and `Y` are passed by reference. The expression `.X` yields the address of the object passed to `X`. It is necessary to use two levels of dereferencing, `..X`, to obtain the value of the formal parameter. A call of `Plus2` with an equivalent effect to the call of `Plus1` above is

```
Plus2(A, B);
```

Table 6-1 gives the binding methods used by the experimental compiler for each non-generic type in each possible binding mode.

When passing a non-generic parameter the most efficient binding for that parameter's type and mode may be used. For a parameter of a generic type, however, the compiler must take into account all possible types for the parameter.

	Mode		
Type	***in***	***out***	***in out***
Integer	*value*	*reference*	*reference*
Enumeration	*value*	*reference*	*reference*
Non-generic array	*reference*	*reference*	*reference*

Table 6-1: Bindings for Non-generic Types

To see this, consider the following generic procedure and instantiations

```
generic
   type T is private;
procedure Counted_Assign(X : out T; Y in T);

procedure Counted_Assign(X : out T; Y in T) is
begin
   Assignment_Count := Assignment_Count + 1;
   X := Y;
end Counted_Assign;

procedure Int_Assign is new Counted_Assign(Integer);
procedure String_Assign is new Counted_Assign(String);
```

In this example `X` and `Y` may be of type `Integer` or of type `String`. For `Integer` objects, the desired binding for `Y` is call-by-value. But call-by-reference is desirable for objects of type `String`. The compiler must use the most general method and pass `Y` by reference.

Several binding methods may be used for passing parameters of generic types. For private or limited private types, the worst must be assumed and call-by-reference used. An implementation may, however, be able to treat generic discrete types in a more efficient manner. For example, the experimental compiler takes advantage of the fact that all values of a discrete type may be stored in a single word on the target machine, the PDP-20 [DEC 78b]. This allows parameters of a generic discrete type to be passed by value, if their binding mode is **in**. Table 6-1 can now be expanded to include the binding methods used by the experimental compiler for generic types. This is done in table 6-2.

Type	Mode		
	in	*out*	*in out*
Integer	*value*	*reference*	*reference*
Enumeration	*value*	*reference*	*reference*
Non-generic array	*reference*	*reference*	*reference*
Generic discrete	*value*	*reference*	*reference*
Generic array	*reference*	*reference*	*reference*
Private	*reference*	*reference*	*reference*
Limited private	*reference*	*reference*	*reference*

Table 6-2: Bindings for All Types

The compiler could take advantage of instances where a parameter of a private type could be passed by value. This could happen, for example with the formal parameter `Y`, if all instantiations of `Counted_Assign` had a discrete type corresponding to `T`. But this leads to trouble within the bodies of primeval subprograms, which are defined prior to compilation. This will be explained in section 6.4.1.2.

6.4.1.1. Interface Points

The following subprogram, `Int_Assign`:

```
procedure Int_Assign is new Counted_Assign(Integer);
```

has the specification

```
procedure Int_Assign(X : out Integer; Y : in Integer);
```

Calls of this instantiation will pass the actual parameter corresponding to `Y` by value (table 6-2). Recall that the body of `Int_Assign` is a call of the generic routine `Counted_Assign` and that `Counted_Assign` expects the explicit parameters to be passed by reference. This will cause improper references to `Y` during execution of `Counted_Assign`.

When Y is passed, the value of a variable of a non-generic type (`Integer`) is copied to a variable of a generic type (`private`). Places like this, where a value with a non-generic type is treated like a value of a generic type, or vice versa, are known as *interface points*. At these points a conversion between binding methods may be necessary. There are three possible interface points when using the PR scheme:

1. the call that forms the body of a generic instantiation (discussed in this section);
2. within the body of a primeval subprogram (discussed in section 6.4.1.2);
3. the invocation of a generic subprogram parameter (discussed in section 6.7).

In the example of `Counted_Assign`, the call from within `Int_Assign` must pass the explicit parameters in the manner expected by the generic routine, as follows:

```
routine Int_Assign(X : desc(Pointer, desc(Integer)),
                   Y : desc(Integer)) =

begin
   Counted_Assign(.X, Y, desc(Integer))
end Int_Assign;
```

The generic routine `Counted_Assign` expects `X` to be passed by reference. As `Int_Assign` also expects its parameter `X` to be passed by reference, all that is necessary is to dereference `X` before passing. But the parameter `Y` is passed to `Int_Assign` by value. To satisfy `Counted_Assign`, which expects its parameter `Y` to be passed by reference, the parameter `Y` is passed without dereferencing. This passes the address of `Y`.

The subprogram `String_Assign`, defined as

```
procedure String_Assign is new Counted_Assign(String);
```

has the specification

```
procedure String_Assign(X : out String; Y : in String);
```

and calls of this instantiation will pass both actual parameters by reference. As `Counted_Assign` also expects its parameters by reference, `X` and `Y` are dereferenced once before being passed. No binding conversion is needed. The translation of `String_Assign` is thus

```
routine String_Assign(X, Y : desc(Pointer, String_Desc),) =
begin
   Counted_Assign(.X, .Y, String_Desc)
end String_Assign;
```

During construction of the call that forms the body of a generic instantiation, the compiler determines the necessary conversions by examining the binding methods for each pair of corresponding

formal parameters: one from the instantiation and one from the generic subprogram. If the bindings are the same, a dereference of the formal parameter is adequate. If a parameter passed by reference is being bound to a parameter expected to be passed by value, then a *dereferencing conversion* is necessary. This consists of applying an additional level of dereferencing to the formal parameter. If a parameter passed by value is being bound to a parameter expected to be passed by reference, then an *upreferencing conversion* is applied. This is done by passing the undereferenced parameter (and, thus, its address) to the generic routine.

Table 6-3 lists the binding conversions used in the experimental compiler. Next to each binding method is an IL expression that illustrates the appropriate binding conversion, assuming that the formal parameter in question is *F*.

	Binding at Generic Routine			
Binding at Instantiation	***value***	example	***reference***	example
value	*none*	*.F*	upreference	*F*
reference	dereference	*..F*	*none*	*.F*

Table 6-3: Parameter Conversions

6.4.1.2. Interface Points in a Primeval Subprogram

The bodies of primeval subprograms are constructed by the compiler from IL instructions read from a file (section 6.2). This IL code must know how to reference the formal parameters to the primeval subprogram. To do this, the binding methods of the parameters must be known. This implies that each parameter to a primeval subprogram must always be bound in the same manner. To achieve this, the compiler always uses the methods listed in table 6-2 for generic type parameters.

Within the body of the primeval subprogram `ASSIGN`, for example, it is known a priori that the formal parameters `Dest` and `Source` are both passed by reference. Thus, within the arm for integer assignment, the IL code following is used:

```
when Integer =>
   .Dest ← ..Source;          ! Extra dereference required
```

It would be impossible to use this code if the bindings were not known prior to compilation.

6.5. Translating Function Return

Just as there are several methods for parameter passing, there are several methods for yielding function values. The most common function return methods are *return-by-value* and *return-by-reference*. A value that fits in a register can be returned by value. But a larger value is returned by placing its address in a register; this is return-by-reference. Furthermore, return-by-reference is the more general method, and a compiler could employ this technique alone. But as with parameter passing, this is likely to lead to inefficient implementation.

There are also a set of interface points with function return, like those for parameter passing:

1. returning a value from the body of a generic instantiation;
2. returning a value from a primeval function;
3. returning a value from the invocation of a generic subprogram parameter.

These interface points are handled like the interface points for parameter passing, generating return conversions when necessary. The reader should be able to extrapolate the actions required.

Function return in the source language presents an additional difficulty that is independent of generic types. The problem occurs for a function whose return type is an array type specified without bounds. For example,

```
function Cat(X, Y : in String) return String;
```

This is legal in the source language and indicates that the type of the return value is `String` but that the bounds will be unknown until the value is returned.

Allocation and deallocation of storage for the return values of such functions is messy. There are elegant solutions to this dilemma, but they are beyond the scope of this dissertation. The interested reader may consult the paper by Moss [Moss 78] for a discussion of several solutions.

6.6. Type Parameters

At run time a type is represented by an IL descriptor. For example, the generic instantiation

```
procedure Int_Swap is new Exchange(Integer);
-- Exchange from page 71
```

is translated into

```
routine Int_Swap(X, Y : desc(Pointer, desc(Integer))) =
   Exchange(.X, .Y, desc(Integer))
end Int_Swap;
```

Descriptors are created for a type only if necessary. A descriptor is necessary for a type if the type is used as a generic actual parameter, or if the creation and destruction of an object is implemented by explicit calls on `Create` and `Destroy` routines.

Initially, all object creation and destruction is represented implicitly by calls of `Create` and `Destroy` routines (section 6.1). The need for most of these implicit calls is removed during the OPT phase (section 8.2.3). But variables of a complicated type may require large amounts of code to create and destroy. In these cases the implicit calls to `Create` and `Destroy` become explicit. In addition, the translation of a variable of a generic type will require explicit calls of `Create` and `Destroy`. For example, the declaration of `Temp` from the generic procedure `Exchange` (page 71) is translated ultimately as

```
constant Inst_Desc = desc(Union, desc(Integer),
                                 desc(Array,
                                      desc(Integer),
                                      Character_Desc,
                                      1, 10));

routine Exchange(X, Y : desc(Pointer, Inst_Desc),
                 T : desc(Descriptor)) =
begin
   local Temp : Inst_Desc;

   Create(Temp, .T);
   . . .
   Destroy(Temp, .T)
end Exchange;
```

A type used like a generic actual parameter may contain expressions with side effects. As in the GE method, these expressions require special handling to insure proper semantics. Otherwise, the expressions will be evaluated each time the subprogram defined by the generic instantiation is invoked. The same solution that works for GE will suffice for PR, and the reader is referred to section 4.1 for this solution.

6.7. Subprogram Parameters

The implementation of subprograms as parameters is a well-understood technique. The representation for a subprogram parameter is a tuple:

(*address of* IL *routine*, *address of active display for* IL *routine*)

The *address of active display* is necessary to establish the proper addressing context for the sub-

program parameter during its invocation.[42]

The proper handling of displays is part of the semantics of IL, so a routine actual parameter is represented in IL by its name. The code generator is responsible for passing the appropriate structure for a routine.

To invoke a routine parameter, it is necessary to dereference the parameter before its invocation. For example, within the body of `Sort` (whose specification is given on page 69) the call

```
LT(X, Y)            -- source language
```

is translated as

```
(.LT)(.X, .Y)       ! IL
```

In general, however, the routine passed as an implicit actual parameter cannot be the translation of the subprogram that appeared as the generic actual parameter. For example, consider the generic procedure `Debug_Write`:

```
generic
   type T is limited private;
   with procedure Write(X : in T);
procedure Debug_Write(Val : in T);

procedure Debug_Write(Val : in T) is
begin
   Output("**Debug: ");
   Write(Val);
end Debug_Write;
```

This procedure is translated to

```
routine Debug_Write(Val : desc(Pointer, desc(Union, as appropriate)),
                    T : desc(Descriptor),
                    Write : desc(Routine)) =
begin
   Output("**Debug: ");
   (.Write)(.Val)
end Debug_Write;
```

`Val` is dereferenced once in the call of the subprogram parameter `Write`. This is because `Debug_Write` must assume that the parameter to `Write` is passed by reference, as the formal parameter `X` is of a private type. Because `Val` is passed by reference also, no conversion is used when passing `Val` to `Write`. But `Write` may be bound to a subprogram with the following specification:

---

[42] This discussion assumes that the language does not allow call-by-name binding for formal parameters. A thorough discussion of an implementation technique for subprogram formal parameters, including call-by-name binding, is contained in chapter 8 of the book by Gries [Gries 71].

```
procedure Octal_Write(N : in Integer);
```

as in

```
procedure Memory_Debug is
                new Debug_Write(Integer, Octal_Write);
```

`Octal_Write` expects its parameter to be passed by value, but it will receive `N` by reference from `Debug_Write`. This is another interface point: passing parameters in the invocation of a generic formal subprogram parameter. This problem is solved by the construction of an *interface routine*, which serves as an interface between `Debug_Write` and `Octal_Write` as follows:

```
routine Octal_Write(N : desc(Integer)) = ...;          ! Unchanged

routine Interface_Octal_Write(N : desc(Pointer, desc(Integer)) =
begin
   Octal_Write(..N)              ! A dereferencing conversion
end Interface_Octal_Write;

routine Memory_Debug(N : desc(Integer)) =
begin
   Debug_Write(N, Integer_Desc, Interface_Octal_Write)
                        ! an upreferencing conversion of N
end Memory_Debug;
```

Each time a subprogram is used as a generic actual subprogram parameter, the compiler inspects the bindings expected by the actual subprogram and by the formal subprogram. If the bindings are identical, no interface routine is needed. If there are any differences in the bindings, an interface routine is constructed to call the actual routine with appropriate binding conversions. This interface routine is passed in place of the actual routine within the body of the translated generic instantiation. If an interface routine for the actual subprogram with the appropriate conversions has already been constructed, it is reused.

6.8. Object Parameters

An object that is used as a generic actual object parameter is treated like an explicit parameter with the same type and binding. But generic formal object parameters have a slightly different semantics than normal object parameters. For generic formal object parameters the expression denoting the object is evaluated once, at instantiation time. The same problem with object parameters appears in GE (section 4.1) and the same solution applies.

6.9. An Example: The PR Translation of Concat

This section contains a PR translation of the generic function `Concat` (figure 5-1). It is assumed that the following generic instantiations exist for `Concat`:

```
function Cat is new Concat(Character, String);

type Flags is array(Integer range <>) of Boolean;
function Append is new Concat(Boolean, Flags);
```

The translation of these declarations appears immediately following the translation of `Concat`. To improve readability, the definitions of compiler-generated routines that are not invoked have been omitted.

The translation begins with declarations of the descriptors required by the program. The descriptors are followed by the definitions of the necessary primeval routines: `ASSIGN`, `LESS_THAN_OR_EQUAL`, `LAST` and `SUBSCRIPT`. Next are the generic instantiations of the primeval subprograms that serve to define the actions for the predefined types, such as assignment for `Integer`. Finally, there is the translation of the generic subprogram `Concat`.

```
!*************************************!
! PR Translation of function Concat   !
!          (figure 5-1)               !
!*************************************!

! Declarations from package STANDARD

constant Integer_Desc = desc(Integer);
constant Boolean_Desc = desc(Boolean);
constant Character_Desc = desc(Enumeration, 128);
constant String_Desc = desc(Array, Integer_Desc,
                            Character_Desc);

! Descriptor for type parameters
constant Desc_Desc = desc(Descriptor);

! Descriptor for user-defined type Flags
constant Flags_Desc = desc(Array, Integer_Desc,
                           Boolean_Desc);
```

```
! Descriptors for generic formal parameters
constant Elt_Desc = desc(Union, Boolean_Desc,
                                Character_Desc));

constant Ref_Elt_Desc = desc(Pointer, Elt_Desc);
constant Arr_Desc = desc(Pointer, desc(Union,
                                       String_Desc,
                                       Flags_Desc));

constant Ref_Ass_Desc = desc(Union, desc(Union,
                                         Integer_Desc
                                         Boolean_Desc,
                                         Character_Desc));


constant Index_Desc = desc(Union, Integer_Desc);

! Primeval routines

! Primeval assign; possible types of
!                  actuals are {Integer, Boolean, Character}
routine ASSIGN(Dest, Source : Ref_Ass_Desc, T : Desc_Desc) =
begin
   case kind(.T) of
      when Integer =>
         .Dest ← ..Source
      when Boolean =>
         .Dest ← ..Source
      when Enumeration =>
         .Dest ← ..Source
   esac
end ASSIGN;

! Primeval less than or equal; possible types of actuals
!                              are {Integer}
routine LESS_THAN_OR_EQUAL(X, Y : Index_Desc,
                           T : Desc_Desc) =
begin
   case kind(.T) of
      when Integer =>
         return ..X ≤ ..Y
   esac;
end LESS_THAN_OR_EQUAL;

! Primeval LAST attribute; possible types for A are {String, Flags}
routine LAST(A : Arr_Desc, Index, Element, Arr : Desc_Desc) =
begin
   case kind(.Arr) of
      when Array =>
         return .(.A+UB_Offset)     ! Offset in dope vector
                                    ! of upper bound
   esac
end LAST;
```

```
SS

! Primeval SUBSCRIPT; possible types for A are {String, Flags},
!                      possible types for I are {Integer}
routine SUBSCRIPT(A : Arr_Desc, I : Index_Desc,
                  Index, Element, Arr : Desc_Desc) =

begin
   case kind(.Arr) of
      when Array =>
         return subscript(.A, .I)

   esac
end SUBSCRIPT;

! Instantiation of actions for Integer
routine ASSIGN_Integer(Dest : desc(Pointer, Integer_Desc),
                       Source : Integer_Desc) =

begin
   ASSIGN(.Dest, Source, Integer_Desc)
end ASSIGN_Integer;

routine LESS_THAN_OR_EQUAL_Integer(X, Y : Integer_Desc) =
begin
   return LESS_THAN_OR_EQUAL(X, Y, Integer_Desc)
end LESS_THAN_OR_EQUAL_Integer;

! Integer addition
routine PLUS(X, Y : Integer_Desc) =
begin
   return .X + .Y
end PLUS;

! Translation of Concat

routine Concat(A, B : Arr_Desc, Elt, Arr : Desc_Desc) =
begin
   ! Definitions of actions for generic types

   routine ASSIGN_Elt(Dest, Source : Ref_Elt_Desc) =
   begin
      ASSIGN(.Dest, .Source, .Elt)
   end ASSIGN_Elt;

   routine LAST_Arr(A : Arr_Desc) =  ! LAST attribute for Arr arrays
   begin
      return LAST(.A, Integer_Desc, .Elt, .Arr)
   end LAST_Arr;

   routine SUBSCRIPT_Arr(A : Arr_Desc, I : Integer_Desc) =
   begin
      return SUBSCRIPT(.A, .I, Integer_Desc, .Elt, .Arr)
   end SUBSCRIPT_Arr;

   ! End of action definitions
```

```
    local I, J : Integer_Desc;
    local Result : desc(Array, Integer_Desc,
                        .Elt,    ! Run-time descriptor expression
                        1,
                        PLUS(LAST_Arr(.A), LAST_Arr(.B)));

    ASSIGN_Integer(I, 1);
    while LESS_THAN_OR_EQUAL_Integer(.I, LAST_Arr(.A)) do
       ASSIGN_Elt(SUBSCRIPT_Arr(Result, .I),
                  SUBSCRIPT_Arr(.A, .I));
       ASSIGN_Integer(.I, PLUS(.I, 1))
    od;

    ASSIGN_Integer(J, .I);
    while LESS_THAN_OR_EQUAL_Integer(.J, LAST_Arr(.B)) do
       ASSIGN_Elt(SUBSCRIPT_Arr(Result, .I),
                  SUBSCRIPT_Arr(.B, .J));
       ASSIGN_Integer(I, PLUS(.I, 1));
       ASSIGN_Integer(J, PLUS(.J, 1))
    od;

    return Result;              ! Return by reference

end Concat;

! Translation of instantiations

routine Cat(A, B : desc(Pointer, String_Desc)) =
begin
   return Concat(.A, .B, Character_Desc, String_Desc)
end Cat;

routine Append(A, B : desc(Pointer, Flags_Desc)) =
begin
   return Concat(.A, .B, Boolean_Desc, Flags_Desc)
end Append;
```

6.10. Advantages of PR

The best thing about PR is that it can produce a significantly smaller translation than GE. Chapter 9 presents results that demonstrate this and shows how those programs that should use PR can be detected. Statistics about compilation times in chapter 9 reveal that programs that have smaller translations under PR also compile faster with PR than they do with GE.

As presented in the current chapter, PR requires that all the instantiations of a generic subprogram be known before the subprogram is translated. This was necessary to construct the bodies of primeval subprograms and to construct Union descriptors for variables with generic types. If this requirement

could be relaxed, then it would be possible to compile a generic subprogram prior to its generic instantiations. A generic subprogram could then be compiled once and the code placed in a library, which would save compilation time for programs that instantiate the subprogram. In addition, it would be unnecessary to have the body of a separately compiled generic subprogram available to translate a generic instantiation. This could simplify the compiler and library mechanisms.

It is simple to modify the PR method so that generic subprograms can be compiled prior to their generic instantiations. To perform this modification, the body of a primeval subprogram must be constructed to handle all possible descriptor kinds. This does not lead to the smallest code but ensures that the primeval subprograms will handle all argument types. A variable with a generic type is translated by making the variable a pointer that can reference an object of any type. The object associated with this variable is obtained by applying an extra level of dereferencing. This requires additional addressing code, but makes the translation of a generic subprogram general enough to remove the need to see any of its instantiations.

6.11. Disadvantages of PR

A PR translation usually will execute more slowly than the GE translation would because of the overhead in routine calls imposed by PR. The times required to execute a group of test programs translated by both PR and GE are given in chapter 9. The two translations of each program were executed with identical data, and the results show that for most of the programs the PR translation was, indeed, slower. (For the rest of the programs, the execution times were equal.) The slowdown is not terrible—on the order of 10%—but may be unacceptable in time-critical applications.

For programs with a light use of generic subprograms the PR technique requires more time and the resultant program is larger than with a GE translation. The size differential is not much, but if PR is the only translation method employed, such programs will suffer. These results are also displayed in chapter 9.

Chapter 7
Optimizations

Standard compiler optimization methods (constant folding, global register allocation, lifetime analysis and code motion, for example) perform well on the results of the GE translation of a program. These techniques are especially effective because the subprogram copies produced by GE contain a large amount of precise information; each copy is tailored to (and by) the generic actual parameters appearing in its corresponding instantiation.

Translation by PR, on the other hand, produces code that is littered with routine definitions and invocations. This code is large, due to the proliferation of small routines, and tends to resist improvement by standard optimizations. This problem, however, is overcome by the application of three optimizations: *constant folding*, *routine deletion* and *inline substitution*. The optimizations constant folding and routine deletion, used to improve the effect of inline substitution, are described in sections 7.1 and 7.2. Inline substitution, the primary optimization, is described in section 7.3.

Despite the extra effort required for optimization, the heavy use of subprograms by PR has distinct advantages. Consider the translation of `Concat`, which appears in section 6.9. This translation contains several invocations of `ASSIGN_Integer`. Ultimately, each invocation of the form

```
ASSIGN_Integer(X, .Y);
```

is executed as

```
X ← .Y;
```

The assignment statement is superior to the invocation for several reasons:

- The assignment statement is likely to require less code than the invocation.[43]
- The assignment statement provides the compiler with more flow information than the invocation.
- If all of the invocations of `ASSIGN_Integer` are replaced with assignment statements, then the definition of the routine `ASSIGN_Integer` can be deleted, saving code space.
- The assignment statement will execute faster than the invocation.

It is the source language semantics and the target machine architecture that determined that integer assignment could be implemented by the IL assignment operator. For some target machines, the code to perform assignment of integers could entail a relatively considerable amount of code. For example, on an 8-bit, byte-addressable machine like the Intel 8080 [Intel 78], an integer is likely to occupy two bytes. The IL code for integer assignment on this machine might look like

```
X ← .Y;
(X+1) ← .(Y+1);
```

It is likely that an invocation of a procedure that performs assignment would be smaller (but slower)

---

[43] For example, on the PDP-20 [DEC 78b], the invocation

```
ASSIGN_Integer(X, .Y);
```

generates the code (this is the worst case code assuming X and Y are local variables)

```
MOVEI     AC,X(SP)
PUSH      SP,AC
PUSH      SP,Y(SP)
PUSHJ     SP,ASSIGN_Integer
ADJSP     SP,-2
```

while the assignment statement

```
X ← .Y;
```

generates the code

```
MOVE      AC,Y(SP)
MOVEM     AC,X(SP)
```

than inline code.[44]

The compiler should generate inline code for integer assignment on the PDP-20. But, for a target machine like the 8080, integer assignment should be a routine invocation. The key to the flexibility of PR is that actions are defined by routines and performed by invocations. Optimizations then analyze the code and expand inline the appropriate invocations.

7.1. Constant Folding

Constant folding replaces a program fragment containing information that can be determined at compile time by a simpler, equivalent fragment. Constant folding is applicable to every kind of statement and expression, but it is particularly useful when applied to arithmetic expressions and control statements such as **if** and **case** statements.

For example, the application of constant folding to the following **if** expression

```
if (4+3) > 0 then .A+0 else F() fi
```

yields the equivalent expression

```
if 7 > 0 then .A else F() fi
```

which becomes

```
if True then .A else F() fi
```

---

[44] The machine language required for the assignment statement is something like (this is, once again, worst case code assuming that X and Y are local variables)

```
LXI     HL,X
DAD     SP
MOV     D,M
INX     HL
MOV     E,M
LXI     HL,Y
DAD     SP
MOV     M,D
INX     HL
MOV     M,E
```

while the code for the invocation is (the parameters are passed in register pairs DE and HL)

```
LXI     HL,X
DAD     SP
XCHG
LXI     HL,Y
DAD     SP
CALL    ASSIGN_Integer
```

which requires two bytes fewer than the inline code.

```
.A
```

Inline substitution provides a fertile area for constant folding due to the customized code produced. This makes constant folding an especially important optimization for use in PR translation.

7.2. Routine Deletion

The optimization *routine deletion* deletes the definitions of routines that are not invoked. Such routines exist in the IL graph because the compiler generates all possible actions for a generic type (section 5.4) and some of these actions may not be used. In addition, constant folding may remove the only invocations of a routine. For example, constant folding applied to the **if** expression

```
if (3+4) > 0 then .A+0 else F() fi
```

yields

```
.A
```

and an invocation of F has been deleted.

7.3. Inline Substitution

The technique of *inline substitution* (abbreviated *IS*) is used to rewrite a routine invocation. The invocation is replaced with a specialized copy of the body of the target routine; the task of copying and replacing is known as *inline expansion*. The copy is specialized so that its execution will have the same effect as the invocation. The specialization process is described in section 7.3.2.

IS may reduce the required code space or execution time of a program. An example of a reduction in code space was given in the introduction to this chapter. The inline expansion of the invocation of `ASSIGN_Integer` requires less code than the invocation. Execution time is almost always improved by the use of IS, since the replacement results in the elimination of a subprogram invocation.[45] This savings can be significant because subprogram invocation is a relatively expensive machine operation.

This dissertation is concerned with the use of IS to decrease the code space required by a PR translation. As discussed in the next section, the use of IS to yield absolutely minimal code space

---

[45] On systems with demand paging, execution time can be increased if the substitution results in an increase in program size that leads to anomalous page swapping behavior.

seems to be impractical for use in a compiler. IS, therefore, depends on heuristics; hope of obtaining minimal code size must be abandoned.

A simple approach to IS was used in the experimental compiler. As will be seen in sections 9.3 and 9.8 the technique produced good results and was considered adequate. The actual method used to perform IS is unimportant to the success of PR. If a better IS technique is employed, the PR method will benefit accordingly.

The IS technique used in this work is described in the remainder of this chapter. The reader should consider this description an indication of the complexity of optimization that is sufficient to make PR attractive. Details about the implementation of IS in the experimental compiler can be found in section 8.2.3.

7.3.1. Determining the Invocations to Expand

To perform IS, one must determine the invocations to expand. The replacement of an invocation may produce new invocations for consideration. For example, inline expansion of the invocation of `Digit`:

```
routine Digit(C : Character_Desc) =
begin
   return GREATER_THAN_OR_EQUAL(.C, '0') and
          LESS_THAN_OR_EQUAL(.C, '9')
end Digit;

if Digit(.CurChar)
   then . . .
```

yields

```
routine Digit(C : Character_Desc) =
begin
   return GREATER_THAN_OR_EQUAL(.C, '0') and
          LESS_THAN_OR_EQUAL(.C, '9')
end Digit;

if (GREATER_THAN_OR_EQUAL(.Curchar, '0') and
    LESS_THAN_OR_EQUAL(.Curchar, '9'))
    then . . .
```

This contains two new invocations: one of `GREATER_THAN_OR_EQUAL` and one of `LESS_THAN_OR_EQUAL`. Both of these invocations should be considered for inline expansion.

A useful method for determining the invocations to expand inline is a research problem. Chapter

2 describes two studies in this area. Scheifler's work (section 2.3.1) deals with the use of IS to decrease expected execution time. Ball (section 2.3.2) investigated IS for time and space reduction. To handle the new invocations created by the application of inline expansion, both Scheifler and Ball employ the following scheme, which is also used in this dissertation:

1. Determine an invocation to expand inline. If there is no such invocation, terminate.
2. Inline expand the invocation (section 7.3.2).
3. Perform optimizations on the expansion (in this dissertation constant folding and routine deletion are used).
4. Repeat step 1, including for consideration any invocations created in step 2.

One might hope to implement step 1 by the following greedy method:[46]

> Choose that invocation whose inline expansion leads to the greatest reduction in space. If there is no such invocation, terminate.

Unfortunately, this procedure is not guaranteed to find all desirable inline expansions. There are cases where there is no invocation whose inline expansion reduces code space but inline expansion of several invocations leads to a reduction. For example, in the following program fragment:

```
routine R(N : Integer_Desc) =
begin
   if .N = 0 then
      Q(X, Y, Z)
   else
      a lot of code
   fi
end R;

R(0);
R(0);
```

the inline expansion of either call of R would result in the replacement of the invocation by

```
Q(X, Y, Z);
```

which requires more code than the original invocation:

```
R(0);
```

Thus, neither invocation is worth expanding inline by itself. But if both invocations are expanded inline, the definition of routine R can be deleted, saving a lot of code space.

So it is not possible to choose an invocation to expand without considering all invocations. It is this difficulty that makes the performance of IS for minimum program size infeasible for practical use. A

---

[46] For a description of the greedy method of algorithm design see chapter 4 of the book by Horowitz and Sahni [Horowitz 78].

formal proof of the intractability of IS, which is shown to be NP-hard, is given in appendix A.[47]

Despite the possibility of failure in considering an invocation independently, this method is used with a minor modification in the experimental compiler, as follows:

> Choose an invocation whose inline expansion leads to a reduction in space. If there is no such invocation, terminate.

There seemed to be no reason to search for the "best" invocation since all invocations whose expansions will immediately decrease the size of the program will eventually be found. Thus, any invocation whose inline expansion leads to a reduction in space is accepted.

To choose an invocation whose inline expansion effects a reduction in space requires determining the final code size for the expansion of the invocation. But this is the real stumbling block in performing IS because other optimizations are applied to the expansion and these optimizations may affect the final size drastically. As an example, consider

```
routine Checked_Assign(Dest : desc(Pointer, Integer_Desc),
                       Source, L, U : Integer_Desc) =

begin
    if (.Source < .L) or (.Source > .U) then
        Write_String("Assignment out of range: ");
        Write_Int(.Source);
        Error_Handler(Assign_Error)
    fi;
    .Dest ← .Source
end Checked_Assign;

Checked_Assign(Index, 1, 0, 1023);
```

When expanded inline and constant folded, the invocation of `Checked_Assign` becomes

```
Index ← 1;
```

The code size could be determined precisely by expanding the invocation, performing the auxiliary optimizations and then estimating the size of the resultant code. This may be prohibitively expensive to perform.

What is desired is an efficient method for predicting the results of an inline expansion. The method need not be a good predictor of actual code size since a good relative estimate would reveal that the size of the invocation of `Checked_Assign` above is larger than its ultimate expansion. An elegant method, investigated by Ball, is described in section 2.3.2. Ball's method attempts to take into account the effects of constant folding.

---

[47] Scheifler has shown [Scheifler 77] that using IS to minimize expected execution time is also NP-hard.

This research takes a much simpler approach to estimating the size of an inline expansion (the method is described in detail in section 8.2.4). The compiler guesses at the final size of an inline expansion by inspecting the invocation and the body of the target routine. The complexity of each actual parameter is taken into account since simple expressions—constants and variables, for example—will likely lend themselves to further optimizations in the expansion.

The size estimation for most statements and expressions is done by table lookup, but special provisions are made for the structures that appear in compiler-generated routines. For example, the compiler simulates the effect of constant folding on a **case** statement that occurs as the body of a routine because this is the form that the primitive routines take (section 6.2). As a general purpose expansion estimator, this method might leave much to be desired. For use in IS, however, this technique yields good results. The technique is judged adequate because IS, which depends on the method, yielded good results, as shown in section 9.8.

7.3.2. Inline Expansion: The Basic Schema

Inline expansion replaces an invocation by a copy of the target routine. Although inline expansion is not difficult to perform, it must be done carefully to avoid code space overhead that could destroy the benefits of IS. The remainder of this section presents a detailed discussion of inline expansion. A discussion of some implementation details of inline expansion in the experimental compiler is presented in section 8.2.3.

Inline expansion in the experimental compiler is performed on the internal IL representation of a program. The remainder of this section will illustrate the technique with examples expressed in the external representation of IL. But the reader should remember that the indicated transformations are performed on graph structures. When necessary, the compiler creates new symbol table nodes during the inline expansion to avoid naming conflicts.

To perform inline expansion it is not sufficient to copy the body of the target routine substituting the actual parameters for occurrences of formal parameters. Even ignoring name conflicts (taken care of by creation of new symbol table entries), the possibility of side effects prevents this simple approach. If a formal parameter is referenced more than once, the actual will be evaluated multiple times. This will lead to an incorrect expansion if the the corresponding actual parameter has side effects.

To handle this problem, the compiler uses a general schema that yields a worst case translation that avoids all possible problems. After translation, optimizations are used to reduce the size of the expansion.

Consider the following template for an IL routine named R:

```
routine R(f1, f2, ..., fF) =
begin
    l1; l2; ...; lL;
    s1; s2; ...; sS
end R;
```

where f_i, $0 \le i \le F$, is a formal parameter; l_i, $0 \le i \le L$, is a local declaration; s_i, $0 \le i \le S$, is a statement. Given an invocation of R:

```
R(a1, a2, ..., aF)
```

where a_i is an actual parameter, the inline expansion of the invocation is illustrated in figure 7-1. In this schema, the f_i are constant names, and l_i and s_i are modified copies of l_i and s_i. These copies are modified by performing the substitutions indicated in table 7-1. These rules must be applied in the order displayed. For example, if an occurrence of f_i is preceded by the IL dereferencing operator, it is handled by the the first rule. Only an instance of f_i not preceded by the dereferencing operator is transformed according to the second rule.

```
begin
    constant f1 = a1;
    constant f2 = a2;
    . . .
    constant fF = aF;
    label: begin
        l1; l2; ...; lL;
        s1; s2; ...; sS
    end label
end
```

Figure 7-1: Inline Expansion Schema

Original	Replacement
.f_i	f_i
f_i	ref(f_i)
return	leave *label*
return *e*	leave *label* with *e*

Table 7-1: Substitutions for Inline Expansion Schema

The schema produces two nested blocks. The outer block is used to introduce auxiliary constant

declarations that hold the results of the evaluations of actual parameters. The inner block is necessary to allow for local declarations within the routine body. These declarations must have access to the formal parameters (now replaced by constants), and the inner block provides the proper scope.

Within the body of the routine, the value of the formal parameter f_i is obtained by dereferencing, $.f_i$. The constant declarations in the expansion bind the value of each actual parameter to a new constant. The value of a constant is obtained without dereferencing (section 1.4.3), so the value of formal parameter f_i is obtained by the expression f_i. This substitution is indicated in the first line of table 7-1. Likewise, a reference to a formal parameter, indicated by f_i in the routine, is obtained by creating a reference to the corresponding constant, as indicated in line two of the table.

Return statements in the routine are replaced by exits from the inner block, which is labeled for this purpose. If the **return** statement yields a value, e, the corresponding **leave** statement yields the equivalent transformed value, e. Lines three and four of the table show these substitutions.

7.3.3. Optimizations to the Schema

Because the schema for inline expansion is general, it sometimes produces poor code. For example, the inline expansion of the invocation of `Abs`, below:

```
routine Abs(N) =
begin
   if .N ≥ 0 then .N else -.N fi
end Abs;

Base ← Abs(-10);
```

by the general schema is

```
Base ←
   begin
      constant N1 = -10;
      Abs1: begin
         if N1 ≥ 0 then N1 else -N1 fi
      end Abs1
   end;
```

But one would really like the code to be

```
Base ← 10;
```

If the code generator performs constant propagation, then it is possible that this code would result eventually. It is advantageous, however, for the compiler to have this information earlier than code generation. This would be important especially if the inline expansion of `Abs` were involved in further optimizations.

To improve the results of inline expansion, several optimizations are performed on the basic schema. The optimizations are constant folding (section 7.1), routine deletion (section 7.2) and the deletion of unnecessary constant declarations, which is described here.

The constant declarations serve two purposes: to prevent reevaluation of expressions with side effects and to avoid multiple evaluation of expressions that are expensive to compute. If a formal parameter is referenced once (or not at all) within the expansion, then no constant declaration is necessary for the corresponding actual parameter.[48]

A constant declaration can be avoided also for an actual parameter that is a *simple expression*. The definition of a simple expression will vary for different implementations and target machines, but such an expression must have no side effects and be inexpensive to evaluate. Within the experimental compiler, simple expressions are defined as follows:

- integer, string and character literals (`45`, `"Angry Samoans"`, `'@'`);
- variable, constant, descriptor and routine references (`A`, `Base`, `desc(Integer)`, `Exchange`);
- dereferences of simple expressions (`.A`, `..A`);
- upreferences of constant expressions (`ref(0)`, `ref("Killing Joke")`).

Flow information is gained by placing these expressions directly inline, instead of in constant declarations. The evaluation of these expressions is cheap enough to make the transformation worthwhile.

These optimizations transform the inline expansion of `Abs` (page 100) into

```
Base ←
   begin
      Abs1: begin
         if 10 ≥ 0 then 10 else -10 fi
      end Abs1
   end;
```

Constant folding applied to this expansion folds the **if** statement and removes the blocks, since they serve no purpose, to obtain

```
Base ← 10;
```

---

[48] This optimization is applicable only if the semantics of subprogram invocation does not specify the order of evaluation of actual parameters with side effects. In addition, the semantics must not guarantee that all actual parameters will be evaluated. The source language, of course, obeys these restrictions. Performing this optimization for a language that is more rigid with regard to actual parameter evaluation requires careful analysis.

The three optimizations—constant folding, routine deletion and deletion of unnecessary constant declarations—are not independent. The performance of one may present further possibilities for another optimization. To obtain the maximum benefit, these optimizations are applied in a loop until there is no change. This is illustrated in figure 7-2, which is a schema of the optimization loop used in the experimental compiler.

```
routine Optimize_IE(Tree) =
begin
   local Change : Boolean;

   Change ← True;
   while Change do
      Change ← False;
      !
      ! The three optimizations below will set
      ! Change to True if any modifications are done.
      !
      Constant_Fold(Tree, Change);
      Routine_Deletion(Tree, Change);
      Delete_Constant_Declarations(Tree, Change)
   od
end Optimize_IE;
```

Figure 7-2: Inline Expansion Loop

7.4. Improvement from Optimizations: An Example

As an example of the utility of the optimizations discussed in this chapter, the result of optimizing the translation of `Concat` (section 6.9) is displayed in section 7.4.1.

Before displaying the result of the optimizations, it is instructive to follow the optimizations applied to a particular invocation:

```
LAST_Arr(.A)
```

The order in which invocations are considered for expansion during IS is undetermined; to demonstrate the optimizations this example will assume an order.[49]

When the invocation of `LAST_Arr` is considered for expansion, the size estimator decides that the expansion is not worthwhile. This is because the expansion would be the invocation

[49] The results of IS are order-independent. The number of invocations considered, however, is usually order-dependent, and choosing an appropriate order may reduce the time required for IS.

```
LAST(.A, Integer_Desc, .Elt, .Arr)
```

that has three more actual parameters than the original invocation.

The invocation within the body of LAST_Arr is now considered. In this case, the expansion is deemed desirable, and the body of LAST_Arr becomes

```
return
   begin
      constant A1 = .A;
      constant Index1 = Integer_Desc;
      constant Element1 = .Elt;
      constant Arr1 = .Arr;
      LAST:begin
         case kind(Arr1) of
            when Array => leave LAST with .(A1+UB_Offset)
         esac
      end LAST
   end
```

Constant folding is applied to this expansion. Since the body of the **case** statement contains a single arm, the **case** statement can be eliminated to yield

```
return
   begin
      constant A1 = .A;
      constant Index1 = Integer_Desc;
      constant Element1 = .Elt;
      constant Arr1 = .Arr;
      LAST:begin
         leave LAST with .(A1+UB_Offset)
      end LAST
   end
```

Constant folding then removes the inner block and the **leave** to yield

```
return
   begin
      constant A1 = .A;
      constant Index1 = Integer_Desc;
      constant Element1 = .Elt;
      constant Arr1 = .Arr;
      .(A1+UB_Offset)
   end
```

Routine deletion is applied and has no effect since the expansion contains no routines.

All of the constant declarations are unnecessary: `A1` because it is only referenced once; `Index1`, `Element1` and `Arr1` because they are not referenced. The expansion is now rewritten:

```
return .(.A+UB_Offset)
```

Eventually the invocation of LAST_Arr, the invocation originally under consideration, is reconsidered. Since the body of LAST_Arr has changed, the expansion is judged worthwhile:

```
begin
   constant A1 = .A;
   LAST_Arr:begin
      leave LAST_Arr with .(A1+UB_Offset)
   end LAST_Arr
end
```

This expansion is transformed by the optimizations to become

```
.(.A+UB_Offset)
```

In the results of optimizing Concat, displayed below, all but one of the routines that implement actions have been deleted because all of their invocations have been expanded inline. The routine ASSIGN_Elt remains because the compiler determines that it will be unable to perform constant folding on the case that composes the body of the routine. The value of Elt is unknown because it is a formal parameter of the routine Concat, which surrounds ASSIGN_Elt.

7.4.1. The Optimization of Concat

```
!*****************************************!
! Optimization of translation of Concat   !
!              (page 86)                  !
!*****************************************!

constant Integer_Desc = desc(Integer);
constant Boolean_Desc = desc(Boolean);
constant Character_Desc = desc(Enumeration, 128);
constant String_Desc = desc(Array,
                        Integer_Desc,
                        Character_Desc);
constant Desc_Desc = desc(Descriptor);

constant Flags_Desc = desc(Array, Integer_Desc,
                           Boolean_Desc);

constant Elt_Desc = desc(Union, Boolean_Desc,
                         Character_Desc));
constant Ref_Elt_Desc = desc(Pointer, Elt_Desc);
constant Arr_Desc = desc(Pointer, desc(Union,
                                 String_Desc,
                                 Flags_Desc));
```

```
routine Concat(A, B : Arr_Desc, Elt, Arr : Desc_Desc) =
begin
   routine ASSIGN_Elt(Dest, Source : Ref_Elt_Desc) =
   begin
      case kind(.Elt) of
         when Integer =>
            .Dest ← ..Source
         when Boolean =>
            .Dest ← ..Source
         when Enumeration =>
            .Dest ← ..Source

      esac
   end ASSIGN_Elt;

   local I, J : Integer_Desc;
   local Result : desc(Array, Integer_Desc,
                              .Elt,
                              1,
                              .(.A+UB_Offset)+.(.B+UB_Offset));


   I ← 1;
   while .I ≤ .(.A+UB_Offset) do
      ASSIGN_Elt(subscript(Result, .I), subscript(.A, .I));
      I ← .I + 1
   od;

   J ← .I;
   while .J ≤ .(.B+UB_Offset) do
      ASSIGN_Elt(subscript(Result, .I), subscript(.B, .J));
      I ← .I + 1;
      J ← .J + 1
   od;

   return Result

end Concat;

! Instantiations

routine Cat(A, B : desc(Pointer, String_Desc)) =
begin
   return Concat(.A, .B, Character_Desc, String_Desc)
end Cat;

routine Append(A, B : desc(Pointer, Flags_Desc)) =
begin
   return Concat(.A, .B, Boolean_Desc, Flags_Desc)
end Append;
```

Chapter 8
The Experimental Compiler

To test the efficacy of the PR translation method an experimental compiler was constructed that implements the techniques described in the preceding four chapters. To give the reader a feel for the complexity of a system that generates space-efficient code for generic subprograms this chapter provides some details about the implementation of the compiler.

The compiler is composed of three parts: the *Front End* does syntactic processing, the *Translator* translates the parsed source language program into the BLISS-36 language [DEC 78a], and the *Back End* produces PDP-20 machine language [DEC 78b] from the BLISS-36 representation. These compiler parts are illustrated in figure 8-1.

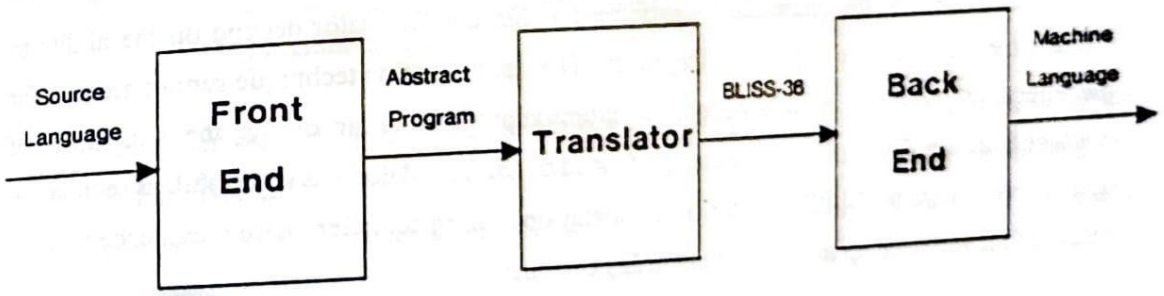

Figure 8-1: The Parts of the Experimental Compiler

In this figure a part is represented by a box. The lines emanating from a part denote its input, to the left, and its output, to the right. The inputs and outputs are representations of the program expressed in some language. The input and output languages for each part are listed above the lines. For example, the input to the Translator is represented in the language Abstract Program and the output is represented in the language BLISS-36.

Each part is made up of a number of autonomous *phases*. The phases, each of which performs a specific task, are executed sequentially. In the experimental compiler the input and output from each phase are placed in text files. Each phase reads its input from a file and converts it into an internal representation on which to perform its task. The phase converts the internal data structure to an external text format before writing its output.

The compiler's phase structure was inspired by the BLISS-11 compiler [Wulf 75]. The separation into phases allows a clean division of compilation tasks, which simplifies design, implementation and debugging. The PQCC project (section 2.2.1) coupled this compiler structure with textual representations of phase outputs [Newcomer 79], as is done in the experimental compiler. Compared to a binary representation the textual representation is easy to read and to write and this aids in debugging. This representation also makes it easy to construct input by hand, which allows the phases to be implemented independently.

The production of a BLISS-36 program as the output of the Translator simplified the construction of the experimental compiler by allowing the generation of fairly high-level code and avoidance of the complexities of PDP-20 relocatable code.

However, the use of BLISS-36 is important for more than convenience. To validate the utility of the translation techniques it is important that the final code generation phase perform many optimizations. This is because the optimizations performed within the Translator depend on the ability to estimate the final code size of program fragments. The size estimation technique cannot account for the effects of low-level, machine-dependent optimizations that occur during the generation of machine language. The inline substitution performed by the Translator was successful, as seen in the next chapter. Since the BLISS-36 compiler is a highly optimizing compiler, there is evidence that the results of PR will stand up in a production-quality compiler.

The Front End and Back End, though largely uninteresting, were required to get an appropriate form of the input program to the Translator and to produce machine language from the Translator's output. The Front End is extremely simple, although it does perform some specialized operations (section 8.1). The Back End is composed of standard PDP-20 programs (section 8.3).

The interesting sections of the experimental compiler lie within the Translator. The Translator operates in a straightforward manner: The program is translated into an intermediate representation (represented in IL), which is then optimized. The translation can be performed by the GE or PR method as requested by the programmer at compile time. GE translation uses standard techniques

and so requires no specialized operations other than the copying of generic subprograms, which is easy (section 8.2.2.1). A program is translated by PR as if it were composed of a large number of subprogram invocations (chapter 5). This necessitates an approach to translation that is different than that of GE, and requires a number of specialized operations. Fortunately, the techniques used by PR integrate nicely with the standard methods of GE.

The primary optimization used within the Translator to improve the results of PR is inline substitution (section 7.3). This optimization, which is essential to the success of PR, can be performed in a phase prior to the standard optimizations in a compiler. This is done in the experimental compiler. The standard optimizations are handled by the BLISS-36 compiler. It seems likely that inline substitution could be integrated with the other optimizations of a compiler; this would eliminate the need for a separate phase. But the proper means for merging these optimizations is a research topic.

8.1. The Front End

The Front End is made up of two phases, shown in figure 8-2. The functions of each phase are as follows:

FE	Parse source program and generate parse tree.
COMPILE	Read parse tree and convert to internal format. Perform semantic analysis of parse tree and produce a program in Abstract Program language.

The phases composing the Front End are not very interesting (although they are crucial), so for the remainder of this chapter the term *compiler* will refer exclusively to the phases comprising the Translator (section 8.2). For completeness, the next two sections provide descriptions of the FE and COMPILE phases.

8.1.1. The FE Phase

The FE phase was produced by the FEG parser generator [Nestor 82]. The functions of FE are to parse the source language program and produce a parse tree. The parse tree is written to a text file in a notation developed by the PQCC project [Newcomer 79]. Some simplifications of the parse tree are performed by FE, which, for example, compresses long chains of reductions like

```
<expr> ⇒ <factor> ⇒ <term> ⇒ <identifier>
```

to the single reduction

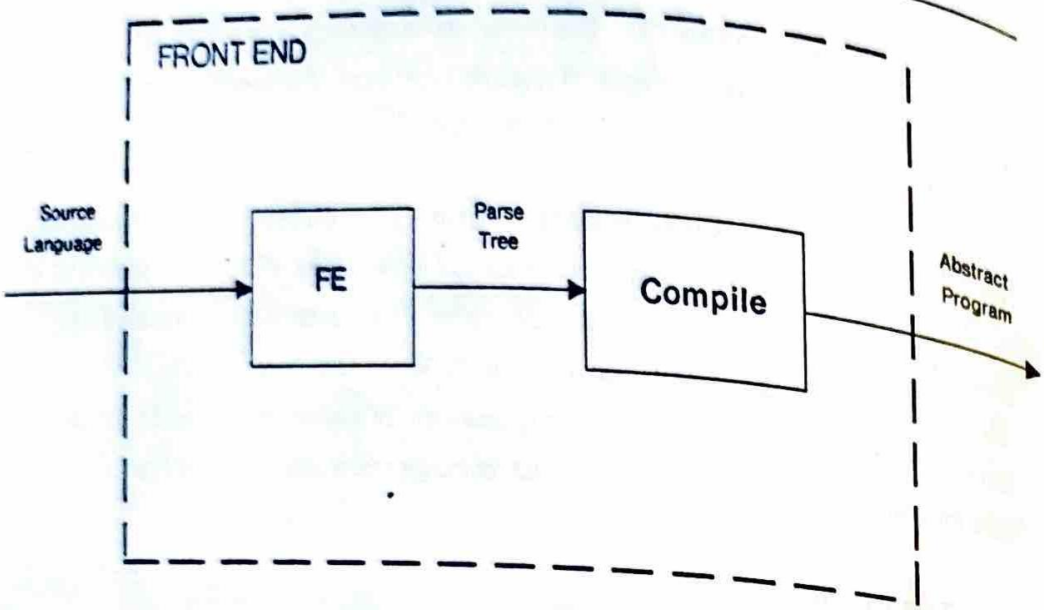

Figure 8-2: The Front End

```
<expr> = <identifier>
```

These transformations are not crucial, but are convenient and save the next phase, COMPILE, some work.

8.1.2. The COMPILE Phase

The COMPILE phase reads the output of FE and converts it to an internal format. This internal form is then analyzed in a recursive descent fashion and an Abstract Program graph is produced.

The Abstract Program graph is a representation of the source language program annotated with semantic information. The COMPILE phase produces this information by performing the following transformations:[50]

- *Name identification and symbol table construction:* Replace identifiers in the source program with references to symbol table entries.
- *Produce explicit specifications for generic instantiations:* Attach a specification to each generic instantiation to enable overload resolution. For example, the generic instantiation following (function `Concat` is defined in figure 5-1):

[50] Although some of these actions are unique to the translation of generic subprograms, none of the actions are difficult enough to warrant discussion.

```
function Cat is new Concat(Character, String);
```

is annotated with the specification

```
function Cat(A, B : in String);
```

- *Overload resolution:* Resolve the targets of subprogram invocations.
- *Produce a canonical form for types:* Transform types to a standard form and create symbol table entries for the types.
- *Produce a canonical form for expressions using operators:* Convert these expressions to subprogram invocations.

A compilation unit (section 3.2.5) may refer to identifiers in other compilation units that have been run through COMPILE. The COMPILE phase obtains the necessary symbol table information for external units by reading the output that COMPILE produced for these units.

To handle cross-unit references, the COMPILE phase generates a unique name for each symbol table entry. The unique names allow unambiguous references across compilation units. These references are resolved during the INFO phase, which is described in section 8.2.1.

8.2. The Translator

The phases of the Translator, illustrated in figure 8-3, are written in the MACLISP [Moon 74, Touretzky 82] dialect of LISP. A brief description of the job of the phases of the Translator follows:

INFO	Read output of COMPILE from all separately compiled units. Resolve units into a single program graph. Annotate program with information if PR is being used. Generate bodies of primitive operations if PR is being used.
TRANSLATE	Read output of INFO and translate to IL.
OPT	Read output of TRANSLATE and optimize it. Produce optimized IL program.
CODE	Read output of OPT and generate a BLISS-36 program.

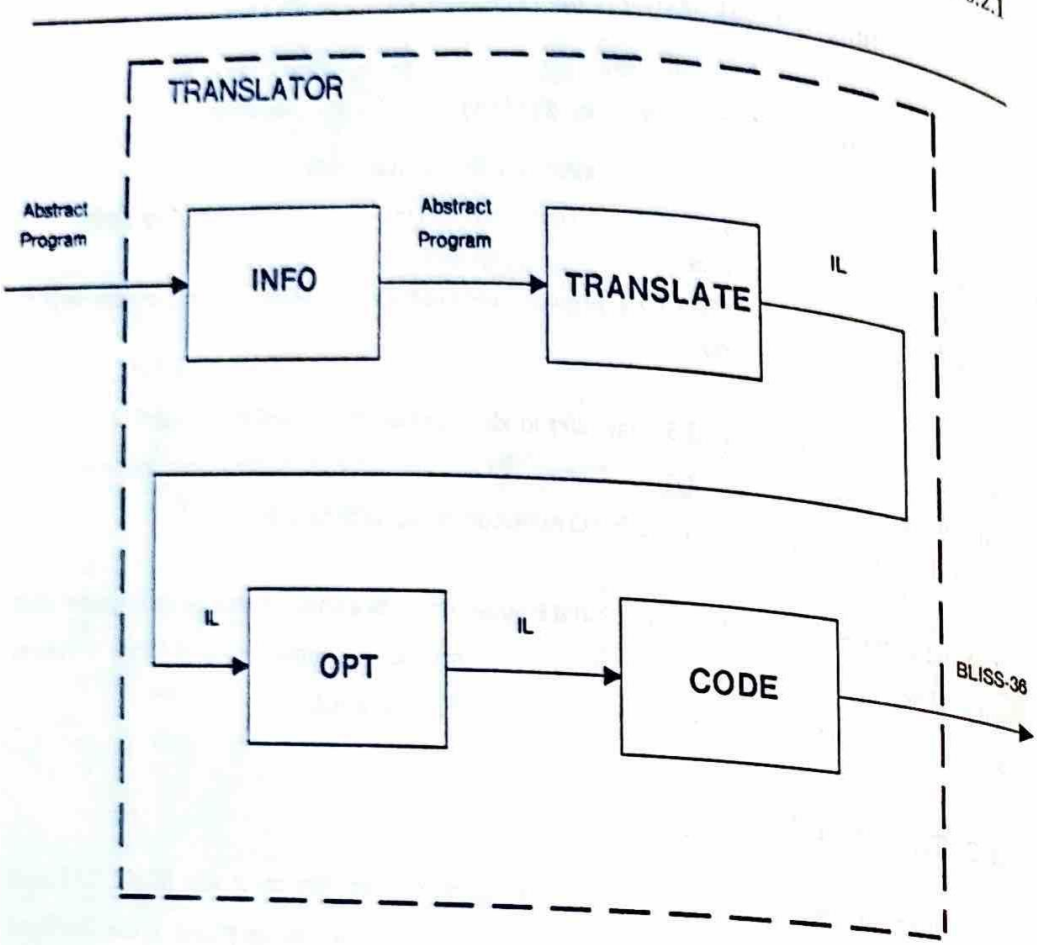

Figure 8-3: The Translator

8.2.1. The INFO Phase

The first function of the INFO phase is to collect the outputs of COMPILE into a single program graph by resolving cross unit references. Cross unit references are resolved automatically by the LISP system because unique atom names are used for each symbol table entry. Each atom name is unique in LISP, so all references to a particular name represent the same object.[51]

If the GE technique is being used, INFO's job is done. For PR translation, INFO must perform two additional tasks: annotate the program graph with information about the use of generic formal type parameters and create the bodies of the primeval subprograms, which define the primitive actions (section 5.3).

[51] In a compiler written in a standard algebraic language, cross unit references might be resolved by performing a pass over the program graphs and building a cross reference table.

To translate variable declarations that involve a generic type, and to construct the primitive operations, TRANSLATE must know all the non-generic types to which a generic type might be bound. The INFO phase performs a traversal of the complete program graph to gather this information. The program must be walked carefully to insure that instantiations of a generic subprogram are analyzed before the subprogram. To see this, consider the program fragment

```
generic
   type S is private;
procedure P();

generic
   type T is private;
procedure Q();

procedure Q is
   procedure New_P is new P(T);
begin
   . . .
end Q;

procedure New_Q is new Q(Character);
procedure Also_New_P is new P(Boolean);
```

The generic formal type parameter T is passed as a generic actual type parameter in the generic instantiation of New_P. Thus, all of the bindings for T must be propagated to S, the generic formal type parameter of procedure P. In this case, S may be bound to Boolean (directly from the generic instantiation of Also_New_P) and Character (indirectly from the generic instantiations of New_Q and New_P).

The following graph traversal is used to obtain the desired order:

- The body of a subprogram is traversed before the subprogram's local declarations. This is necessary because the body may contain nested blocks that have generic instantiations of generic subprograms defined in the local declarations. Processing the body first guarantees that the instantiations are found before their targets.
- Declarations are traversed in reverse textual order, since a generic instantiation may occur after its target generic subprogram in a list of declarations. Reverse order guarantees that instantiations are visited before their targets.

Once the bindings of generic type parameters have been determined, INFO constructs the bodies of the primeval subprograms. The bindings of the generic formal type parameters for a primeval subprogram determine the set of types over which the subprogram must operate.

For example, the primeval subprogram LESS_THAN has the specification

```
generic
   type T is (<>);
function LESS_THAN(X, Y : in T) return Boolean;
```

If the generic type parameter T is bound to the set of types {Integer, Boolean, Character}, then the body of LESS_THAN must operate on parameters of one of these types. Recall (section 6.2) that the body of this routine will be

```
routine LESS_THAN(X, Y : descriptor, T : desc(Descriptor)) =
begin
   case kind(.T) of
      when Integer =>
         code for integer compare
      when Boolean =>
         code for boolean compare
      when Enumeration =>
         code for enumeration compare
   esac
end LESS_THAN;
```

The code for each arm of the **case** statement is obtained from a file that contains IL routines to perform each action on each descriptor kind. The file contains, for example, the following routine for assignment of objects with integer descriptors:

```
routine LESS_THAN_Integer(X, Y : Integer_Desc) =
begin
   return .X < .Y
end LESS_THAN_Integer;
```

The compiler places an invocation of each routine in the appropriate arm of the **case** statement. The complete definition of LESS_THAN is, therefore

```
routine LESS_THAN(X, Y : descriptor, T : desc(Descriptor)) =
begin
   case kind(.T) of
      when Integer =>
         LESS_THAN_Integer(.X, .Y)
      when Boolean =>
         LESS_THAN_Boolean(.X, .Y)
      when Enumeration =>
         LESS_THAN_Enumeration(.X, .Y)
   esac
end LESS_THAN;
```

(It is left to the optimization phase to expand inline the invocations to the predefined routines.) As can be seen in the body of LESS_THAN above, the construction of primeval routines involves binding conversions. These conversions are necessary because of the interface points that occur in the bodies of primeval subprograms (section 6.4.1.2).

Section 8.2.2

8.2.2. The TRANSLATE Phase

The real work begins with TRANSLATE, which makes a pass over the program graph output by INFO and translates it to an equivalent program expressed in IL. The translation of code that is not within a generic subprogram, and is not a generic instantiation, is straightforward. But translation of generic subprograms and instantiations depends on the translation method being used.

8.2.2.1. GE

If GE is being used, there is nothing to do for a generic subprogram because the work is done at the generic instantiations. A generic instantiation is processed in two steps. First the generic actual parameters in the generic instantiation and the generic formal parameters in the generic subprogram are analyzed and a list is formed that indicates the replacements to be made. The copying of the generic subprogram then begins, using this list to direct substitutions.

As an example, consider the generic subprogram and generic instantiation following:

```
generic
   type T is private;
procedure Exchange(X, Y : in out T);

procedure Exchange(X, Y : in out T) is
   Temp : T;
begin
   Temp := X;
   X := Y;
   Y := Temp;
end Exchange;

procedure Int_Swap is new Exchange(Integer);
```

In expanding `Int_Swap`, the replacement list constructed is

```
T      → Integer
```

During the copying process additional entries may be made in the replacement list. These additions are necessary because the copy being constructed may contain new symbol table nodes, and references internal to the copy must be adjusted accordingly. For example, when `Exchange` is being copied a new node for the variable `Temp` is created; the old node can not be used because its type is wrong. The creation of the new node causes the replacement list to be amended as

```
T      → Integer
Temp   → Temp
```

(The notation `Temp` denotes the new node created as the copy of `Temp`. This copy has the type

`Integer`, which replaced the generic type T.) As copying proceeds, references to `Temp` will be replaced by references to `Temp`. Finally, the subprogram copy is translated like a non-generic subprogram.

8.2.2.2. PR

If PR translation is being performed, then a generic subprogram is translated, not thrown away as in GE. The TRANSLATE phase is designed so that the translation of a generic subprogram is much like the translation of a non-generic subprogram. The compiler need only detect references to generic formal parameters and translate them as described in section 6.2.

A generic instantiation is especially simple to translate. The body of a predefined generic instantiation has already been constructed by the INFO phase. A user-defined generic instantiation is translated to a routine, as described in section 6.3. The only tricky thing is to handle the interface points appropriately (section 6.4.1.1).

8.2.3. The OPT Phase

The OPT phase is the workhorse of the compiler. The purpose of this phase is to reduce the size of the program as much as possible. OPT does this primarily by using inline substitution, described in section 7.3, and the auxiliary optimizations of constant folding (section 7.1) and routine deletion (section 7.2). As described in section 7.3.3, these three optimizations are not independent: the performance of one frequently presents further opportunities for the application of another optimization. To capture all possible improvements, the optimizations are applied in a loop until there are no further changes. This technique is illustrated in figure 8-4, which is a schema of the optimization loop used in the OPT phase.

Each of the IS and routine deletion optimizations tends to produce transformations that allow the optimization to be reapplied fruitfully. To capture these effects, IS and routine deletion are applied in loops similar to the primary optimization loop (figure 8-4). Schemas for the IS and routine deletion operations are displayed in figures 8-5 and 8-6. There is no need to apply constant folding in a loop because it is performed bottom-up on a subtree, which ensures that all possible transformations are done.

```
routine Optimize(Root) =
begin
   local Change : Boolean;
   local Invocations, Routines : Set;

   Invocations ← {all invocations};
   Routines ← {all routines};

   Change ← True;
   while Change do
      Change ← False;
      !
      ! The three routines below will set Change
      ! to True if any modifications are done.
      !
      Constant_Fold(Root, Change);
      IS(Invocations, Change);
      Routine_Deletion(Routines, Change)
   od
end Optimize;
```

Figure 8-4: Primary Optimization Loop

```
routine IS(Invocations, Change) =
begin
   local New : Set;

   Change ← True;
   while Change do
      Change ← False;
      New ← Invocations;
      Invocations ← {};
      for I in New do
         if I should be expanded then
            IE(I, Invocations);          ! Inline expand I, add new
                                         ! invocations to Invocations.
                                         ! Figure 7-2
            Optimize_IE(I);
            Change ← True
         else
            Invocations ← Invocations ∪ {I}
         fi
      od
   od
end IS;
```

Figure 8-5: Inline Substitution Loop

```
routine Routine_Deletion(Routines, Change) =
begin
   local New : Set;

   Change ← True;
   while Change do
      Change ← False;
      New ← Routines;
      Routines ← {};
      for R in New do
         if R is not called then
            Delete R & any routines it contains;
            Change ← True
         else
            Routines ← Routines ∪ {R}
         fi
      od
   od
end Routine_Deletion;
```

Figure 8-6: Routine Deletion Loop

8.2.4. Determining the Invocations to Expand

Within the inner loop of the inline substitution routine it is necessary to determine whether each invocation should be expanded. This is shown in figure 8-5 by the test

```
if I should be expanded
```

An invocation should be expanded inline if its expansion requires less code than the invocation does. As discussed in section 7.3.1, exact determination is impractical and heuristics must be used to make this decision.

The size of a subgraph is estimated by traversing it and summing the estimated sizes of its nodes. The estimated size for a node that occurs as an expression or a statement is found by table lookup. The size of a node that occurs as a declaration is determined by special routines that obtain a better estimate than table lookup could provide. The size estimation routine is outlined in figure 8-7. The size table used for expressions and statements in the experimental compiler is given in appendix D.

The size of a program fragment is estimated fairly accurately by this method. But more work is done to estimate the final size of an inline expansion in an attempt to capture the effects of the optimizations that will be performed on the expansion. The OPT phase analyzes the actual parameters in the

```
routine Size(Subtree) =
begin
    local Sum : Integer;

    Sum ←
        (if Subtree is a declaration then
            special routine(Subtree)
        else
            Size_Table[kind of Subtree]
        fi);
    for Node in Sons(Subtree) do
        Sum ← Sum + Size(Node)
    od;
    return Sum
end Size;
```

Figure 8-7: Size Estimation Routine

invocation and scans the body of the target routine to determine which actual parameters will require auxiliary constant declarations (section 7.3.2). This information is taken into account in estimating the final size of the expansion, since it takes additional code to produce a constant declaration.

The estimated sizes for an invocation and its inline expansion are then compared to decide whether to expand the invocation. The sizes could be compared directly and the expansion predicate defined as

expand I ≙ (*size of expansion*) ≤ (*size of invocation*)

(If the sizes are equal, then expansion is chosen since it will improve execution time.) But the size estimates are rough and are likely to be biased against performing the expansion. This is because the estimate can not take flow analysis into account and so will usually overestimate the size of an expansion. The comparison is, therefore, altered in favor of performing an expansion: If the sizes are close, expansion will be chosen since it lowers expected execution time and later optimizations may improve its size. The value of the variable `Slop` in the experimental compiler is used to prejudice the comparison as follows:

expand I ≙ (*size of expansion*) ≤ (*size of invocation*) + `Slop`

The value of `Slop` in the experimental compiler is 1. This value, which was determined by experimentation, is the least value of `Slop` that yielded good results. For values less than 1, many small routines were left unexpanded. The size estimation performs well, however, for $1 \leq$ `Slop` ≤ 3. Thus, `Slop` is not very sensitive for values greater than 1. If `Slop` is made too large, of course, the bias towards performing expansion will have a detrimental effect on program size.

8.2.5. The CODE Phase

The CODE phase, the simplest phase of the Translator, reads the output of OPT and writes a BLISS-36 program. This program is the completed translation of the original source language program. Most of the CODE phase is straightforward and requires little work, since IL is similar to BLISS-36.

The CODE phase does, however, make all decisions about object representation and creation. Of particular interest is the handling of objects with a Union descriptor (section 6.1) because CODE attempts to be clever about storage allocation for these objects. To do this, CODE views the physical representation of an object as consisting of two parts: a *static part* and a *dynamic part*. Every object has a static part that is the part of the object whose size can be determined at compile time. For example, on many target machines the static part of an integer is a single word. The static part of an array is usually referred to as its dope vector and contains information about the bounds of the array and the location of the array components. The static part of an object is allocated at subprogram entry in the static part of the subprogram frame.

The dynamic part of an object is everything that did not appear in the static part. Not all objects have a dynamic part; integer and boolean objects do not, for example. An array does have a dynamic part, which is the storage for the array components. The dynamic part of an object is allocated at declaration time in the dynamic part of the run-time stack.

An object with a Union descriptor also has a static part and a dynamic part. The static part has the same size as the largest static part of the descriptors specified in the Union descriptor. For example, the IL translation of the source language declaration of `Temp`, following:

```
generic
   type T is private;
procedure P();

procedure P() is
   Temp : T;
. . .
```

might be (assuming appropriate generic instantiations)

```
local Temp : desc(Union, desc(Integer),
                         desc(Boolean),
                         desc(Array,
                              desc(Integer),
                              desc(Enumeration, 128),
                              1, 10));
```

In this case, `Temp` could be bound to the address of an integer object, a boolean object or an array

object. The array has the largest static part, since its dope vector requires several words (five words in the experimental compiler). So the declaration of Temp is translated to the BLISS-36 code

```
local Temp : Vector[5];
```

which allocates five words for Temp.

The routine Create, which is part of the run-time system (section 8.4), is responsible for allocating and initializing the dynamic parts of objects with Union descriptors. This routine also initializes the static part to allow access to the dynamic part. The Create routine was discussed in section 6.1; recall that invocations of this routine are implicit in variable declarations but become explicit for the creation of objects with Union descriptors. So, the declaration of Temp is translated to

```
routine P(T : desc(Descriptor)) =
begin
    local Temp : Vector[5];

    Create(Temp, .T);
    . . .
```

8.3. The Back End

The Back End comprises the BLISS-36 compiler [DEC 78a] and the linker [DEC 80]:

BLISS-36 compiler — Converts the BLISS-36 source program to PDP-20 relocatable code.

Linker — Merges the relocatable code with the run-time system (section 8.4) to form an executable program.

The BLISS-36 compiler and the linker are standard products for the PDP-20.

8.4. The Run-time System

A small number of BLISS-36 routines form a run-time system for executing programs. The routines are divided into three groups, or modules, which are described in table 8-1. The table includes the size in words (*code+data*) of each group of routines. The source code for the routines in ALLOC and IO is included in appendix C. The code for SPACE is not included since this module is part of the BLISS routine library.

Module	Function	Size (code + data)
ALLOC	object creation	43 + 0
IO	I/O	124 + 0
SPACE	dynamic space allocation	284 + 127

Table 8-1: Run-time Modules

Part Three

Evaluation and Conclusions

Chapter 9
Test Programs, Statistics and Evaluation

Chapters 5 through 7 suggest polymorphic routine translation (PR) as a technique for generating compact code for generic subprograms. Chapter 8 describes a compiler that implements both this technique and the alternative method generic expansion (GE). This chapter evaluates the PR translation technique based on data obtained by executing the compiler on test programs written in the source language (chapter 3).

The selection of the test programs, and statistics about them, are described in section 9.1. Some notations that are used throughout this chapter are defined in section 9.2.

The most important criterion for judging a compilation method is the quality of the code it generates. In this work quality is measured by size, and section 9.3 gives statistics about the object code size of each test program as compiled by GE and PR. For some of the test programs the PR technique yields a considerable savings in size. For the other test programs, however, the two methods produce similar results, and for these programs the compilation times for GE are shorter than the times for PR. Statistics about the compilation times of the test programs are provided in section 9.4.

The strategy of the experimental compiler is to apply a single translation technique—GE or PR—to the whole program. It might seem that superior results could be obtained by choosing an appropriate technique for each generic subprogram. This strategy was tried and, as described in section 9.5, proved to be inferior.

Because of the difference in compilation times one would like to determine at compile time those programs that will fare better under PR. If the determination could be made cheaply, this would obtain shorter compilation time and less object code. A statistic, described in section 9.6, was developed that estimates the desirability of PR over GE for a program by simulating the translations performed by the two techniques. For each test program this statistic, called ρ, can be used to predict the appropriate translation method.

Despite the savings in space attained, PR might be undesirable if there were a stiff penalty to pay in the execution speed of the resultant code. Section 9.7 gives statistics about the execution times of the test programs. These measurements show that there is indeed an execution speed penalty, but that it is tolerable.

Section 9.8 gives statistics that demonstrate the value of performing inline substitution. The final section is a summary of this chapter.

9.1. Test Programs

The test programs provide the experimental compiler with a set of data points for obtaining measurements. One would like to believe that these measurements give an indication of how the compilation techniques will perform in a non-research situation with a production quality compiler. This indeed would be the case if the test programs are representative of those that appear in a realistic environment. Unfortunately, few implemented languages provide generic facilities and there is, therefore, little data to direct the choice of test programs. In addition, no one knows how generic subprograms will be used until a language that implements them efficiently is available widely. It is not sufficient, therefore, to look at only existing programs that use generic subprograms.

The best that can be done is to use a variety of test programs and hope to capture an appropriate set of data points. The test programs in this work were chosen with this view. One program was rewritten from a routine used in the CLU compiler [Liskov 77].[52] Another program was designed to push hard on the use of generic instantiations, so instantiations were nested deeply within each other. There are only 12 test programs because these programs were difficult and time-consuming to produce. Each test program required the debugging of three programs: the source language program, the compiler and the BLISS-36 code.

Most of the programs show realistic uses of generic subprograms. There are several programs that display an unrealistic use of generic subprograms; these programs were constructed to yield a predetermined value for the statistic ρ to test the statistic's discriminative ability. The test programs also include large programs and small programs; there are programs with a heavy use of generic subprograms and programs with little use of such subprograms.

Table 9-1 provides a list of each program identified by name, a brief description of the program's

[52] The CLU program was supplied by R. W. Scheifler of M.I.T.

function and an indication of the nature of the program. Table 9-2 gives, for each test program, its size in lines of source code and the static and expanded counts of the generic subprograms and generic instantiations in each program. The static counts are obtained by counting the generic subprogram definitions and generic instantiations in a program. The expanded counts are made after expanding fully the generic instantiations. The static and expanded counts may be different if a generic subprogram contains a generic instantiation. For example, in the following:

```
generic
   type T is private;
procedure P();

procedure P() is
   procedure New_Q is new Q(T);                    -- 1
   . . .
end P;

procedure New_P1 is new P(Integer);                -- 2
procedure New_P2 is new P(Character);              -- 3
```

the static number of generic instantiations is three (italicized numbers). But the expanded count for this fragment is four, since there are two generic instantiations of P, and each introduces a generic instantiation of Q.

Program	Function	Notes
Arrtst	Reading & writing arrays	*Large*
BLT	Array assignment	*Small*
Calc	Desk calculator	*Large*
Comand	Command parser	*Heavy use of generics*
List1	Generic list parsing	*Heavy use of generics*
Sort1	Generic bubble sort	*Light use of generics*
Sort2	Generic heapsort	*Light use of generics*
Sort3	Uses heapsort	*Light use of generics*
Sort5	Uses bubble sort	*Constructed synthetically: light use of generics*
Sort6	Uses bubble sort	*Constructed synthetically: heavy use of generics*
Sort7	Uses bubble sort	*Constructed synthetically: heavy use of generics*
Sym1	Generic symbol table	*Small*

Table 9-1: Descriptions of Test Programs

Program	Size	Static Count		Expanded Count	
		Subprograms	Instantiations	Subprograms	Instantiations
Arrtst	204	4	12	4	12
BLT	50	1	1	1	1
Calc	508	6	12	6	12
Comand	330	8	18	8	24
List1	282	7	19	7	23
Sort1	79	2	2	2	2
Sort2	93	1	1	1	1
Sort3	107	5	9	5	10
Sort5	119	5	9	5	10
Sort6	230	5	9	5	10
Sort7	107	5	33	5	40
Sym1	165	2	2	2	2

Table 9-2: Statistics of Test Programs

9.2. Notation

Before proceeding with the remainder of the statistics it is useful to introduce notations for the measurements that denote the *object code size*, the *compilation time* and the *execution time* of a test program. The meaning of these measurements will be described precisely in the sections following. The notations are described in table 9-3; in each case, P is a test program.

Notation	Meaning
$Size_{GE}(P)$	Size of P compiled by GE
$Size_{PR}(P)$	Size of P compiled by PR
$Ctime_{GE}(P)$	Compilation time of P by GE
$Ctime_{PR}(P)$	Compilation time of P by PR
$Etime_{GE}(P)$	Execution time of GE translation of P
$Etime_{PR}(P)$	Execution time of PR translation of P

Table 9-3: Notations for Measurements

9.3. Object Code Size

Table 9-4 lists object code sizes for each test program when compiled by the GE and PR methods. The object code size is the sum of code and data spaces as given by the BLISS-16 compiler.[53] The table also includes the statistic $sr(P)$, which is a measure of the desirability of PR over GE for program P. The statistic *sr* (size ratio) is defined as follows:

$$sr(P) \triangleq Size_{GE}(P) \ / \ Size_{PR}(P)$$

If GE and PR produce object code programs of equal size when applied to program P, then it will be the case that

$$sr(P) = 1$$

If GE produces less code than PR for program P, then

$$sr(P) < 1$$

Finally, PR produces a more space-efficient translation for program P in case

$$sr(P) > 1$$

Program	$Size_{GE}(P)$	$Size_{PR}(P)$	$sr(P)$
Arrtst	1085	1091	0.99
BLT	224	219	1.02
Calc	1548	1556	0.99
Comand	1885	1331	1.42
List1	2604	1176	2.21
Sort1	190	190	1.0
Sort2	251	251	1.0
Sort3	309	384	0.8
Sort5	389	424	0.92
Sort6	1450	907	1.6
Sort7	1079	787	1.37
Sym1	684	684	1.0

Table 9-4: Object Code Size of Test Programs (words)

PR performed significantly better than GE on four of the 12 test programs: Comand, List1, Sort6 and Sort7. For six of the test programs the values of *sr* are close enough to make GE and PR equally desirable. These programs are Arrtst ($sr = 0.99$), BLT ($sr = 1.02$), Calc ($sr = 0.99$), Sort1 ($sr = 1$), Sort2 ($sr = 1$) and Sym1 ($sr = 1$). In two cases—Sort3 and Sort5—GE produced results superior to PR.

[53] The BLISS-16 ... program's run-time stack. The size of this segment was

PR attempts to save space by sharing a single generic subprogram among multiple generic instantiations. To do this, the translation of a generic subprogram is made general enough to operate on objects with types that can differ for each invocation. GE, on the other hand, produces a customized copy of the generic subprogram for each generic instantiation, and the code operates on objects with fixed types. It requires more code to be general, so the PR translation of a generic subprogram usually is larger than the GE translation. In addition, a small amount of code is generated for each generic instantiation under PR.

If there is a single generic instantiation of a generic subprogram, the overhead required by PR will likely result in a larger program than GE would produce. But much of the additional code is removed by inline substitution during the OPT phase. This explains why PR never does much worse than GE.

It seems that PR would pay off for programs in which the average number of generic instantiations for each generic subprogram is large, because the overhead required by PR for each generic subprogram will be overcome by the larger size of generic instantiations produced by GE. An inspection of table 9-2 shows that the average number of instantiations per generic subprogram is as follows (using the expanded counts):

Arrtst	3.0
BLT	1.0
Calc	2.0
Comand	3.0
List1	3.3
Sort1	1.0
Sort2	1.0
Sort3	2.0
Sort5	2.0
Sort6	2.0
Sort7	8.0
Sym1	1.0

The average number of instantiations is not a sufficiently sensitive measure to predict the effects of GE and PR reliably; the results above do not separate the three classes of programs. This measure is inadequate because it does not include anything about the structure of the program. In particular, no account is made of the size of a generic subprogram. The size is crucial because the larger the generic subprogram, the more advantageous it is to share its code. The measure also fails to consider how much of the program is inside of generic subprograms; if this percentage is small, the size of the code generated to handle generic subprograms and instantiations becomes insignificant. A more discriminating measure is described in section 9.6.

It seems that the compiler could be simplified by eliminating the use of the GE method. After all, even in the worst cases PR produced code comparable in size to that produced by GE. It would be desirable to use GE, however, if compilation by PR required significantly more time for some programs.

To test this possibility, each phase of the Translator was instrumented to determine the compilation times under the PR and GE translation techniques.[54] Table 9-5 lists the compilation times in seconds for each program by GE and PR. The times given are the sum of the times for the phases of the Translator (INFO, TRANSLATE, OPT and CODE).[55] Also included in the table is the statistic cr(P). This statistic is similar to *sr* as it measures the desirability of PR over GE. In this case, desirability is earned by requiring less compilation time, so *cr* (compilation-time ratio) is defined as

$$cr(P) \triangleq Ctime_{GE}(P) \;/\; Ctime_{PR}(P)$$

Program	$Ctime_{GE}$(P)	$Ctime_{PR}$(P)	cr(P)	sr(P)
Arrtst	27.04	45.89	0.59	0.99
BLT	1.16	4.63	0.25	1.02
Calc	33.43	86.78	0.39	0.99
Comand	90.98	85.22	1.07	1.42
List1	86.28	48.41	1.78	2.21
Sort1	2.19	8.05	0.27	1.0
Sort2	3.15	10.79	0.29	1.0
Sort3	9.6	17.24	0.56	0.8
Sort5	8.96	17.08	0.52	0.92
Sort6	72.44	49.19	1.47	1.6
Sort7	48.34	30.45	1.59	1.37
Sym1	6.82	23.49	0.29	1.0

Table 9-5: Compilation Times of Test Programs (seconds)

The astonishing thing about the compilation times is that PR performs faster than GE for those programs with a high value for *sr*. These are, of course, just the programs for which PR translation saves object code space. For those programs that are smaller when compiled by GE, the GE compilation requires less time than compilation by PR.

[54] The times were obtained using the METER% JSYS on the PDP-20 [DEC 78b], which provides accurate and repeatable timings. Each timing run was performed twice and the values obtained were averaged. The worst discrepancy between two runs of the same phase using the same translation method was less than 0.1%.

[55] The time required to perform the input and output of the program graphs for each phase is excluded. This yields a more accurate model of a production-quality compiler, which would likely avoid the intermediate phase I/O.

This is surprising. Inline substitution is a time-consuming optimization, and PR requires many more applications of the transformation than GE does. Not only that, but the INFO phase must perform additional work for PR, further increasing the compilation time. A breakdown of compilation times by phases shows that this is indeed true for all programs.[56] But for those programs that fare better under PR, the TRANSLATE and CODE phases of GE took longer than the corresponding PR phases. This increase in time more than makes up for the additional time required for the INFO and OPT phases under PR.

GE requires no time to translate a generic subprogram. The time for translation of a generic instantiation under GE is linear in the size of the target generic subprogram, since this subprogram is copied. Under PR, translation of a generic subprogram is linear in the subprogram's size, but PR translates a generic instantiation in constant time. PR uses additional time generating the implicit declarations of primitive actions, and there are usually many of these.

If a generic subprogram is instantiated once, then PR will do much more work than GE and require more compilation time. But when the amount of code that is copied under GE is large enough, the time required for each generic subprogram outstrips the overhead of PR.

Of course, code generation requires time that is linear in the size of the object code, and the CODE phase of GE takes longer than PR just when GE produces more code than PR.

9.5. A Hybrid Translation Strategy

The behavior of GE and PR seems to make it desirable to develop a hybrid translation strategy that chooses a translation method for each generic subprogram rather than for the program as a whole. Such an approach would expand generic subprograms that are rarely instantiated but apply PR to the remaining generic subprograms. It seems that this technique should generate code whose size is comparable to the size of the object programs produced by GE or PR. In addition, this hybrid method should require less compilation time than GE or PR because for some generic subprograms the proliferation of small routines generated by PR is avoided.

To test this hypothesis, the experimental compiler was modified to use a hybrid strategy: GE was applied to generic subprograms that were instantiated once, and PR was used on the other generic subprograms. Surprisingly, the results showed that this hybrid strategy was inferior to using the

---

[56] The times for all phases executing on each test program are displayed in appendix E.

better of GE and PR on a program. For each program, the hybrid method did, indeed, produce code that was comparable in size to the best object program produced by GE and PR. But the compilation times for the new scheme were longer than or equal to (in two cases) the times for the better of the non-hybrid methods.

This result is not surprising for those programs that did better under GE, because it is not worth sharing code among generic instantiations in these programs. The hybrid method, therefore, wastes time determining generic subprograms to share and then spends additional time trying to optimize all the routines and invocations generated by this strategy.

The results for the programs that did better with PR are harder to describe. For these programs, the hybrid method spent more time in the OPT phase than PR spent. This is counter-intuitive: the hybrid method expands some generic subprograms and the expansions are already in a compact form because they contain more fixed types than their PR translations would contain. This should reduce the amount of work done during optimization.

The problem is that routines containing contain generic instantiations become larger when GE is performed on some generic subprograms. This is because GE replaces a generic instantiation with a customized copy of the target generic subprogram, while PR translates a generic instantiation into a small routine. These large routines slow down inline substitution since the analyses done during this optimization are performed on whole routines. This defect may not be inherent in a hybrid strategy: it may be possible to revise the analysis techniques to avoid the overhead caused by the larger routines.

9.6. The Statistic ρ

It would be nice if the compiler could predict the desirable translation method for a program. The ideal situation would be the discovery of a statistic that is simple to compute on the source program. This statistic could be computed during the first phase of compilation and its value used to direct the translation. A statistic that worked in this manner for the test programs was developed. This statistic is called ρ, and its derivation is described in this section, followed by graphs of the object code sizes and compilation times of the test programs plotted against ρ.

The statistic ρ estimates the sizes of the translations obtained by GE and PR. Understanding the derivation of ρ requires a high-level view of the GE and PR techniques. To understand this view consider that a source language program may be viewed as composed of three parts:

- *non-generic code:* all statements and declarations, except generic instantiations, that occur outside of generic subprograms;
- *generic code:* all statements and declarations, except generic instantiations, that occur inside of generic subprograms;
- *generic instantiations:* all generic instantiations.

The size of a translated program may be estimated to be the size of the non-generic code plus the size of the code produced for the generic code and generic instantiations, which will be different for the two translation methods. An easy way to estimate the size of translated source code is to count the number of lines in the source code. The mapping between source language lines and target language words is not one-to-one, but this technique frequently produces sufficient results. Thus, all sizes of source language code are measured in lines. For example, the following program fragment:

```
while J <= A'LAST loop                -- 1
   if A(I) > A(J) then                -- 2
      Swap(A(I), A(J));               -- 3
   end if;                            -- 4
   J := SUCC(J);                      -- 5
end loop;                             -- 6
```

has size six, counted as shown by the italicized numbers. The way in which lines are counted is not important as long as the method is consistent, always yielding the same value for the same program fragment.

GE translates a generic instantiation by creating a copy of the target generic subprogram. The generic subprogram is not translated. So the size of a program translated by GE is the size of the non-generic code plus, for each generic instantiation, the size of its target generic subprogram. Because each generic instantiation is expanded fully, the number of generic instantiations to consider is the expanded count (described in section 9.1).

Using the notations $\hat{ge}(P)$ for the estimated size of program P as translated by GE, and $Inst_E(P)$ for the set of expanded generic instantiations in program P, $\hat{ge}$ is defined as

$$\hat{ge}(P) \triangleq (\textit{size of non-generic code}) + \sum_{I \in Inst_E(P)} (\textit{size of generic subprogram of } I)$$

PR translates a generic instantiation into a small routine that invokes its target generic subprogram. Each routine can be approximated by the source language subprogram schema

```
procedure instantiation() is
begin
   generic subprogram(parameters);
end New_P;
```

which is four lines long, so four is used as the size of a translated generic instantiation. Since no expansion is performed, only the set of static generic instantiations (section 9.1) is considered.

Under PR, each generic subprogram is also translated into IL code. It is reasonable to estimate the size of each translated generic subprogram as the number of lines of source code it contains. Using the notations $\hat{pr}(P)$ for the estimated size of program P translated by PR, and $Inst_S(P)$ for the set of static generic instantiations in P, the definition of $\hat{pr}$ is

$$\hat{pr}(P) \triangleq (\textit{size of non-generic code}) + (\textit{size of generic code}) * 4|Inst_S(P)|$$

The statistic ρ serves the same purpose as the statistic sr: it is a ratio that indicates the desirability of PR over GE. Thus, ρ is defined as

$$\rho(P) \triangleq \hat{ge}(P) / \hat{pr}(P)$$

It would be nice if, like the other ratios in this chapter, PR is desirable for a program P if

$$\rho(P) > 1$$

Unfortunately, this is not so, as seen by an inspection of table 9-6, which lists each test program along with its values of $\hat{ge}$, $\hat{pr}$ and ρ.

Program	$\hat{ge}$(P)	$\hat{pr}$(P)	ρ(P)	sr(P)
Arrtst	284	252	1.13	0.99
BLT	50	54	0.93	1.02
Calc	555	556	1.0	0.99
Comand	678	402	1.69	1.42
List1	621	358	1.73	2.21
Sort1	79	87	0.91	1.0
Sort2	93	97	0.96	1.0
Sort3	161	143	1.13	0.8
Sort5	185	155	1.19	0.92
Sort6	407	266	1.53	1.6
Sort7	485	239	2.03	1.37
Sym1	165	173	0.95	1.0

Table 9-6: $\hat{ge}$, $\hat{pr}$ and ρ for Test Programs

But all is not lost. Figure 9-1 is a plot of ρ versus sr for the test programs. The dashed line is drawn at $sr = 1$; points above this line indicate programs for which PR produces a more compact translation than GE. The points above and below the line $sr = 1$ are separated horizontally by a line determined empirically to be at around $\rho = 1.3$.

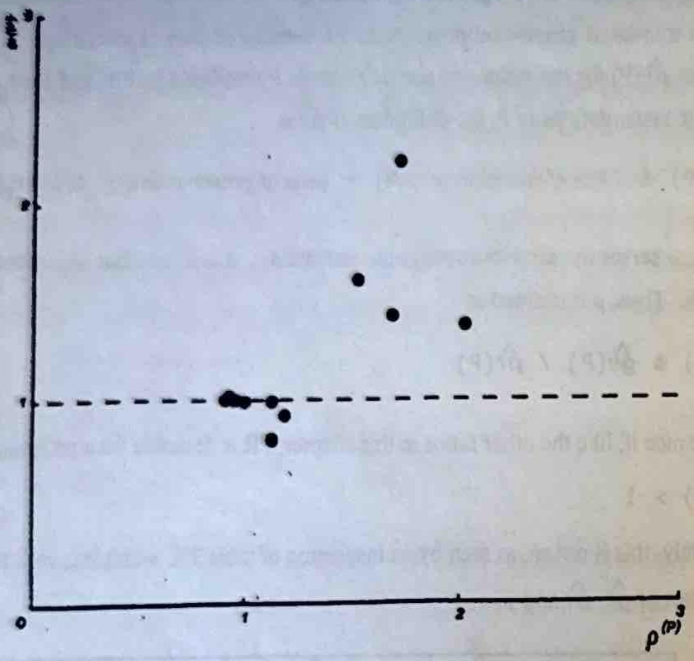

Figure 9-1: ρ(P) versus *sr*(P)

To test the discrimination ability of ρ, three test programs were constructed to have particular values for ρ. The programs, which have been included in all of the statistics so far, are Sort5, Sort6 and Sort7. These programs were designed so that their ρ values fell to the sides of the line ρ=1.3. The ρ values of the three programs are

```
ρ(Sort5) = 1.19 (< 1.3)
ρ(Sort6) = 1.53 (> 1.3)
ρ(Sort7) = 2.03 (>> 1.3)
```

As seen in table 9-6, ρ predicted the appropriate translation method for each program, taking ρ(P)>1.3 to indicate that P should be translated by PR.

The value ρ=1.3 also differentiates the programs that will compile faster under PR from those that should use GE, as can be seen in figure 9-2, plotting ρ(P) versus *cr*(P). This result is not surprising since the faster compilation time goes to the translation method that produces the least code.

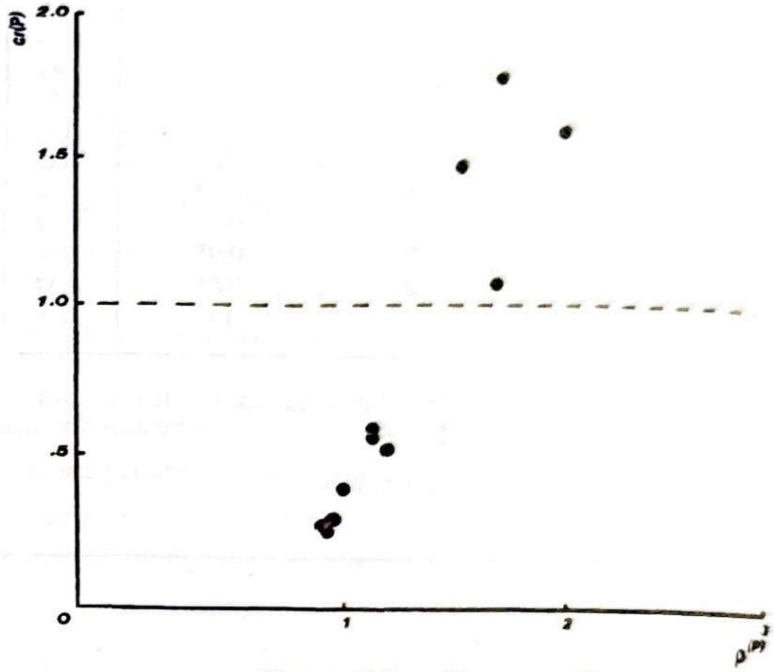

Figure 9-2: $\rho(P)$ versus $cr(P)$

9.7. Execution Time

The previous sections showed that the PR translation technique produces significant savings in object code size for some programs. For these programs, the PR translation also compiles faster than the GE translation. But it is expected that programs translated by PR will execute more slowly than the GE translation due to the additional routine invocations. If this execution overhead were small, PR would be extremely desirable.

To compare execution times, the translation of each test program was instrumented to measure its execution time. Both translations of each test program were then executed on identical test data and the results collected. These results are provided in table 9-7. The fourth column of the table lists the value of the ratio $er(P)$ (execution time ratio), which is defined similarly to sr and cr, as follows:[57]

$$er(P) \triangleq Etime_{GE}(P) \;/\; Etime_{PR}(P)$$

[57] The times are measured in milliseconds, but the actual values are unimportant. Only the values of er are relevant.

Program	$Etime_{GZ}(P)$	$Etime_{PR}(P)$	er(P)	sr(P)
Arrtst	295	295	1.0	0.99
BLT	41	44	0.93	1.02
Calc	131	162	0.81	0.99
Comand	89	101	0.88	1.42
List1	182	195	0.93	2.21
Sort1	34	34	1.0	1.0
Sort2	52	52	1.0	1.0
Sort3	53	65	0.82	0.8
Sort5	72	85	0.85	0.92
Sort6	193	204	0.95	1.6
Sort7	22	26	0.85	1.37
Sym1	143	143	1.0	1.0

Table 9-7: Execution Times of Test Programs (milliseconds)

Figure 9-3 shows a plot of ρ versus *er* for the test programs.

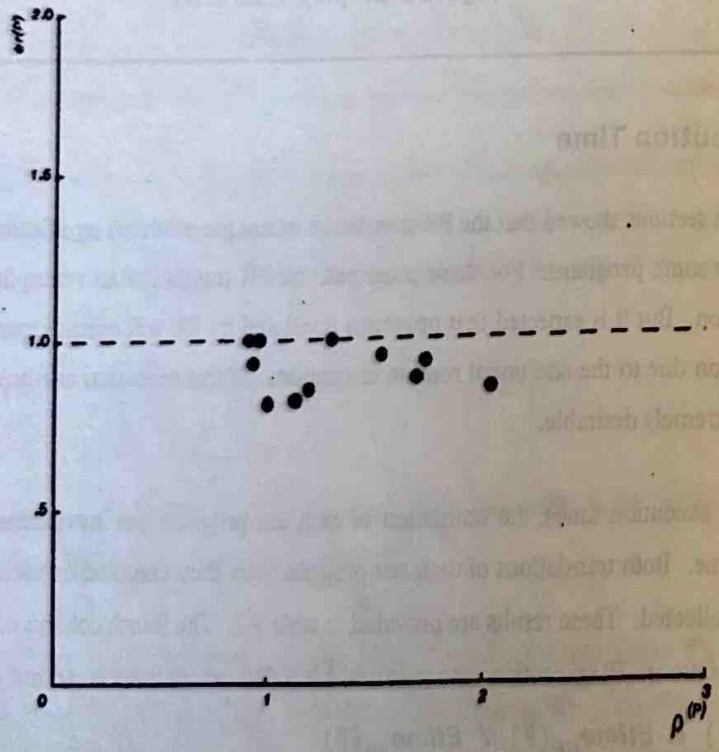

Figure 9-3: $\rho(P)$ versus *er*(P)

The four test programs with $sr \geq 1$ are the programs that concern us, since the other programs will be compiled by GE. For these test programs, the values of er fall within the interval (0.85, 0.95). It would be nice if the value of er for any program translated by PR could be guaranteed to lie in this range; 15% is a small price to pay. Unfortunately, er can, in principle, become arbitrarily small.

To see this, consider where the execution time overhead for a PR translation comes from. The GE translation of a generic instantiation is a copy of the target generic subprogram. Under PR, a generic instantiation becomes a routine that invokes the generic subprogram. The invocation of a generic instantiation in a PR translation will thus entail one more routine invocation than the GE translation. In addition, each copy of a generic subprogram under GE is customized and presents opportunities for optimization by global flow analysis. The PR translation of a generic subprogram is general and does not provide the compiler with as many possibilities for optimization. Many of the additional routine invocations produced by PR may be eliminated during the optimization phase. It is possible, however, to construct programs where the invocations will remain. If a program contains enough invocations of generic instantiations, the execution time overhead for its PR translation can become as large as desired. This did not occur for any of the test programs, and it seems unlikely to occur in practice.

9.8. Inline Substitution

Inline substitution is the primary optimization used to improve the results of PR translation. Although the space requirements of a GE translation may also be decreased by inline substitution, the effect is not as pronounced as for PR. To determine the effect of inline substitution the experimental compiler may be executed so that no inline substitutions are performed. The compiler was run this way on each test program, under both GE and PR, and the results are listed in tables 9-8 and 9-9. For each translation method, the columns labeled **IS Ratio** contain the ratio of the size of the translation of program P without inline substitution to the size of its translation with inline substitution, as follows:

$$\textbf{IS Ratio} \triangleq (\textit{size without IS}) \ / \ (\textit{size with IS})$$

The tables demonstrate that inline substitution has greater effect on the results of PR than GE. The IS ratios for the GE translations range from 1.04 to 2.03 with the average value around 1.3. The average IS ratio for the PR translations is about 1.9, with the lowest ratio 1.43. This is to be expected considering the small routines created indiscriminately by PR.

Program	$Size_{GL}$(P) without IS	with IS	IS Ratio
Arrtst	1320	1085	1.22
BLT	239	224	1.07
Calc	1685	1548	1.09
Comand	2111	1885	1.12
List1	2976	2604	1.14
Sort1	252	190	1.33
Sort2	308	251	1.23
Sort3	627	309	2.03
Sort5	533	389	1.37
Sort6	1503	1450	1.04
Sort7	1576	1079	1.46
Syml	725	684	1.06

Table 9-8: Improvement of GE Object Code Size Due to IS

Program	$Size_{PR}$(P) without IS	with IS	IS Ratio
Arrtst	1785	1091	1.64
BLT	345	219	1.58
Calc	2701	1556	1.74
Comand	2229	1331	1.67
List1	1909	1176	1.62
Sort1	516	190	2.72
Sort2	598	251	2.38
Sort3	850	384	2.21
Sort5	804	424	1.9
Sort6	1583	907	1.75
Sort7	1123	787	1.43
Syml	1259	684	1.84

Table 9-9: Improvement of PR Object Code Size Due to IS

It is interesting to note that without inline substitution, only two test programs, List1 and Sort7, have a smaller PR translation. Inline substitution is certainly desirable for PR.

9.9. Summary

This chapter has presented results obtained by executing the experimental compiler on a group of 12 test programs. The results showed that programs seem to be divided into two disjoint groups. The first group consists of those programs that are translated to object code of comparable sizes by GE and PR. But because of faster compilation time, GE is the method of choice for these programs. The second group comprises programs whose PR translation yields less object code than GE translation. In addition, the PR translation of a program in this group is faster than the translation of the program by GE.

The savings in code space attained by PR can be significant: four of the test programs were over 30% larger when translated by GE. When the two translation methods yielded object programs of comparable size, there was compelling motivation for using GE. For the seven programs that fall in this class, GE translation was over 25% faster than PR translation, with an average savings of about 45%.

Given these statistics, it seemed that a hybrid translation strategy might produce the best possible results. A hybrid method would choose an appropriate translation technique for each generic subprogram rather than for the entire program, as the experimental compiler does. The compiler was modified to use a hybrid strategy that shared code for a generic subprogram if it was instantiated more than once. This scheme did produce code comparable in size to the best non-hybrid translations. Unfortunately, the new strategy required more compilation time than the original program-wide strategy. The additional compilation time occurred during optimization (sections 7.3 and 8.2.3) because performing GE on some generic subprograms, and expanding them, produces larger routines. These routines require more time to analyze than the smaller routines produced by PR.

Because of the difference in compilation times for GE and PR, it is desirable for the compiler to be able to choose the best translation technique for each program. The ideal situation would be the discovery of a measure that was cheap to calculate on the source program and could be used to direct translation. Section 9.6 described a statistic, ρ, that is a first step towards finding such a measure. This statistic, whose determination is inexpensive, could be used to predict the desired translation method for each of the test programs. There is no evidence that such a simple measure will have widespread utility, but the results provide the possibility that inexpensive measures can be found.

Section 9.7 showed that the space saved by PR does not come free: the resulting object code executes more slowly than the GE translation. This is not surprising since the PR translation does the same work as the GE translation, but with additional routine invocations. Although the execution speed

penalty for PR translation can be arbitrarily large, the test programs suffered only small slowdowns. The worst case was a program whose PR translation executed 23% slower than its GE translation, and the average slowdown was only 10%.

Chapter 10
Conclusions

Abstraction—the task of isolating what is relevant—is an important concept for reducing the complexity of a task. The complicated process of programming is especially amenable to abstraction. The programmer has a number of language tools available to aid abstraction. These tools include subprograms, data typing and encapsulation mechanisms. Among the newest tools suggested are generic facilities, which allow definitions to be parameterized by types. But until there are efficient implementations of languages with generic facilities, we can only surmise about the value of this tool.

To enable a fair test of the utility of generic facilities, programmers must perceive the implementation of these facilities as efficient. The use of a language will otherwise be contorted; programmers are notorious for their avoidance of language features suspected of being inefficient. Generic expansion (GE), the obvious translation technique for generic facilities, may not be good enough to assuage programmer's fears. GE generates replicate code, and heavy use of generic instantiations in a program may lead to unduly large object code. In addition, the compilation time required for GE is sometimes greater than necessary.

Generic subprograms are a crucial component of a generic facility, as well as being useful in their own right. By focusing on generic subprograms this dissertation has taken the first step in investigating the production of compact code for generic facilities. The primary thrust of this research is that a compiler can do better by not using GE as the sole translation method for generic subprograms. Smaller object code is obtained by amending the compiler with a new translation technique—polymorphic routine translation (PR).

PR takes a stance opposite that of GE by refusing to copy a generic subprogram for each generic instantiation. Instead, PR shares a single generic subprogram among its instantiations. Generating code that operates correctly under these conditions requires some care and the dissertation contains a detailed description of the concepts and methods involved in implementing PR.

The PR technique treats each predefined operation, such as integer assignment, as if it were a generic instantiation of a predefined generic subprogram. Much of a program can thus be represented by generic subprograms, generic instantiations and subprogram invocations. This allows a small number of techniques to be used to translate a program and guarantees that user-defined subprograms are provided with the same chances for optimization as predefined operations.

The translation, which is table-driven to be machine independent, produces a representation of the program in an intermediate language, IL. This representation consists of only a few primitive actions, the primary one being routine invocation. Thus, the code generator is simple because it need only concentrate on a small set of actions.

The experimental compiler, built to compare GE and PR, demonstrates that there are programs for which PR yields a significant savings in object code size and compilation time. For the remainder of the programs, the sizes of the object code produced by GE and PR are about the same size. This seems to indicate that a compiler could be simplified by implementing only PR. A comparison of compilation times, however, reveals that GE is desirable for those programs whose object code sizes are about the same under GE and PR.

Statistics gathered on a group of test programs show that these programs fall into two classes. The first class contains programs that are translated by GE and PR into object code of comparable size. For these programs the compilation time for GE is shorter than the compilation time for PR. The second class comprises programs that have significantly smaller object code sizes when translated by PR. In addition, the compilation time by PR for these programs is less than the time required for GE translation.

An ideal situation would allow the compiler to predict the desired translation method for a program. Each program could then be compiled in the manner that requires the least time and produces the smallest object code. As a first step towards this goal, the dissertation developed a statistic, ρ, that estimates the desirability of PR over GE for the test programs. Measurements showed that the statistic could be used to divide the test programs into two groups: those programs that have a smaller object code size and require less compilation time with PR and those programs whose object code sizes are comparable under PR and GE but whose compilation by GE is faster than compilation by PR. There is no guarantee that this statistic will be universally applicable, but it is a promising beginning.

PR translation can save space, but at a price: the resulting program can be expected to execute more

slowly than its GE translation would. Although the slowdown could be large, in practice it seems to be acceptable, on the order of 10%.

The dissertation demonstrates that PR is a useful technique that is easy to implement. The use of GE and PR in conjunction with a predictive statistic, like ρ, can provide an efficient implementation methodology for generic subprograms. The next section discusses how the techniques in the experimental compiler might be used in a non-research environment.

A fringe benefit of this research was the discovery of a simple method for performing inline substitution. The reductions in code sizes for PR translations are inflated due to the large number of routines generated by PR. But there is nothing out of the ordinary in the results of GE translation. Inspection of the improvements in code size from applying inline substitution to GE translations indicates that the method can be expected to yield respectable results for programs in general.

10.1. Using PR and GE

The behavior of the statistic ρ presents hope that a simple measure may be found that can be used to predict the desired translation method for a program. This scenario is all well and good, but compilation time and object code size are not always the most important considerations for a program. There are programs for which speed of execution is crucial, and here the small degradation in the execution speed for PR translation is unacceptable. It is not, however, necessary to abandon the techniques of this dissertation for these programs. In fact, the use of PR and ρ can be of help in these situations.

Knuth [Knuth 71] observed that a typical program spends more than 50% of its execution time in about 4% of its code, and this fact can be used to advantage. The 96% of a program, where little time is spent, is submitted to the translation techniques suggested in this dissertation. This results in as little object code as possible for most of the program. The additional execution time required for this part of the program will not be noticed since the code is executed rarely.[58] The 4% of the program where most time is spent can usually be identified by instrumentation. This part of the program can then be compiled to obtain maximum execution speed. The compilation of this part can be by generic expansion, which normally achieves the fastest execution time. Or the part can be hand coded if greater efficiency is required.

[58] If the execution overheads of the test programs (section 9.7) are representative of all programs, then the slowdown for a typical program will be less than 50%×10%=5%.

10.2. Directions for Further Research

During the pursuit of a doctoral research topic unexplored pathways inevitably are discovered. But for one reason or another—usually time—these paths cannot be investigated in the dissertation. This work is no different. The following sections discuss some topics that suggested themselves during my research into generic subprograms.

10.2.1. Generic Packages

Although generic subprograms are useful, the real power of generic facilities promises to be found in the ability to define generic abstract data types. In the source language, an abstract data type is constructed most easily by a package definition (section 3.2.1), which encapsulates data and operations and enforces visibility restrictions. This section focuses on the translation of generic packages in the source language because these problems are representative of the difficulties that need to be solved to provide efficient implementations of generic abstract data types.

The translation of generic packages presents difficulties not faced in this work, due to the lifetime of entities declared in a package. Consider the following generic package:

```
generic
   type Elt is private;
   Buffer_Size : in Integer;
package Buffer_Output is

   procedure Write(E : in Elt);
   procedure Flush();

end Buffer_Output;


package body Buffer_Output is

   type Buffer_Type is array(Integer range <>) of Elt;

   Buffer : Buffer_Type(1..Buffer_Size);
   . . .

   procedure Write(E : in Elt) is ... ;
   procedure Flush() is ... ;

end Buffer_Output;
```

A subprogram defined within a generic package can be treated as a generic subprogram that inherits the generic formal parameters of the package. So `Write` is treated as if declared as

```
generic
   type Elt is private;
   Buffer_Size : in Integer;
procedure Write(E : in Elt);
```

If `Write` were a generic subprogram, the generic formal parameters of the package would be appended to those of `Write`.

This simple approach ensures that correct access is provided to the package's generic formal parameters. But the method fails to handle references from within the body of `Write` to objects declared within the package, known as *local objects*. The problem is that distinct generic instantiations have the effect of creating semantically distinct copies of all entities in the package. For example, the execution of the generic instantiations following:

```
package LPT_Buffer is new Buffer_Output(Character, 133);
package Disk_Buffer is new Buffer_Output(Integer, 512);
```

creates two copies each of subprograms `Write` and `Flush` and two copies of local object `Buffer`.

If the body of subprogram `Write` references the local object `Buffer`, for example as

```
procedure Write(E : in Elt) is
begin
   . . . Buffer(I) . . .
end Write;
```

consider what this reference to `Buffer` means when the subscript expression is executed as a result of the invocation

```
Disk_Buffer.Write(15);
```

The identifier `Buffer` is a reference to the object created by the generic instantiation of `Disk_Buffer`. But if the other copy of `Write` is invoked, for example, by the invocation

```
LPT_Buffer.Write('#');
```

then the reference to `Buffer` must be to the copy created for `LPT_Buffer`.

Since each version of a local object may contain a distinct value, copies must be made of the local objects for each generic instantiation. A manner of allowing access from the body of a shared subprogram to the proper copy of each object is needed.

The investigation into implementing generic packages does not end with local objects. A generic package may contain declarations of other generic packages, exceptions and entities besides objects. Methods must be developed to translate each of these entities in a space-efficient manner.

10.2.2. Operational Equivalence

It is possible that two generic instantiations of a single generic subprogram can be implemented identically. Consider, once again, the generic subprogram `Exchange` and the two generic instantiations of it that follow:[59]

```
generic
   type T is private;
procedure Exchange(X, Y : in out T);

procedure Exchange(X, Y : in out T) is
   Temp : T;
begin
   Temp := X;
   X := Y;
   Y := Temp;
end Exchange

procedure Int_Swap is new Exchange(Integer);
procedure Real_Swap is new Exchange(Real);
```

Within the body of `Exchange` the actions that depend on the generic type `T` are parameter passing, object creation and destruction and the three assignments. On many machines each action may be performed identically for objects of type `Integer` and `Real` because objects of either type occupy a single word and `Integer` and `Real` values are copied using the same machine instruction.

Implementing two generic instantiations identically is called *operational equivalence.* This optimization can be applied regardless of the translation method. If GE is being used, operational equivalence is especially useful, because only one copy of a generic subprogram need be made for several generic instantiations. The benefits of operational equivalence in PR likely will be less spectacular, because the translation of a generic instantiation is small.

The potential for operational equivalence occurs when all actions for two generic instantiations can be implemented by identical object code. Recall that actions include not only operations that appear explicitly (for example, `":="` and `"and"`), but also implicit operations such as parameter passing and function return. Following is an intriguing example of operational equivalence that involves only the implicit action of parameter passing.

```
generic
   type T is limited private;
   with function Convert(E : in T) return String;
procedure File_Write(E : in T);
```

---

[59] For this example, assume that there is a predefined type `Real` that defines single precision floating point numbers.

```
procedure File_Write(E : in T) is
begin
    Put(Out_File, "*Output: ");
    Put(Out_File, Convert(E));
end File_Write;
```

In this case, the only action within the subprogram that depends on the generic type T is parameter passing. In the experimental compiler, all objects of a limited private type use call-by-reference (section 6.4.1). Since T is a limited private type, we know a priori that all generic instantiations of `File_Write` can be implemented identically, and only a single copy of the subprogram will be needed.

10.2.3. Range Constraints and Constraint Checking

Ada [DOD 82] has a richer type system than the source language used in this dissertation.[60] Ada's type system provides many ways for the programmer to express his intentions so that the compiler may validate some of the uses of objects within a program.

One particularly useful feature is the *range constraint*, which allows the legal values for an integer object to be restricted to a subset of the integers. Each attempt to store a value in an integer object is checked to ensure that the value to be assigned is a member of the legal values for the object. Range constraints are useful since they allow the programmer to be explicit about his needs for an integer object. As an example of the use of a range constraint, consider the declaration of a function in the source language to perform the factorial function:

```
function Fact(N : in Integer) return Integer is
begin
    if N <= 1 then
        return 1;
    else
        return N * Fact(N-1);
    end if;
end fact;
```

This function would yield the value 1 for actual parameters that are negative. If this is undesirable the function could be modified to test explicitly for `N<0`.

In Ada, the definition of `Fact` could be made cleaner by use of a range constraint, as follows:

```
function Fact(N : in Integer range 0..Max) return Integer;
```

This makes the compiler responsible for checking that the value of `N` is non-negative.

[60] The source language, which is derived from Ada, is described in chapter 3.

The nice thing about range constraints is that a compiler can eliminate the need for a run-time check in many instances. In the invocation of `Fact` following:

```
Val : Integer range 1..10;
. . .
Put(Fact(Val));
```

a check on the value of `Val` is unnecessary (if $10 \leq$ `Max`).

Range constraints and constraint checking were not included in the source language because I saw no good way to handle them. The obvious way to obtain the checking would be to place the range test within the definition of assignment for an `Integer`, as follows

```
routine ASSIGN_Integer(Dest : desc(Pointer, Integer_Desc),
                       Source : Integer_Desc) =
begin
   if (.Source < minimum value of .Dest) or
      (.Source > maximum value of .Dest)
      then range violation
   fi;
   .Dest ← .Source
end Integer_Assign;
```

This works, but is likely to hinder the inline expansion of assignment for integers. Consider the assignment statement below:

```
X : Integer range 0..511;
Y : Integer range 1..128;
. . .
X := Y;
```

A compiler performing flow analysis would determine that no constraint check is needed for this statement. But the inline expansion of this statement in IL is, after some simplification,

```
begin
   if (.Y < 0) or (.Y > 511)
      then range violation
   fi;
   X ← .Y
end;
```

It requires constant propagation and flow analysis, which the experimental compiler does not have, to determine that the test in the `if` statement yields `true`.

Range constraints and constraint checking are desirable features, and a method of incorporating them into the compilation framework suggested in this dissertation would be helpful.

10.2.4. Unused Parameter Removal

There is an optimization, not done by the experimental compiler, that can be performed on the PR translation of a program. This optimization, *unused parameter removal*, was excluded because I thought it would prove unprofitable. In hindsight, I believe that the optimization is useful, especially since it is easy to perform.

Unused parameter removal operates on a routine and deletes formal parameters that are not referenced within the body of the routine. In addition, the corresponding actual parameters are removed from invocations of the routine.[61]

This optimization is unlikely to be worthwhile for user-defined routines but may be desirable for compiler-generated routines, especially the translation of generic subprograms. This is because a subprogram may not reference some of the implicit parameters generated during PR. This can happen if the code obtained ultimately for a generic subprogram does not depend on any of its implicit type parameters. For example, consider the generic function `Length`, which returns the length of its array parameter:

```
generic
   type Index is (<>);
   type Element is limited private;
   type Arr is array(Index range <>) of Element;
function Length(A : in Arr) return Integer;

function Length(A : in Arr) return Integer is
begin
   if A'LAST < A'FIRST then
      return 0;
   else
      return A'LAST - A'FIRST + 1;
   end if;
end Length;
```

After optimization, the PR translation of this function is

[61] Unused parameter removal may delete actual parameters that have side effects. This will be unacceptable if the source language requires that all actual parameters be evaluated, and the optimization will have to be modified to avoid removal of such parameters.

```
routine Length(A : descriptor,
               Index, Element, Arr : desc(Descriptor)) =
begin
   if .(.A+UB_Offset) < .(.A+LB_Offset) then
      return 0
   else
      return .(.A+UB_Offset) - .(.A+LB_Offset) + 1
   fi
end Length;
```

Because all arrays have dope vectors that are represented identically, the body of `Length` does not depend on any of the three implicit parameters `Index`, `Element` and `Arr`. Unused parameter removal would save the need to transmit three actual parameters for each invocation of `Length`.

This optimization will be especially important for the implementation of generic packages if the technique suggested in section 10.2.1 is employed. In this technique, all of the generic formal parameters of the package are appended to the parameters of each subprogram nested in the package. It is likely that many of these subprograms will have no need for some of these implicit parameters.

10.3. Summary

Generic facilities promise to be a useful abstraction tool for programming languages. To determine their utility requires efficient implementations of generic facilities to prevent programmers from contorting their programming.

This dissertation investigated the implementation of generic subprograms and thus began the development of good implementations of generic facilities. The research described here demonstrates that the translation of generic subprograms can be improved by augmenting the obvious translation method, generic expansion (GE), with the technique of polymorphic routine translation (PR).

By sharing a single translation of a generic subprogram among all of its instantiations, PR attempts to reduce the amount of code generated. Indeed, for some programs PR produces significantly smaller object code than GE produces. In addition, compilation by PR takes less time than compilation by GE for these programs. The remaining programs are translated into object programs of approximately the same size by GE and PR. But the compilation times for this group of programs is shorter for GE than the times for PR compilation.

These results led to the search for a statistic that could be measured on the source program and would predict the appropriate translation technique for a program. For the experimental compiler, built to

compare the GE and PR methods, a statistic was discovered that neatly divided the test programs into the two groups mentioned above. It is certainly not clear that the statistic, ρ, will produce such nice results for all compilers and target machines. Nonetheless, ρ presents hope that a simple, intuitive measure can be found that will enable prediction of the translation method.

This dissertation has demonstrated that there is a viable alternative to the GE translation method. The new method, polymorphic routine translation, is simple to implement and, if used in conjunction with GE and a predictive statistic, can provide an efficient implementation methodology for generic subprograms.

References

[Ada 79] Written under contract with the United States Department of Defense.
Preliminary Ada reference manual.
SIGPLAN Notices 14(6), June, 1979.
This is the first part of a special issue of *SIGPLAN Notices* about the Ada language. The second part of the issue contained the rationale for the design of the language [Ichbiah 79].

[Allen 77] F. E. Allen, J. L. Carter, W. H. Harrison, P. G. Loewner, R. P. Tapscott, L. H. Trevillyan and M. N. Wegman.
The Experimental Compiling Systems Project.
Technical Report RC 6718 (#28922), IBM Thomas J. Watson Research Center, September, 1977.

[ANSI 66] American National Standards Institute.
American National Standards FORTRAN (X3.9)
New York, 1966.

[Atkinson 78] Russell R. Atkinson, Barbara H. Liskov and Robert W. Scheifler.
Aspects of implementing CLU.
In *Proceedings of the ACM National Conference*, pages 123-129. December, 1978.

[Ball 82] J. Eugene Ball.
Program Improvement by the Selective Integration of Procedure Calls.
PhD thesis, University of Rochester, December, 1982.

[Callahan 77] J. J. Callahan.
The curvature of space in a finite universe.
In *Cosmology + 1*, chapter 3. W. H. Freeman, 1977.
Readings from *Scientific American.*

[CLI 75] Computer Linguistics Incorporated.
A Survey of Optimization Techniques in Compilers.
Technical Report RADC-TR-75-223, Rome Air Development Center, September, 1975.

[Cook 71] S. A. Cook.
The complexity of theorem-proving procedures.
In *Proceedings of 3rd Annual Symposium on Theory of Computing*, pages 151-158. 1971.

[DEC 78a] Digital Equipment Corporation.
BLISS Language Guide
Maynard, MA, 1978.
Order number AA-H275A-RK.

[DEC 78b] Digital Equipment Corporation.
DECsystem-10/DECsystem-20: Hardware Reference Manual (Volume I: Central Processor).
1978.
Order number EK-10/20-HR-001.

[DEC 80] Digital Equipment Corporation.
TOPS-20: Commands Reference Manual
Marlboro, MA, 1980.
Order number AA-5115B-TM.

[DOD 82] United States Department of Defense.
Reference Manual for the Ada Programming Language
1982.
Draft Revised MIL-STD 1815, Draft Proposed ANSI Standard Document for Editorial Review. A more recent version of this manual is available [DOD 83].

[DOD 83] United States Department of Defense.
Ada Programming Language
1983.
ANSI/MIL-STD-1815A-1983.

[Garey 79] Michael R. Garey and David S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-completeness.
W. H. Freeman and Co., 1979.

[Gehani 80] Narain Gehani.
Generic procedures: An implementation and an undecidability result.
Computer Languages 5(3/4):155-161, 1980.

[Goguen 78] J.A. Goguen, J.W. Thatcher and E.G. Wagner.
An initial algebra approach to the specification and implementation of abstract data types.
In R. Yeh (editor), *Current Trends in Programming Methodology*, pages 80-149. Prentice Hall, 1978.

[Gries 71] David Gries.
Compiler Construction for Digital Computers.
John Wiley and Sons, Inc., 1971.

[Habermann 83] A. Nico Habermann and Dewayne E. Perry.
Ada for Experienced Programmers.
Addison-Wesley, 1983.

[Hanson 78] David R. Hanson and Ralph E. Griswold.
The SL5 procedure mechanism.
Communications of the ACM 21(5):392-400, May, 1978.

[Hibbard 83] Peter Hibbard, Andy Hisgen, Jonathan Rosenberg, Mary Shaw and Mark Sherman.
Studies in Ada Style.
Springer-Verlag, 1983.
Second edition.

[Hilfinger 81] Paul Hilfinger, Gary Feldman, Robert Fitzgerald, Izumi Kimura, Ralph L. London, K. V. S. Prasad, V. R. Prasad, Jonathan Rosenberg, Mary Shaw and Wm. A. Wulf (editor).
(Preliminary) An informal definition of Alphard.
In Mary Shaw (editor), *ALPHARD: Form and Content* [Shaw 81], pages 195-252.
Springer-Verlag, 1981.

[Holt 79] Richard C. Holt and David B. Wortman.
A model for implementing Euclid modules and type templates.
In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 8-12.
August, 1979.

[Horowitz 78] Ellis Horowitz and Sartaj Sahni.
Fundamentals of Computer Algorithms.
Computer Science Press, Inc., 1978.
Chapter 4 of this book is about the greedy method of algorithm design.

[Ichbiah 79] J. D. Ichbiah, J. G. P. Barnes, J. C. Heliard, B. Krieg-Brueckner, O. Roubine and B. A. Wichmann.
Rationale for the design of the ADA programming language.
SIGPLAN Notices 14(6), June, 1979.
This is the second part of a special issue of *SIGPLAN Notices* about the Ada language. The first part of the issue contained the language reference manual for Ada [Ada 79].

[Intel 78] Intel Corporation.
Intel Component Data Catalog
1978.
Chapter 11 discusses the 8080 line of microprocessors.

[Jensen 78] Kathleen Jensen and Niklaus Wirth.
PASCAL User manual and Report.
Springer-Verlag, 1978.
Second edition.

[Knuth 68a] Donald E. Knuth.
Fundamental Algorithms.
Addison-Wesley, 1968.
Volume 1 in the series *The Art of Computer Programming.*

[Knuth 68b] Donald E. Knuth.
Semantics of context-free languages.
Mathematical Systems Theory 2(2):127-145, 1968.

[Knuth 71] Donald E. Knuth.
An empirical study of FORTRAN programs.
Software Practice and Experience 1(2):105-133, 1971.

[Lampson 77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell and G. L. Popek.
Report on the programming language Euclid.
SIGPLAN Notices 12(2), February, 1977.
This issue was devoted to the Euclid language reference manual.

[Lampson 82] Butler W. Lampson.
Fast procedure calls.
In *Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 66-76. March, 1982.

[Leverett 80] Bruce W. Leverett, Roderic G. G. Cattell, Steven O. Hobbs, Joseph M. Newcomer, Andrew H. Reiner, Bruce R. Schatz and William A. Wulf.
An overview of the Production-Quality Compiler-Compiler project.
Computer 13(8):38-49, August, 1980.

[Liskov 77] Barbara Liskov, Alan Snyder, Russell Atkinson and Craig Schaffert.
Abstraction mechanisms in CLU.
Communications of the ACM 20(8):564-576, August, 1977.

[Mitchell 79] James G. Mitchell, William Maybury and Richard Sweet.
Mesa Language Manual
XEROX Palo Alto Research Center, 1979.
Mesa Version 5.0.

[Moon 74] David A. Moon.
MACLISP Reference Manual (revision 0)
Project MAC, Massachusetts Institute of Technology, 1974.
This manual is next to impossible to find—unfortunately, it is the only official manual for MACLISP that was published.

[Morris 79] James B. Morris.
Data abstraction: A static implementation strategy.
In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 1-7. August, 1979.

[Moss 78] John Eliot Blakeslee Moss.
Abstract data types in stack based languages.
Master's thesis, Massachusetts Institute of Technology, February, 1978.
Pages 40-43 contain a discussion of several implementations for returning large objects from functions.

[Naur 63] P. Naur (editor).
Revised report on the algorithmic language ALGOL 60.
Communications of the ACM 6(1):1-17, 1963.

[Nestor 82] John R. Nestor and Margaret A. Beard.
Front end generator system.
In *Computer Science Research Review* (1980-1981, Carnegie-Mellon University), pages 75-92. Department of Computer Science, 1982.
This review is published annually.

[Newcomer 79] Joseph M. Newcomer, Roderic G. G. Cattell, Paul N. Hilfinger, Steven O. Hobbs, Bruce W. Leverett, Andrew H. Reiner, Bruce R. Schatz and William A. Wulf.
PQCC Implementor's Handbook
Carnegie-Mellon University, Computer Science Department, 1979.
PQCC internal document.

[Scheifler 77] Robert W. Scheifler.
An analysis of inline substitution for a structured programming language.
Communications of the ACM 20(9):647-654, September, 1977.

[Shaw 81] Mary Shaw (editor).
ALPHARD: Form and Content.
Springer-Verlag, 1981.

[Standish 67] Thomas A. Standish.
A Data Definition Facility for Programming Languages.
PhD thesis, Carnegie-Mellon University, May, 1967.

[Stroustrup 81] B. Stroustrup.
An Abstract Data Type Facility for the C Language.
Computing Science T. R. 84, Bell Laboratories, August, 1981.

[Touretzky 82] David S. Touretzky.
A summary of MACLISP functions and flags.
4th edition. 1982.
Computer Science Department, Carnegie-Mellon University.

[van Wijngaarden 77]
A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Foster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. G. Fisher.
Revised report on the algorithmic language ALGOL68.
SIGPLAN Notices 12(5):1-70, May, 1977.

[Wegbreit 72] Ben Wegbreit et al.
The ECL Programmer's Manual
Center for Research in Computing Technology, Harvard University, 1972.

[Wegbreit 74] Ben Wegbreit.
The treatment of data types in ELI.
Communications of the ACM 17(5):251-264, May, 1974.

[Wulf 71] W. A. Wulf, D. B. Russell and A. N. Habermann.
BLISS: a language for systems programming.
Communications of the ACM 14(12):780-790, December, 1971.

[Wulf 75] William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs and Charles M. Geschke.
The Design of an Optimizing Compiler.
American Elsevier Publishing Company, Inc., 1975.

[Wulf 80] Wm. A. Wulf.
PQCC: A Machine-Relative Compiler Technology.
Technical Report CMU-CS-80-144, Carnegie-Mellon University, September, 1980.

[Lampson 77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell and G. L. Popek.
Report on the programming language Euclid.
SIGPLAN Notices 12(2), February, 1977.
This issue was devoted to the Euclid language reference manual.

[Lampson 82] Butler W. Lampson.
Fast procedure calls.
In *Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 66-76. March, 1982.

[Leverett 80] Bruce W. Leverett, Roderic G. G. Cattell, Steven O. Hobbs, Joseph M. Newcomer, Andrew H. Reiner, Bruce R. Schatz and William A. Wulf.
An overview of the Production-Quality Compiler-Compiler project.
Computer 13(8):38-49, August, 1980.

[Liskov 77] Barbara Liskov, Alan Snyder, Russell Atkinson and Craig Schaffert.
Abstraction mechanisms in CLU.
Communications of the ACM 20(8):564-576, August, 1977.

[Mitchell 79] James G. Mitchell, William Maybury and Richard Sweet.
Mesa Language Manual
XEROX Palo Alto Research Center, 1979.
Mesa Version 5.0.

[Moon 74] David A. Moon.
MACLISP Reference Manual (revision 0)
Project MAC, Massachusetts Institute of Technology, 1974.
This manual is next to impossible to find—unfortunately, it is the only official manual for MACLISP that was published.

[Morris 79] James B. Morris.
Data abstraction: A static implementation strategy.
In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 1-7. August, 1979.

[Moss 78] John Eliot Blakeslee Moss.
Abstract data types in stack based languages.
Master's thesis, Massachusetts Institute of Technology, February, 1978.
Pages 40-43 contain a discussion of several implementations for returning large objects from functions.

[Naur 63] P. Naur (editor).
Revised report on the algorithmic language ALGOL 60.
Communications of the ACM 6(1):1-17, 1963.

[Nestor 82] John R. Nestor and Margaret A. Beard.
Front end generator system.
In *Computer Science Research Review* (1980-1981, Carnegie-Mellon University), pages 75-92. Department of Computer Science, 1982.
This review is published annually.

[Newcomer 79] Joseph M. Newcomer, Roderic G. G. Cattell, Paul N. Hilfinger, Steven O. Hobbs, Bruce W. Leverett, Andrew H. Reiner, Bruce R. Schatz and William A. Wulf.
PQCC Implementor's Handbook
Carnegie-Mellon University, Computer Science Department, 1979.
PQCC internal document.

[Scheifler 77] Robert W. Scheifler.
An analysis of inline substitution for a structured programming language.
Communications of the ACM 20(9):647-654, September, 1977.

[Shaw 81] Mary Shaw (editor).
ALPHARD: Form and Content.
Springer-Verlag, 1981.

[Standish 67] Thomas A. Standish.
A Data Definition Facility for Programming Languages.
PhD thesis, Carnegie-Mellon University, May, 1967.

[Stroustrup 81] B. Stroustrup.
An Abstract Data Type Facility for the C Language.
Computing Science T. R. 84, Bell Laboratories, August, 1981.

[Touretzky 82] David S. Touretzky.
A summary of MACLISP functions and flags.
4th edition. 1982.
Computer Science Department, Carnegie-Mellon University.

[van Wijngaarden 77] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Foster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. G. Fisher.
Revised report on the algorithmic language ALGOL68.
SIGPLAN Notices 12(5):1-70, May, 1977.

[Wegbreit 72] Ben Wegbreit et al.
The ECL Programmer's Manual
Center for Research in Computing Technology, Harvard University, 1972.

[Wegbreit 74] Ben Wegbreit.
The treatment of data types in EL1.
Communications of the ACM 17(5):251-264, May, 1974.

[Wulf 71] W. A. Wulf, D. B. Russell and A. N. Habermann.
BLISS: a language for systems programming.
Communications of the ACM 14(12):780-790, December, 1971.

[Wulf 75] William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs and Charles M. Geschke.
The Design of an Optimizing Compiler.
American Elsevier Publishing Company, Inc., 1975.

[Wulf 80] Wm. A. Wulf.
PQCC: A Machine-Relative Compiler Technology.
Technical Report CMU-CS-80-144, Carnegie-Mellon University, September, 1980.

Appendix A
Complexity of Inline Substitution

The inline substitution optimization (IS) is crucial to the success of the PR translation scheme (chapter 7). It would be nice to have an efficient algorithm that performs inline substitution to obtain the smallest possible program. Unfortunately, as in many optimization problems, the use of IS for optimum results is intractable. This result forces a compiler to use heuristics to obtain some benefit from IS.

This appendix presents a proof that IS performed for minimum size is NP-hard. Since any problem in NP can be reduced in polynomial time to an NP-hard problem, an NP-hard problem is at least as complex as any problem in NP. Furthermore, an NP-hard problem is at least as difficult as the NP-complete problems, which are believed to be inherently intractable.[62]

The classical manner of demonstrating that a problem is NP-hard is to display a polynomial-time reduction to the problem from a known NP-complete problem. The proof in this chapter uses the NP-complete problem 3-SAT [Cook 71, Garey 79]. The next three sections formalize the inline substitution problem. Section A.1 provides a definition of the language, L, on which inline substitution is carried out. The definition of the language also gives a mechanism for measuring the size of a program and some auxiliary definitions that are used during constant folding (section A.2.2). The transformations that make up inline substitution are given in section A.2. These transformations are analogous to the three optimizations used to perform inline substitution in the compiler (section 7.3). Finally, section A.3 states the inline substitution problem. Last, but not least, the reduction from 3-SAT to IS is presented in section A.4.

[62] The discussion in this appendix assumes the reader is familiar with NP-completeness. For a nice introduction to the theory of NP-completeness the reader should consult the book by Garey and Johnson [Garey 79].

A.1. The Language L

To formalize IS, it is necessary to have a language on which to perform the transformations involved in inline substitution; I designed the language L to serve this purpose. The language L is so simple that it is useless as a programming language, yet too complex for optimum use of IS. A real programming language, which would be a superset of L, will also prevent use of IS for optimal results.

The language L is defined by the attribute grammar presented in figures A-1 and A-2.[63] The synthesized attribute SIZE represents the size of the program. The other synthesized attribute, VALUE, denotes the value of an expression, or Ω if the expression has no value. Values for the attributes SIZE and VALUE are often omitted from productions of the form

$$\langle non\text{-}terminal_1 \rangle \rightarrow \langle non\text{-}terminal_2 \rangle$$

The convention followed is that, for such productions as these,

$$\langle non\text{-}terminal_1 \rangle.\text{SIZE} = \langle non\text{-}terminal_2 \rangle.\text{SIZE}$$
$$\langle non\text{-}terminal_1 \rangle.\text{VALUE} = \langle non\text{-}terminal_2 \rangle.\text{VALUE}$$

In addition, no productions are given for the non-terminals <identifier> and <integer>. The reader may use his favorite definitions for them.

Most of the semantics of L is obvious, but several points need to be mentioned:

- The language is expression oriented: all statements yield a value and any expression may be used like a statement.
- The value of a function call is the value of the last executed statement in the function body.
- The **gen** statement is a notational convenience used to denote code of a given size. The size of a **gen** statement is the integer specified in the statement.
- Parameter passing is by value.
- The value of a <block> is the value of the last statement in the <block>.
- Integer expressions serve as boolean expressions with the convention that an expression is true if and only if its value is 1.

Since the inline substitution problem is concerned with minimizing size, the language must have an associated size measure. A convenient size measure when working with a program represented like a string is the number of tokens in the program. For example, the expression

[63] The definition and use of attribute grammars are beyond the scope of this dissertation. The paper by Knuth [Knuth 68b] gives an introduction to attribute grammars.

```
<program>                → prog
                             <declaration list> ; <statement list>
                           end.
    { <program>.size = 2 + <declaration list>.size + <statement list>.size }

<declaration list>1      → <declaration list>2 ; <declaration>
    { <declaration list>1.size = <declaration list>2.size + <declaration>.size }

<declaration list>       → <declaration>

<declaration>            → func <id>(<formals>) is
                             <statement list>
                           end
    { <declaration>.size = 4 + <formals>.size + <statement list>.size }

<formals>1               → <formals>2 , <id>
    { <formals>1.size = <formals>2.size + 1 }

<formals>                → <id>
    { <formals>.size = 1 }

<statement list>1        → <statement list>2 ; <statement>
    { <statement list>1.size = <statement list>2.size + <statement>.size
      <statement list>1.value = <statement>.value }

<statement list>         → <statement>
```

Figure A-1: Attribute Grammar for L, Part I

```
X + Y - 1
```

could be of size five—one token each for "X", "+", "Y", "-" and "1". This measure is nice because it seems that there should be a direct relation between the number of tokens in an expression and its complexity. This size measure is also machine independent.

A.2. The Transformations

The transformations that form inline substitution are derived from the three optimizations performed on the results of PR (chapter 7). The three transformations, each of which has the same name as its counterpart, are *inline expansion*, *constant folding* and *routine deletion*.

```
<statement>          → <gen>
                     | <expression>

<gen>                → gen <integer>
  { <gen>.size = <integer>.value }

<expression>         → <id>
  { <expression>.size = 1
    <expression>.value = Ω }

<expression>         → <integer>
  { <expression>.size = 1
    <expression>.value = <integer>.value }

<expression>         → <call>
                     | <if>
                     | <block>

<call>               → <id>(<actuals>)
  { <call>.size = 1 + <actuals>.size
    <call>.value = Ω }

<actuals>1           → <actuals>2 , <actual>
  { <actuals>1.size = <actuals>2.size + 1 }

<actual>             → <id> | <integer>
  { <actual>.size = 1 }

<if>                 → if <expression> then
                           <statement list>1
                       else
                           <statement list>2
                       fi
  { <if>.size = 4 + <expression>.size + <statement list>1.size + <statement list>2.size
    <if>.value = Ω }

<block>              → begin <statement list> end
  { <block>.size = 2 + <statement list>.size
    <block>.value = <statement list>.value }
```

Figure A-2: Attribute Grammar for L, Part II

A.2.1. Inline Expansion

The inline expansion transformation replaces a <call> by a modified copy of the body of the target function. The copy is modified by surrounding it by a <block> and by replacing all references to each formal parameter by its corresponding actual parameter. Renaming of variables necessary to prevent conflicts is also performed by this transformation.

For example, inline expansion could be performed on the <call> of F below:

```
func F(N) is
   G(0);
   if N then 1 else N fi
end;

F(Z);
```

to yield

```
func F(N) is                    ! Notice that F remains
   G(0);
   if N then 1 else N fi
end;

begin
   G(0);
   if Z then 1 else Z fi
end;
```

A.2.2. Constant Folding

The *constant folding* transformation may be applied only to <if> expressions. Constant folding may be applied to an <if> expression, whose syntax is

```
if <expression> then
   <statement list>1
else
   <statement list>2
fi
```

if and only if <expression>.VALUE ≠ Ω, in which case the <if> expression may be replaced by

<statement list>1, if <expression>.VALUE=1;

<statement list>2, otherwise.

```
<statement>          → <gen>
                     | <expression>

<gen>                → gen <integer>
   { <gen>.size = <integer>.value }

<expression>         → <id>
   { <expression>.size = 1
     <expression>.value = Ω }

<expression>         → <integer>
   { <expression>.size = 1
     <expression>.value = <integer>.value }

<expression>         → <call>
                     | <if>
                     | <block>

<call>               → <id>(<actuals>)
   { <call>.size = 1 + <actuals>.size
     <call>.value = Ω }

<actuals>1           → <actuals>2 , <actual>
   { <actuals>1.size = <actuals>2.size + 1 }

<actual>             → <id> | <integer>
   { <actual>.size = 1 }

<if>                 → if <expression> then
                           <statement list>1
                        else
                           <statement list>2
                        fi
   { <if>.size = 4 + <expression>.size + <statement list>1.size + <statement list>2.size
     <if>.value = Ω }

<block>              → begin <statement list> end
   { <block>.size = 2 + <statement list>.size
     <block>.value = <statement list>.value }
```

Figure A-2: Attribute Grammar for L, Part II

A.2.1. Inline Expansion

The inline expansion transformation replaces a <call> by a modified copy of the body of the target function. The copy is modified by surrounding it by a <block> and by replacing all references to each formal parameter by its corresponding actual parameter. Renaming of variables necessary to prevent conflicts is also performed by this transformation.

For example, inline expansion could be performed on the <call> of F below:

```
func F(N) is
   G(0);
   if N then 1 else N fi
end;

F(Z);
```

to yield

```
func F(N) is                    ! Notice that F remains
   G(0);
   if N then 1 else N fi
end;

begin
   G(0);
   if Z then 1 else Z fi
end;
```

A.2.2. Constant Folding

The *constant folding* transformation may be applied only to <if> expressions. Constant folding may be applied to an <if> expression, whose syntax is

```
if <expression> then
   <statement list>1
else
   <statement list>2
fi
```

if and only if <expression>.VALUE ≠ Ω, in which case the <if> expression may be replaced by

<statement list>1,	if <expression>.VALUE=1;
<statement list>2,	otherwise.

A.2.3. Routine Deletion

Routine deletion allows the deletion of functions that are not the targets of any <call> expressions.

A.3. Statement of the Problem

It is now possible to state the IS problem.

> **Definition 1:** Given a program *P* derived from the nonterminal <program> in the grammar in figures A-1 and A-2, the *inline substitution optimization problem* is to find the minimal size program that may be obtained from *P* by applying the transformations listed in section A.2. The minimal size program is that program with the minimal value of the attribute <program>.SIZE over all possible transformed programs.

It is useful to formulate a problem that is to be used in a reduction from an NP-complete problem as a *decision problem*: a problem that can be answered *yes* or *no*. Although the statement above describes the problem we had in mind, it must be recast as follows:

> **Definition 2:** An instance of the *inline substitution decision problem* is a tuple (P, S), where *P* is a program derived from the nonterminal <program> in the grammar in figures A-1 and A-2, and $S>0$. The decision procedure answers *yes* if and only if there is a program *P'* such that <program>.SIZE$\leq S$ and *P'* is obtainable from *P* by applying the transformations listed in section A.2.

The decision problem is no harder than the optimization problem. If the decision problem is NP-hard, then so is the optimization problem.

A.4. The Proof

The theorem to be proved can be stated as follows:

> **Theorem 3:** The inline substitution decision problem is NP-hard.

This is proved by displaying a polynomial-time reduction from the problem 3-SAT, which is defined in section A.4.1. Section A.4.2 gives the reader a feel for the approach taken by the reduction. The heart of the reduction is presented in sections A.4.3 and A.4.4; the proof is brought to a close in section A.4.5.

A.4.1. 3-SAT

The problem 3-SAT is defined in this section after some auxiliary definitions. In the following definitions let $U = \{u_1, u_2, ..., u_m\}$ be a set of *variables*.

Definitions 4: A *truth assignment* for U is a function $t : U \rightarrow \{T, F\}$.

A *literal* over U is "u_i" or "$\overline{u_i}$", where $u_i \in U$. (The double quotes frequently will be omitted from the denotation of a literal.)

A truth assignment t may be extended to included literals in its domain by defining

$$t("u_i") = t(u_i)$$
$$t("\overline{u_i}") = \neg t(u_i)$$

A *clause* over U is a multi-set of literals over U.

A truth assignment t for U *satisfies* a clause c over U if $t(l)=T$ for some $l \in c$.

A truth assignment for U is a *satisfying assignment* for a multi-set of clauses over U if it satisfies simultaneously all of the clauses.

Definition 5: Given a set U of variables and C a multi-set of clauses over U, such that $|c|=3$ for all $c \in C$, the *3-SAT problem* is to determine if there is a satisfying assignment for C.

Theorem 6: 3-SAT is NP-complete.

Proof: By reduction from satisfiability (see the book by Garey and Johnson [Garey 79] for a proof).

A.4.2. The General Idea

The reduction takes an arbitrary instance of the 3-SAT problem and constructs an instance of the inline substitution decision problem. If and only if the inline substitution instance were solved it would yield a solution to the 3-SAT problem. Because the construction can be done in polynomial time, a solution to the inline substitution decision problem would yield a polynomial-time solution to 3-SAT. This proves that the inline substitution decision problem is at least as hard as the NP-complete problems, of which 3-SAT is a representative.

To simulate an instance of 3-SAT the construction will fulfill the following requirements:

- a correspondence between literals over U and program entities;
- a correspondence between an assignment of T to a variable and a transformation on the program;

- a way of ensuring that if $t(\text{"}u_i\text{"}) = T$ then $t(\text{"}\overline{u_i}\text{"}) = F$;
- a correspondence between the set of clauses and program entities;
- a correspondence between the satisfaction of a clause and a transformation on the program;
- a correspondence between the satisfiability of the instance of 3-SAT and an answer of *yes* to the instance of IS;
- a correspondence between the unsatisfiability of the instance of 3-SAT and an answer of *no* to the instance of IS.

Given an instance of 3-SAT, as follows:

$$(U, C) = (\{u_1, u_2, \ldots, u_n\}, \{c_1, c_2, \ldots, c_m\})$$

an instance, (P, S), of the inline substitution decision program will be constructed with the following program for P:

```
prog
    func u1(N) is ... end;
    func ū1(N) is ... end;
    . . .
    func un(N) is ... end;
    func ūn(N) is ... end;
    func C1(N1, N2, ..., N29) is ... end;
    . . .
    func Cm(N1, N2, ..., N29) is ... end;
    C1(1, 1, ..., 1); C1(Z, 1, ..., 1);
    . . .
    Cm(1, 1, ..., 1); Cm(Z, 1, ..., 1)
end.
```

At this level the reader can see that clauses and literals are mapped to functions. The program is constructed so that if the instance of 3-SAT is satisfiable, all of the functions C_i will be deleted. The size S is chosen so that the transformed program will have a size no larger than S if and only if all of the C functions are deleted.

A.4.3. Literals

For each variable $u_i \in U$, two functions are created: u_i and $\overline{u}_i$. These functions will be designed so that the assignment $t(u_i)=T$ will correspond to all calls of u_i being expanded inline. Similarly, the assignment $t(u_i)=F$ will correspond to all calls of $\overline{u}_i$ being expanded inline. To ensure consistency, it must be arranged so that a solution to the instance of inline substitution cannot have both all calls of u_i and all calls of $\overline{u}_i$ expanded inline.

The format of the functions corresponding to u_i and $\overline{u}_i$ is illustrated in figure A-3. The notations $\#u_i$ and $\#\overline{u}_i$ represent the numbers of occurrences of the literals u_i and $\overline{u}_i$ in the instance of 3-SAT. The expressions $\alpha[\#u_i, \#\overline{u}_i]$ and $\overline{\alpha}[\#u_i, \#\overline{u}_i]$ denote values dependent on only $\#u_i$ and $\#\overline{u}_i$. These values are developed below.

```
func u_i(N) is
    if N then
        ū_i(1);
        1
    else
        gen α[#u_i, #ū_i]
    fi
end;

func ū_i(N) is
    if N then
        u_i(1);
        1
    else
        gen ᾱ[#u_i, #ū_i]
    fi
end;
```

Figure A-3: Mapping from Variables in U to Functions

Each u_i function is constructed so that any minimal inline substitution can expand either all calls to u_i or all calls to $\overline{u}_i$, not both.

Initially there are $\#u_i+1$ calls of u_i, all of the form $u_i(1)$. Of these calls, $\#u_i$ are embedded in the C functions and actually look like $u_i(N_1)$ but constant propagation from the expansions of the calls $C_i(1, 1, \ldots, 1)$ creates the appropriate calls.

So the values of α and $\overline{\alpha}$ are chosen to satisfy these inequalities.[64]

A.4.4. Clauses

Each clause $C_i = \{l_{i,1}, l_{i,2}, l_{i,3}\}$ will generate a function of the form

```
func C_i(N_1, N_2, ..., N_29) is
    if l_{1,1}(N_1) then
        gen 0
    else
        gen β[n]
    fi;
    if l_{1,2}(N_1) then
        gen 0
    else
        gen β[n]
    fi;
    if l_{1,3}(N_1) then
        gen 0
    else
        gen β[n]
    fi
end;
```

The expression $\beta[n]$ represents an integer that is dependent on the value of n, the number of variables in U. The value of $\beta[n]$ is developed in the remainder of the proof.

A call $C_i(1, 1, \ldots, 1)$ is of size 30 (1 for C_i plus 1 for each of 29 actuals). An inline expansion of such a call is

---

[64] Note that these inequalities imply that

$$(\#u_1 + \#\overline{u_1}) > 4$$

which restricts the instances of 3-SAT to which the reduction applies. But an arbitrary instance of 3-SAT can be padded with clauses to achieve the necessary number of literals. For example, if $(\#u_1 + \#\overline{u_1}) \leq 4$, then the following clauses can be added:

$$\{u_1, \overline{u_1}, u_1\}, \{\overline{u_1}, u_1, \overline{u_1}\}$$

Because these two new clauses are trivially satisfiable, it is a simple matter to show that this modified form of 3-SAT is equivalent to the original problem.

Each invocation, $u_i(1)$ is of size 2. Thus, the total size of the invocations of u_i and the body of u_i is

$$2(\#u_i+1) + (13 + \alpha[\#u_i, \#\overline{u}_i])$$

An inline expansion of a call $u_i(1)$ is

```
begin
   if 1 then
      ū_i(1);
      1
   else
      gen α[#u_i, #ū_i]
   fi
end
```

Applying constant folding to this yields

```
begin ū_i(1); 1 end
```

which has a size of five. To be worth expanding all calls of u_i it must be the case that

$$5(\#u_i+1) < 2(\#u_i+1) + 13 + \alpha[\#u_i, \#\overline{u}_i]$$

or

$$\alpha[\#u_i, \#\overline{u}_i] > 3(\#u_i+1) - 13$$

Originally there were $\#\overline{u}_i$ calls of $\overline{u}_i$. If all calls of u_i are expanded, however there will be $\#u_i$ new calls of $\overline{u}_i$ created. The deletion of the declaration of function u_i removes one call. So, after expansion of all calls of u_i and deletion of the body, there will be $\#\overline{u}_i + \#u_i$ calls of function $\overline{u}_i$. To ensure that it is not worth expanding the calls of $\overline{u}_i$ it must be the case that

$$5(\#\overline{u}_i+\#u_i) > 2(\#\overline{u}_i+\#u_i) + 13 + \overline{\alpha}[\#u_i, \#\overline{u}_i]$$

or

$$\overline{\alpha}[\#u_i, \#\overline{u}_i] < 3(\#\overline{u}_i+\#u_i) - 13$$

Continuing this argument obtains

$$3(\#u_i+1)-13 < \alpha[\#u_i, \#\overline{u}_i] < 3(\#u_i+\#\overline{u}_i)-13 \qquad \text{(A.1)}$$

$$3(\#\overline{u}_i+1)-13 < \overline{\alpha}[\#u_i, \#\overline{u}_i] < 3(\#u_i+\#\overline{u}_i)-13 \qquad \text{(A.2)}$$

arranged so that if any one call within C_i is expanded, then the invocation $C_i(1, 1, \ldots, 1)$ will be expanded inline.

Once all of the calls of the form $C_i(1, 1, \ldots, 1)$ have been expanded inline, each C_i will be called just once. Hence, all of the calls $C_i(Z, 1, \ldots, 1)$ will be expanded inline, allowing the deletion of the declarations of the C_i.

If the instance of 3-SAT is satisfiable, all of the C functions have been deleted. The maximum size of such a program is

$$2 + 13n + \sum_{1 \le i \le n} \text{Max}\{\alpha[\#u_i, \#\overline{u}_i], \overline{\alpha}[\#u_i, \#\overline{u}_i]\} + m(23 + 3\beta[n])$$

This is the value chosen for S.

It must be ensured that if the instance of 3-SAT is unsatisfiable, and not all of the C functions are deleted, the size of the program will be greater than S. The minimum size of such a program is obtained when all of the functions are deleted except for the following:

- one C function, C_j, whose corresponding clause has only a single variable, u_k;
- the two U functions u_k and $\overline{u}_k$.

The size of this program is

$$82 + \alpha[\#u_k, \#\overline{u}_k] + \overline{\alpha}[\#u_k, \#\overline{u}_k] + 3\beta[n] + m(23 + 3\beta[n])$$

So $\beta[n]$ is chosen so that

$$82 + \min_{u \in U}\{\alpha[\#u, \#\overline{u}] + \overline{\alpha}[\#u, \#\overline{u}]\} + 3\beta[n] + m(23 + 3\beta[n]) >$$
$$2 + 13n + \sum_{1 \le i \le n} \text{Max}\{\alpha[\#u_i, \#\overline{u}_i], \overline{\alpha}[\#u_i, \#\overline{u}_i]\} + m(23 + 3\beta[n])$$

or, rearranging,

$$3\beta[n] > 13n + \sum_{1 \le i \le n} \text{Max}\{\alpha[\#u_i, \#\overline{u}_i], \overline{\alpha}[\#u_i, \#\overline{u}_i]\} - 80 - \min_{u \in U}\{\alpha[\#u, \#\overline{u}] + \overline{\alpha}[\#u, \#\overline{u}]\}$$

Looking at equations (A.1) and (A.2), we note that

$$\text{Max}\{\alpha[\#u_i, \#\overline{u}_i], \overline{\alpha}[\#u_i, \#\overline{u}_i]\} = 3(\#u_i + \#\overline{u}_i) - 14$$

Using this, and the fact that the minimum values of α and $\overline{\alpha}$ are 0, we obtain

$$3\beta[n] > 13n + \sum_{1 \le i \le n} [3(\#u_i + \#\overline{u}_i) - 14] - 80$$

Simplifying this and eliminating negative terms gives the following constraint for the value of $\beta[n]$:

```
begin
  if l1,1(1) then
     gen 0
  else
     gen β[n]
  fi;
  if l1,2(1) then
     gen 0
  else
     gen β[n]
  fi;
  if l1,3(1) then
     gen 0
  else
     gen β[n]
  fi
end
```

The value $\beta[n]$ is chosen so that this expansion has a size larger than 30, which makes the expansion undesirable. This be can accomplished by letting $\beta[n]=4$, but later the reader will see that there is another use for $\beta[n]$ that further constrains its value.

If any of the calls $l_{j,k}(1)$ are expanded inline, then the size of the inline expansion of $C_i(1, 1, \ldots, 1)$ shrinks to at most 28. This will allow the call $C_i(1, 1, \ldots, 1)$ to be expanded inline. The expansion corresponds to the fact that if any literal within a clause is true, then the whole clause is true.

This is the reason for the 29 parameters to the C_i functions. For the trick described above to work, an invocation of C_i must have a size greater than 28 but less than $\beta[n]$. An invocation with 29 parameters provides a size of 30.

A.4.5. Putting it All Together

The construction is now complete and it remains to show that

1. If an instance of 3-SAT is satisfiable, then there is a program obtainable by inline substitution from program P with size no greater than S.
2. If there is a program obtainable from P with size no greater than S, then the instance of 3-SAT is satisfiable.

For the first part, notice that if the instance of 3-SAT is satisfiable, then there is a truth assignment such that in each clause at least one of the literals is true. This implies that there is a way to expand the calls $l_{i,j}(N_1)$ such that at least one call is expanded in each C_i function. Things have been

```
prog
    func f1(X) is ... end;

    func f2(X) is
        f1(1);
        f1(2);
        . . .
        f1(N)
    end;
    . . .

    func fM(X) is
        fM-1(1);
        fM-1(2);
        . . .
        fM-1(N)
    end
end.
```

It may be that the program obtainable from this program with size less than or equal to S requires that all inline expansions be performed. There are a maximum of N^{M-1} inline expansions and this number is not polynomial in the length of the input, which is $O(NM)$.

Of course, this is not a proof that the decision problem is not in NP. It simply shows that the obvious method will not work. But my intuition, which was obtained by working on the problem for some time, is that the problem is not in NP and, hence, not NP-complete.

$$\beta[n] > \sum_{1 \le i \le n} (\#u_i + \#\overline{u}_i)$$

It remains to show is that if there is a program obtainable from P with size no greater than S, then the instance of 3-SAT is satisfiable. If there is a program P' with size no greater than S, then all the declarations of C_i have been deleted because of the construction of $\beta[n]$. It must be that all of the calls $C_i(1, 1, \ldots, 1)$ have been expanded inline. This is only possible if the body of C_i has been reduced in size. Furthermore, this only occurs if one of the calls to a literal in C_i has been expanded inline. The truth function satisfying the instance of 3-SAT can be discovered by noting for each variable that:

- If all calls of the form $u_i(N)$ have been inline expanded, then $t(u_i)=T$.
- If all calls of the form $\overline{u}_i(N)$ have been inline expanded, then $t(u_i)=F$.

□

A.5. Inline Substitution and NP-completeness

It would be tidy to show that the inline substitution decision problem is NP-complete. To prove a problem is NP-complete comprises two parts: prove the problem is NP-hard, and show the problem is in the class NP.[65] This chapter proved that the decision problem is NP-hard, and this is generally the difficult part of a proof of NP-completeness.

The demonstration that a problem is in NP is usually trivial because it is only necessary to show that the following can be done in non-deterministic time polynomial in the length of the input:

1. generate a candidate solution program P' that is obtained from P by applying the transformations;
2. compute the size of P';
3. compare the size of P' to S.

Excepting for step 1, it is clear that this can be done in polynomial time. The problem with step 1 is that producing the solution may require performing a number of transformations that is exponential in the length of the input program P. Consider the program outlined below:

---

[65] NP is the class of problems that can be solved in non-deterministic time polynomial in the length of their input [Garey 79].

Appendix B
Package STANDARD

```
   generic
      type Index is (<>);
      type Element is limited private;
      type Arr is array(Index range <>) of Element;
   function FIRST(A : in Arr) return Index;
        pragma Predefined(FIRST, ARRAY_FIRST);

   generic
      type Index is (<>);
      type Element is limited private;
      type Arr is array(Index range <>) of Element;
   function LAST(A : in Arr) return Index;
        pragma Predefined(LAST, ARRAY_LAST);

   generic
      type Index is (<>);
      type Element is limited private;
      type Arr is array(Index range <>) of Element;
   selector SUBSCRIPT(A : in out Arr; I : in Index) return Ele
ment;
        pragma Predefined(SUBSCRIPT, SUBSCRIPT);

   --
   -- END OF INTERNAL subprograms
   --
```

```
-- Commented "not" out: 6/28/82, J. Rosenberg
--   function "not"(X : in Boolean) return Boolean;
--      pragma Predefined("not", Boolean_Not);

   function "and"(X, Y : in Boolean) Return Boolean;
        pragma Predefined("and", Boolean_And);
    function "or"(X, Y : in Boolean) Return Boolean;
        pragma Predefined("or", Boolean_Or);

    type Integer;          -- No way to say it
        pragma Predefined(Integer, Integer);

    function "+"(X, Y : in Integer) return Integer;
        pragma Predefined("+", Integer_Plus);
    function "-"(X, Y : in Integer) return Integer;
        pragma Predefined("-", Integer_Minus);
    function "*"(X, Y : in Integer) return Integer;
        pragma Predefined("*", Integer_Times);
    function "/"(X, Y : in Integer) return Integer;
        pragma Predefined("/", Integer_Div);

-- Commented out all FLOAT stuff: 6/27/82, J. Rosenberg
--   type Float;                  -- No way to say it
--      pragma Predefined(Float, Float);

--   function "+"(X, Y : in Float) return Float;
--      pragma Predefined("+", Float_Plus);
--   function "-"(X, Y : in Float) return Float;
--      pragma Predefined("-", Float_Minus);
--   function "*"(X, Y : in Float) return Float;
--      pragma Predefined("*", Float_Times);
--   function "/"(X, Y : in Float) return Float;
--      pragma Predefined("/", Float_Div);
```

```
------------------------------------------------------------
--
--      Package STANDARD for my Ada subset.
--
-- Created: 12/8/81, J. Rosenberg
--
------------------------------------------------------------

package Standard is

   type Boolean is (False, True);   -- Boolean must be here
        pragma Predefined(False, False);
        pragma Predefined(True, True);
        pragma Predefined(Boolean, Boolean);

   --
   -- NOTE: The following group of subprograms are for the
   -- compiler's internal use only.  Results not guaranteed
   -- if called by a user.
   --

   generic
      type T is limited private;

   procedure ASSIGN(X : out T; Y : in T);
        pragma Predefined(ASSIGN, ASSIGN);

   generic
      type T is limited private;
   function EQUAL(X, Y : in T) return Boolean;
        pragma Predefined(EQUAL, EQUAL);

   generic
      type T is limited private;
   function NOT_EQUAL(X, Y : in T) return Boolean;
        pragma Predefined(NOT_EQUAL, NOT_EQUAL);

   generic
      type T is (<>);
   function LESS_THAN(X, Y : in T) return Boolean;
        pragma Predefined(LESS_THAN, LESS_THAN);

   generic
        type T is (<>);
   function LESS_THAN_OR_EQUAL(X, Y : in T) return Boolean;
        pragma Predefined(LESS_THAN_OR_EQUAL,
                          LESS_THAN_OR_EQUAL);

   generic
      type T is (<>);
   function GREATER_THAN(X, Y : in T) return Boolean;
        pragma Predefined(GREATER_THAN, GREATER_THAN);

   generic
      type T is (<>);
   function GREATER_THAN_OR_EQUAL(X, Y : in T) return Boolean;
        pragma Predefined(GREATER_THAN_OR_EQUAL,
                          GREATER_THAN_OR_EQUAL);

   generic
      type T is (<>);
   function PRED(X : in T) return T;
        pragma Predefined(PRED, PRED);

   generic
      type T is (<>);
   function SUCC(X : in T) return T;
        pragma Predefined(SUCC, SUCC);
```

Appendix C
The Run-time System

This appendix contains the listing of the run-time system routines that perform I/O and object creation. The routines are contained in the two BLISS-36 modules IO and ALLOC. The third run-time module, not displayed here, performs dynamic space allocation and deallocation. This module is part of the BLISS-36 library on the PDP-20 at Carnegie-Mellon University.

```
   type Character is (nul, soh, stx, etx, eot, enq, ack, bel,
                      bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
                      dle, dc1, dc2, dc3, dc4, nak, syn, etb,
                      can, em,  sub, esc, fs,  gs,  rs,  us,
                      ' ', '!', '"', '#', '$', '%', '&', ''',
                      '(', ')', '*', '+', ',', '-', '.', '/',
                      '0', '1', '2', '3', '4', '5', '6', '7',
                      '8', '9', ':', ';', '<', '=', '>', '?',
                      '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
                      'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
                      'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
                      'X', 'Y', 'Z', '[', '\', ']', '^', '_',
                      '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
                      'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
                      'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
                      'x', 'y', 'z', '{', '|', '}', '~', del
                     );
        pragma Predefined(Character, Character);

   type String is array(Integer range <>) of Character;
        pragma Predefined(String, String);

end Standard;
```

```
%(

                        <HOOK.RUNTIME>IO.B36

    I/O for Ada compilations.

    Created: 4/20/82, J. Rosenberg

)%

module IO =
begin

require '<HOOK.RUNTIME>ARRAY.REQ';

builtin JSYS;

literal
        TTY     = %O'777777',
        BIN     = %O'60',
        BOUT    = %O'61';


literal
        Minus   = %C'-',
        Zero    = %C'0',
        CR      = %O'15',
        LF      = %O'12';

routine Out(C) : novalue =
begin
    register AC1=1, AC2=2;
    AC1 = TTY;
    AC2 = .C;
    JSYS(0, BOUT)
end;

routine In =
begin
    register AC1=1, AC2=2;
    AC1 = TTY;
    JSYS(0, BIN);
    return .AC2
end;
```

```
global routine Int_Put(N) : novalue =
begin
    routine XN(M) : novalue =
    begin
        local R;
        if .M eql 0 then return;
        R = .M mod 10;
        M = .M / 10;
        XN(.M);
        Out(.R+Zero)
    end;
    if .N eql 0 then begin
        Out(Zero);
        return
    end;
    if .N lss 0 then Out(Minus);
    XN(abs(.N))
end;

macro Digit(Char) = (Char geq %C'0' and Char leq %C'9') %;

global routine Int_Get(N) : novalue =
begin
    local C, Value, Neg;
    Value = 0;
    C = In();
    while (not Digit(.C)) and (.C neq Minus) do C = In();
    Neg = (.C eql Minus);
    if .Neg then C = In();
    while Digit(.C) do begin
        Value = .Value*10 + (.C-Zero);
        C = In()
    end;
    if .C eql CR then In(); ! Read LF -- thanks to Andy Hisgen
    .N = (if .Neg then -.Value else .Value)
end;

global routine Str_Put(S) : novalue =
begin
    bind Orig = (Vector[.S, DV_ORIG_FIELD] +
                 .Vector[.S, DV_ORIG_FIELD]) : Vector;
    incr I from .Vector[.S, DV_LB_FIELD] to
                .Vector[.S, DV_UB_FIELD] do
        Out(Orig[.I])
end;
```

```
%(

                        <HOOK.RUNTIME>ALLOC.B36

    Object allocator.

    Created: 7/10/82, J. Rosenberg

)%

module Allocator =
begin

require '<HOOK.RUNTIME>ARRAY.REQ';
require '<HOOK.RUNTIME>DESCS.REQ';

forward routine Array_Alloc : novalue;

external routine GetSpace;

! Object allocator
global routine Alloc(Object, Descriptor) : novalue =
begin
    Map Descriptor : ref Vector;

    if .Descriptor[DESC_KIND_FIELD] = $ARRAY then
       Array_Alloc(.Object, .Descriptor)

end %( Alloc )%;

routine Array_Alloc(Object, Descriptor) : novalue =
begin
    Map Object : ref Vector, Descriptor : ref Vector;
    local LB, UB, Size, Base;

    LB = .Descriptor[DESC_LB_FIELD];
    UB = .Descriptor[DESC_UB_FIELD];
    Size = .UB - .LB + 1;
    Base = GetSpace(.Size);

    ! Initialize virtual 0-origin
    Object[DV_ORIG_FIELD] = .Base - Object[DV_ORIG_FIELD]-LB;

    ! Initialize base of storage
    Object[DV_BASE_FIELD] = .Base - Object[DV_BASE_FIELD];

    ! Initialize size field
    Object[DV_SIZE_FIELD] = .Size;

    ! Initialize bounds
    Object[DV_LB_FIELD] = .LB;
    Object[DV_UB_FIELD] = .UB

end %( Array_Alloc )%;

end eludom
```

Appendix D
The Size Estimation Table

This table contains the estimated size for each IL node kind that may appear as an expression. All nodes corresponding to declarations are handled by routines. The values were used in the estimation routine of the OPT phase, as described in section 8.2.4.

```
global routine Char_Put(C) : novalue =
begin
    Out(.C)
end;

global routine Char_Get(C) : novalue =
begin
    .C = In()
end;

global routine New_Line(N) : novalue =
begin
    decr I from .N-1 to 0 do begin
        Out(CR);
        Out(LF)
    end
end;

end eludom
```

Appendix E
Compiler Phase Timings

Tables E-1 and E-2 contain the compilation times, in seconds, for each phase of the experimental compiler for each test program. A time of 0.0 means that the phase took less than 100 milliseconds.

Node Kind	Size
routine, variable, descriptor	1
leave, **return**	1
while	2
←	0
call	1
=, ≠, >, ≥, <, ≤	1
*, /, +, -	1.25
and, **or**	2
.	0.5
ref	1
subscript	1
integer literal	0.5
string literal	1
descriptor kind	1
if	1
kind	1.5
case	1.5

Table D-1: Size Estimation Table

	Phase				
Program	INFO	TRANSLATE	OPT	CODE	Total
Arrtst	0.0	9.93	14.84	2.27	27.04
BLT	0.0	0.62	0.22	0.32	1.16
Calc	0.0	7.95	20.38	5.1	33.43
Comand	0.0	27.69	54.33	8.96	90.98
Listl	0.0	11.65	65.63	9.0	86.28
Sortl	0.0	1.03	0.64	0.52	2.19
Sort2	0.0	1.58	0.75	0.82	3.15
Sort3	0.0	5.02	2.9	1.68	9.6
Sort5	0.0	4.96	2.8	1.2	8.96
Sort6	0.0	11.69	57.39	3.36	72.44
Sort7	0.0	24.03	20.12	4.19	48.34
Syml	0.0	2.62	2.73	1.47	6.82

Table E-1: Phase Timings for GE

	Phase				
Program	INFO	TRANSLATE	OPT	CODE	Total
Arrtst	0.27	9.28	34.47	1.87	45.89
BLT	0.1	1.77	2.33	0.43	4.63
Calc	0.36	9.69	71.27	5.46	86.78
Comand	0.32	10.74	71.37	2.79	85.22
Listl	0.27	7.45	37.18	3.51	48.41
Sortl	0.13	2.3	5.08	0.54	8.05
Sort2	0.14	2.22	7.59	0.84	10.79
Sort3	0.18	4.45	11.49	1.12	17.24
Sort5	0.18	4.43	11.41	1.06	17.08
Sort6	0.24	5.57	41.15	2.23	49.19
Sort7	0.22	5.29	23.1	1.84	30.45
Syml	0.19	2.98	17.89	2.43	23.49

Table E-2: Phase Timings for PR

Carnegie-Mellon University
Department of Computer Science
Schenley Park
Pittsburgh, Pennsylvania 15213