Nirenberg, Robert Michael

THE MAPPING OF THE REFAL MACHINE ONTO A VON NEUMANN MACHINE

City University of New York PH.D. 1983

THE MAPPING OF THE REFAL MACHINE ONTO A VON NEUMANN MACHINE

by

Robert Michael Nirenberg

A dissertation submitted to the Graduate Faculty in Engineering in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1983

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Aug. 9, 1983
date

Professor Valentin F. Turchin
Chairman of Examining Committee

Aug 15, 1983
date

Professor Paul R. Harmel
Executive Officer

Professor Michael Anshel

Doctor William T. Gewirtz

Professor Stanley Habib

Professor Avgustin Tuzhilin

Supervisory Committee

The City University of New York

ABSTRACT

THE MAPPING OF THE REFAL MACHINE ONTO A VON NEUMANN MACHINE

by

Robert Michael Firenberg

Advisor: Professor Valentin F. Turchin

A supercompiler system based on the language Refal has been implemented at the City University of New York's University Computer Center. The supercompiler takes as its input a program in Refal, and produces an optimized program as its result. The output is in a representation called a graph of states of the abstract Refal machine. In order to implement a supercompiler system on a real computer, therefore, it is necessary to map the resulting graph of states onto an implemented programming language. To that end, a mapping scheme has been devised, which allows for further optimizations to the program represented by the graph of states. An intermediate language, called LISA, has been developed, in order to facilitate manipulations of data types and structures in the abstract program, before code generation takes place.

A code generation scheme, based on a method of template-matching has been implemented. Also, the background environment necessary to execute a mapped program in Pascal has been implemented, so that a complete package now exists for mapping graphs of states onto a sequential programming language. The code generation scheme is retargettable, and optimizations to the mapped graph of states have been identified, and some implemented.

ACKNOWLEDGEMENTS

The author is greatly indebted to many people for their assistance and encouragement in completing this work. Thanks are due to Guillermo Irisarri and Mayer Sasson for their encouragement during the initial phases of the research. Jim Piccarello and Igor Zomb provided support in interfacing with the supercompiler, and with the operating systems at CUNY/UCC. Heartfelt thanks are also given to Professors Michael Anshel and Stanley Habib, and to Dean Paul Karmel, for interfacing with the University and the real world. Professor Valentin F. Turchin provided the genius needed to conceive this work, and without his guidance it would not have come to pass. Finally, warm thanks to my family, whose support and encouragement were unflagging.

This work has been supported in part by National Science Foundation Grant #MCS-8007565.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

A Supercompiler system based on the language Refal has been proposed in [1]. A working supercompiler system has been implemented on the IBM/370 system at the City University of New York University Computer Center[1], using an interpretive Refal system [2], and is now undergoing testing and further development. The present supercompiler system takes a Refal program as its input, and creates a transformed Refal program as its output. Progress has been made in the implementation of input language translators using interpretive definitions of those languages [4:5-8], allowing the transformation of programs written in LISP and Pascal to be optimized by the supercompiler. In order to extend the capabilities of the supercompiler it becomes necessary to map the output of the supercompiler onto other, more "conventional" programming languages. The purpose of this work is to show how such a mapping can be made, and to open a large field of investigation for further development of the Refal Supercompiler System.

A concise description of the Supercompiler project may be found in [7]:

---

[1] This work has been supported by National Science Foundation grant #MCS-8007565, initiated in Sept. 1980, and renewed in Sept. 1982.

The programming system to be described in this proposal aims to facilitate the creation and implementation of specialized algorithmical languages at low expense, and to perform by computer a great deal of work on optimization of algorithms, and even on the algorithmization process itself, which is now performed by man. We hope to create a programming system, in which the introduction of a new special language, or a hierarchy of languages, ad hoc for each large-scale programming problem is just as natural and as practicable as is the introduction of an ad hoc hierarchy of procedures when we are programming, say, in ALGOL-60. With this system, the programmer will have to formulate only the definition of his problem, its mathematical model, without bothering about the details of algorithmic efficiency and data structures in the real computer.

The importance of this aim and the fact that the present situation in computer art and science is unsatisfactory is fully recognized now. We quote one of the leading computer scientists in this country who starts his latest paper in the Communications of the ACM with these words:

> Computer programming today is in a state of crisis (or, more optimistically, a state of creative ferment). There is a growing recognition that the available programming languages are not adequate for building a computer system. Of course, as any first year student of computation theory knows, they are logically sufficient. But they do not deal with the problems we face in the day-to-day work of programming. We become swamped by the complexity of large systems, lost in code written by others, and mystified by the behavior of our almost debugged systems. When we look to the integrated multiprocessor systems that will soon dominate computing, the situation is even worse. This crisis in software production is far greater (in overall magnitude at least) than the situation of the early 50's that led to

> the development of high level languages to relieve the burden of coding. The problems are harder to solve, and the costs of not solving them are in the hundreds of millions.[2]

The solution to the problem given by the author of this excerpt is: higher level programming systems. There is a lot of work being done in this field in the USA, Europe, and the Soviet Union. The distinctive feature of our approach is that we use a special metalanguage REFAL which is a successful compromise between the simplicity necessary to formulate effective rules of transformation of the algorithms defined in this language, and the sophistication necessary to define complicated algorithms in a manageable form. This has allowed essential progress towards the goal of the application of the theory of equivalence transformations of algorithms to the needs of systems programming.

The proposed system will provide a general mechanism which allows one to define an expansible hierarchical system of languages. For this purpose, a metalanguage M must be specified which would allow its user to define each new language Ln in terms of languages of lower levels: Ln-1, Ln-2, etc. Also, a ground-level language L0 must be defined, and must be such that all the languages of the hierarchy can ultimately be expressed in L0. There are two ways to formally define a new language Ln in terms of a lower-level language Lk: in a translation mode, in which one specifies the manner in which a text in Ln is transformed into a text in Lk; and in an interpretation mode, i.e., specifying the process of execution of a text in Ln in terms of the language Lk. In accordance with the distinction that appears here, we can distinguish between two kinds of expansible systems. Systems of the first kind, translation-mode, have machine (assembler) language as the ground-level language L0: such systems may be called macrocode systems, and are widespread now. The metalanguage in this case

---

[2] From [8].

is the language of macro-definitions. Although very useful, these systems do not unburden the programmer, but only put him in a better environment. The system which we propose to build is of the second kind. Here, new languages are defined in interpretation mode, and L0 is then a very elementary language which includes only basic operations on symbol strings and numbers. The description of a language and of an algorithm in that language then takes the shape of a "description of the meaning" rather than a final definition of the program to be executed on the computer. But then one only needs an algorithm -- which may be called a supercompiler -- which would translate this multi-level semantic problem definition into an efficient program for a real computer.

An important feature of our project is that the metalanguage M in which new languages are defined, the ground-level language L0, and the language in which the supercompler is written -- all three are the same language, REFAL. As shown in [9], this has the crucial advantage that only one supercompiler Cp from the metalanguage M into the language of an object machine M0 is needed for all languages Ln of all levels. To attain this surprising economy, we use a method, whose essence is the self-application of Cp. The result: by writing a simple "metasystem-transition formula" and pushing a button one can obtain a program for M0 which can be either

1. an efficient, compiled program P0 which is the translation of a program P written in Ln (if P is given); or

2. an interpreter for the language Ln, which takes a program P in Ln and input data D and executes P on M0 in accordance with the interpretive definition of Ln; or

3. a compiler for the language Ln, which takes a program P in Ln and translates it into an efficient program P0 for M0; or

4. a compiler compiler (if the definition of Ln is not given), which takes the definition of a new language in M and produces a compiler for it.

For the approach we have sketched to be feasible, the following three requirements must be met by the metalanguage M:

1. It must be universal -- not only in the sense that any algorithmic transformation can be described in it, but also in the sense that it must not be aimed at any special system of concepts tied to a particular object language; this makes it possible for one and the same metalanguage to be used with equal success in describing whatever language we may invent, and at all levels of conceptual hierarchy. In programming terms, the metalanguage must have a broad symbol manipulation orientation.

2. The metalanguage must be convenient to use; in particular, a text in it must look not like an intricate program, which in some mysterious way performs algorithms written in the language to be described, but rather must be a semantical description of this language, consisting of a set of sentences which define the meaning of its concepts. Thus, the metalanguage must be essentially a production language, rather than an instruction/statement language of more familiar form.

3. The language must be <u>minimal</u> in the sense that the <u>defining machine</u> which executes algorithms written in this language must be simple enough for the rules dealing with algorithms to be formulated effectively. Otherwise there will be little hope of creating a supercompiler which could perform really deep optimizing transformations of algorithms. However, this requirement may come into conflict with the requirement of convenience. A simple Turing machine or Markov algorithm languages are simple enough to be used for purposes of theory, but certainly are impractical for writing complicated algorithms. A language which deals with itself must be neither too sophisticated,

> nor too elementary, a situation reminiscent of maximizing the product of two factors with a given sum. We can summarize the third requirement of our metalanguage in these words: it must rest upon a minimum of facilities, but still remain convenient enough to be used in practice.

The idea of separating program formulation from the efficiency issue is not new. For example, Kennedy and Schwartz, in [12], state:

> Most programming languages make it impossible to separate issues of problem formulation from those of efficiency. For this reason, the process of problem formulation often becomes entangled with the problem of choosing data structures which will lead to highly efficient program realizations. While in a complete implementation both problems must be faced, it is well to have available a mechanism which allows these two problems to be treated separately during the initial phase of work on a complex program. Observe also that too-early choice of data structures can create subsequent difficulties. In particular, necessary algorithmic changes discovered after the start of programming can require that the data structures originally specified be modified at great cost to a project. The result can be a poorly formed system which can take a great deal of time to debug or which may never work properly. The same comments apply to the specification of interfaces between modules.

Many of the difficulties just noted can be avoided if low-level efficiency-related decisions are postponed until after algorithms are designed and debugged. Putting this another way, if the programmer can first build and verify his system, representing it coarsely, he can subsequently design data structures and interfaces which fit his algorithmic scheme well. This approach might be thought of as a two-stage

> development process. The first stage produces an abstract algorithm in which low-level efficiency considerations are ignored; the second stage converts the abstract algorithm to a concrete algorithm by the addition of data structuring.

The two-stage development process described above is similar contextually to the supercompilation process described in [1] as stage 1, in conjunction with the mapping process detailed in this paper as stage 2. The authors proceed to describe their set-theoretic language, SETL, as a basis for separating development from efficiency problems.

Also, Darlington and Feather [10] described a "transformational" approach to program modification:

> It has become apparent that an alarming proportion of programming effort is devoted to maintaining or modifying existing software as opposed to creating new products. This process of modification is very poorly understood. All too often attempts at modification introduce more errors than they remove. The root cause of this trouble is the way the programming task is approached; the code available for modification is code meant to be run. Even if this has been written in a high level language it has to be written to be efficient at the expense of clarity and therefore modifiability.
>
> The tranformational approach to programming divides the task into at least two stages. First a complete specification of the program is written emphasising clarity and understandibility. Then a runnable program is produced from the specification by applying transformations guaranteed to preserve correctness while improving efficiency. Thus modifications, when necessary, can be made to the specifica-

> tion which is of a form to facilitate this process. As we hope to demonstrate in this paper the correctly modified efficient program can then be produced by redoing the transformations.
>
> ... meta-language programs provide a concise way of recording the tranformations used. ... Even when the meta-language program has to be changed, this, as we hope to demonstrate, involves less work than attempting to modify the efficient program directly with all the dangers of introducing unforeseen errors. For if the system succeeds in carrying out the transformation plan indicated by the meta-language program the program produced is guaranteed correct relative to the modified specification.

The authors of the above go on with examples of their program transformations, leaving the result in NPL, a very high level recursion equation language. They suggest that the next step in their process is the mapping of this NPL output onto Pascal, in order to be able to actually execute the resulting programs. This is quite similar to the current work, with the supercompiled graph of states being the high level algorithmic language, and a sequential language (in this case also Pascal) being the target. Darlington and Feather also refer to previous works on the same mapping problem, in particular, [11], which describes the process of conversion of recursion to iteration.

Compiler-compilers, or automatic compiler generation systems, have intrigued programmers since the first For-

tran compiler was written. These systems typically take some form of syntactic and semantic definition of an ad-hoc programming language, and produce lexical analyzers, and syntax analyzers (parsers). Aho, in [13], gives an overview of the state of the art up to 1980, and includes an exhaustive bibliography. Up-to-date bibliographies have also been written by Ganapathi and Fischer [15,16].

Aho, in [13], describes a template-matching process for code generation, quite similar to the code generation scheme used in the current work:

> ... code generation is often implemented as a template-matching process. The intermediate language program is a tree. Code generation consists of covering the tree with templates representing machine instructions. This tree template matching approach allows many machine dependencies to be isolated into the tables containing the templates. This approach appears attractive for use in compiler-compilers in that it may be possible to derive the template tables automatically from a formal specification of the target machine and in that the same compiler can be used to generate code for a new machine by changing only the contents of the template tables.

Refal, as a pattern matching language, is ideally suited to such a template-matching process, and the intermediate language LISA, developed in this work, was designed with the above goals in mind. The LISA compiler developed in this work is written in Refal, and applies a table of templates to the LISA program in order to produce code in

Pascal. Graham, in [14], describes in more detail the actual design criteria necessary for creating a table-driven code generator, and discusses some of the practical problems uncovered in the development of a retargetable code generation package.

2. THE REFAL MACHINE

Refal (Recursive Functions Algorithmical Language) is a functional programming language which was developed in the Soviet Union by Valentin F. Turchin and his co-workers in the late 1960's. It was developed as a universal metalanguage for the formal definition of algorithmic languages, either oriented towards classes of problems, or invented ad-hoc for specific problems. It can also be viewed as an ordinary algorithmic language, used for symbol manipulation, similar to LISP or SNOBOL.

A Refal program consists of a number of function definitions, and a starting configuration for the Refal machine. Each function definition consists of a number of Refal sentences, each of which is comprised of a pattern expression and a replacement expression. The execution of a Refal function, then, is a process of matching of the successive pattern expressions within a function definition against the argument of the function. When a pattern is found not to

match the argument, the pattern of the next sentence is tried. If and when a match is found, the function call is replaced by the replacement expression, which may itself contain function calls. If no match is found, an abnormal stop of the Refal machine occurs. For a detailed description of the history of Refal, see [1:x-xi].

The present Refal system at CUNY/UCC consists of:

- A Refal compiler, which accepts Refal programs written in Metacode B[3], and converts it into an intermediate language called RASL (for Refal ASsembler Language). The RASL consists of simple Assembler macros and commands which represent the processes of pattern recognition and expression replacement found in the Refal machine. The RASL program is then assembled into a machine code program by the usual IBM Assembler.
- A RASL interpreter. The interpreter takes the assembled RASL program, and places it in its memory field. It then places the starting configuration for the Refal machine in its view field, and begins execution by concretizing this configuration. Execution continues either until all expressions in the view field have been evaluated (i.e., no concretization brackets are left), or until an abnormal stop occurs. An abnormal stop means that an expression must be evaluated for which

[3] See Appendix B for information on Refal source coding.

there is no pattern expression matching it in the memory field.

- A set of external functions which perform actions not defineable in the Refal machine itself*. Examples are functions /CARD/ and /PRINT/, which perform simple input and output for the Refal machine.

2.1 THE REFAL SUPERCOMPILER

The present Refal Supercompiler is itself written in Refal. This fact is one of the bases of the concept of supercompiler: a supercompiler system is written in a language which is its own metalanguage. Programs for modifying and improving the supercompiler, then, are written in Refal, and by applying a series of "metasystem transitions" one can devise algorithms for optimizing algorithms, algorithms for optimizing algorithms that optimize algorithms, etc. (see [1:v-vii] for a discussion of the use of metalanguages in the supercompiler). A description of recent experiments in algorithmic optimization using the supercompiler can be found in [2]. It is possible to apply metasystem transition rules to the supercompiler program as input to the supercompiler itself, and thus to arrive at a more efficient supercompiler, compiled compilers, etc.

---

* See Appendix A for a list of external functions and their uses.

2.2 SAMPLE REFAL PROGRAM

In order to facilitate the description of the mapping process, a simple Refal program is presented below as figure 1. It will be referred to throughout the remainder of this text.

```
1    JOB = K/PROUT/ K/FA/ K/CARD/...
2    FA 'A' EX = 'B' K/FA/ EX.
3     SY EX = SY K/FA/ EX.
4     =
5    =END
```

Figure 1: Sample Refal program

Function /JOB/, defined in line 1 of the program, describes the initial state of the Refal machine. /JOB/ is a special name used by the Refal system to indicate the starting state of the Refal machine, and is not callable from within the program. The definition of /JOB/ then says that the program should start by executing function /CARD/, passing its result to function /FA/, which then passes its result to function /PROUT/. Functions /CARD/ and /PROUT/ are built-in functions in Refal, which serve to perform I/O. Function /FA/, on the other hand, is described in the program itself, in lines 2 through 4. Line 2 says that if the argument to /FA/ starts

with the symbol B, then the result is the symbol A followed by the result of evaluation of /FA/ on the rest of the input. Line 3 indicates that if the argument starts with any other symbol, the symbol is to be left outside the evaluation brackets K and . and concatenated with the result of the further evaluation of /FA/. Line 4 indicates that an empty argument should return an empty result, i.e., no more evaluation brackets are to be stacked. Line 5 indicates the end of the program.

3. THE GRAPH OF STATES

The Refal Supercompiler operates on programs by taking the program, breaking it up into the most elementary operations, and producing a graphic representation of the program. Each arc in such a graph represents a transition from one state of the Refal machine to another, and the operation necessary to transit that arc. Each node represents a state of the machine. The graph of states then represents a history of a computation, with all the possible results of that computation. The supercompiler performs optimizations to the graph by driving the starting configuration of the machine through this history of computation, and determining which parts of the graph may be discarded, which may be looped back on themselves, etc. The reader is referred to [1] for an in-depth description of the process of supercompilation.

This chapter describes the resulting graph of states, which in itself is an abstract algorithm for solution of the original programming problem. The following sections give an example of a graph of states, and detail the concepts necessary to understand a graph of states.

3.1 EXAMPLE OF A GRAPH OF STATES

A graph of states may be represented in two formats. One is a graphic representation, the other is useful for printing out the graph as produced by a computer. As examples of both formats, the graph of states of the program given in figure 1 are given below, as figures 2 and 3.

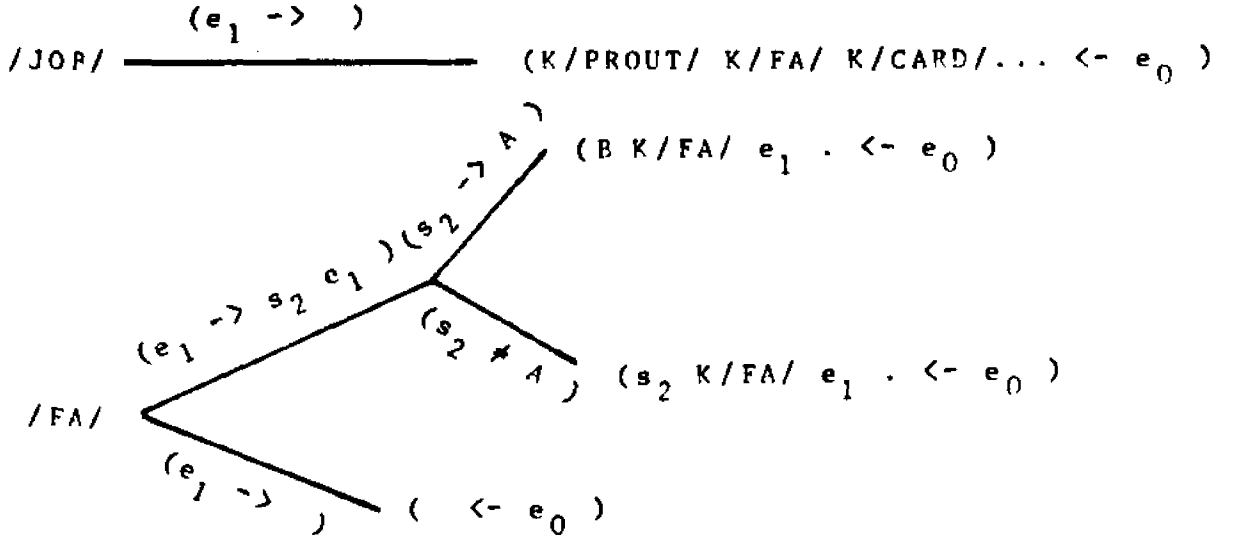

Figure 2: Graphical representation of a graph of states

```
(/JOB/)
:(  (*E/1/=())[ ((*(/PROUT/*(/FA/*(/CARD/))))=*E/0/] )

(/FA/)
:(
  (*E/1/=(*S/2/*E/1/))
   (
    :(
      (  (*S/2/=(A))[ (B*(/FA/*E/1/))=*E/0/] )
      +
      (  (*S/2/*(A))[ (*S/2/*(/FA/*E/1/))=*E/0/] )
     )
   )
  +
   (
    (*E/1/=())[ ()=*E/0/]
   )
 )
```

Figure 3: Computer output representation of a graph of states

3.2 DEFINITION OF GRAPH OF STATES

The following is a Backus-Naur Form definition of a graph of states as produced by the Refal supercompiler. The BNF has been extended with square brackets, for an optional entity, and braces, for repetition zero or more times of the enclosed entity. Parentheses in a BNF definition are Refal structure brackets, which are used to analyze the tree structure of the graph.

```
<total-graph> ::= { (<f-name>) : (<f-graph>) }
<f-name> ::= / <identifier> /  |  C <macrodigit>

<graph> ::= <p-segment> <walk-end> | <p-segment> <branching>
            <branching>  |  <walk-end>
<f-graph> ::= <graph>  |  Z
```

```
<branching> ::= :( (<branch>) + (<branch>) [+ (<branch>) ] )
<branch> ::= <discriminating-operation> <graph>

<discriminating-condition> ::= <condition>
<p-segment> ::= <condition> { <condition> }

<condition> ::= <contraction>  |  <restriction>

<walk-end> ::= { <assignment> } <assignment>
```

3.3 FUNDAMENTAL CONCEPTS

The entity <f-graph> gives the definition of one function (or configuration) of the Refal machine. It is the parallel to one function definition in Refal. Such a graph consists of a tree structure, through which a number of walks may be followed from left to right. The left part of a walk, or condition chain, corresponds to the left side of a Refal sentence, which performs pattern matching. The condition chain consists entirely of contractions and restrictions, as described below. The right hand part of each walk, called the walk-end, consists of a chain of assignments. These assignments are the parallel of the right side of a Refal sentence, which performs the replacement of the function call by its result.

A contraction in a graph of states represents a test of a condition and a (temporary) assignment if the condition is found to be true. A contraction always con-

sists of a variable on the left side, followed by the sign '->', and an expression on the right side. The supercompiler is set up in such a way that there are only seven elementary formats for contractions. In natural Refal code then, we may have a contraction:

```
(e   -> s  e )
  1      2  1
```

This contraction performs two functions: first, we test to see whether the expression e1 starts with a symbol. If it does, then we assign the value of that symbol to a new variable, s2, and then contract the value of e1 by deleting the first symbol from it. If it doesn't, a branch is taken. Thus, e1 has been redefined. Subsequent operations (i.e., those further to the right) in a walk which refer to e1 refer in fact to this contracted value.

In Metacode A, we write the contraction in a slightly different format: (*S/1/=(*S/2/*E/1/)). Notice that the '->' sign is replaced by an equal sign, and that the expression on the right is placed in parentheses. Also notice that, as the supercompiler only uses macrodigits for indices, the metacode A contraction has macrodigits for indices. For simplicity, contractions will usually be shown in metacode A henceforth. The following is a listing of the allowable formats for contractions. Indices are shown as lower case letters, but will in actuality be

macrodigits. The letter C indicates a character or specific object symbol.

```
(*Ei=())                          (type 0 contraction)
(*Ei=(*Sj*Ei))                    (type 1 contraction)
(*Ei=(*Ei*Sj))                    (type 2 contraction)
(*Ei=((*Ei)*Ej))                  (type 3 contraction)
(*Ei=(*Ej(*Ei)))                  (type 4 contraction)
(*Si=(C))                         (type 5 contraction)
(*Si=(*Sj))                       (type 5 contraction)
```

Restrictions take a format similar to contractions, and denote the inverse operation: if the condition is not true, then continue through the walk. We have three elementary restrictions:

```
(*Ei≠())                          (type 0 restriction)
(*Si≠(*Sj))                       (type 5 restriction)
(*Si≠(C))                         (type 5 restriction)
```

Now we can define the format of an assignment, and what it means. As seen above in the BNF definition of a graph of states, an assignment occurs in a <walk-end>, which represents the right-hand side of a Refal sentence. Each assignment represents the assigning of a value to a particular variable. Thus, the format for an assignment will be an expression on the left side, followed by the sign '<-', followed by a variable. The expression on the left may be any, as long as any variables indicated in it are already defined in the <p-segment> part (corresponding to the left side of a Refal sentence) of the particular walk in which the assignment appears. For example, following the above contraction example, we may have an assignment:

```
(e s ABC e <- e )
  1 2     1    0
```

This assignment means: compose an expression consisting of the concatenation of expression e1, symbol s2, object symbols 'ABC', and a copy of expression e1, as defined within the walk. Then assign the resulting value (i.e., expression) to variable e0. In metacode A, we will write the above assignment as: ((*E/1/*S/2/ABC*E/1/)=*E/0/).

The variable *E/0/ has a special significance in a graph of states: it represents the output of a particular configuration. Thus the above assignment would have to be the last one in its <walk-end>, and composes the expression which is the result of evaluation of the function in which it appears.

The graph of states represents a program in Refal. From the above BNF definition, we can see that each <f-graph> consists either of a <graph>, or the letter Z. Z is the symbol which indicates 'recognition impossible', one of the states of the Refal machine. In Refal, each function is defined by a number of sentences. As seen in appendix A, each sentence consists of a pattern expression on the left side, followed by an equal sign, and an (output) ex-

pression on the right side. The Refal machine works by matching the pattern on the left side of each sentence in a function definition against the argument for the function call, and when a match is found, replaces the function call by the expression composed in the right side. This (output) expression may or may not contain concretization brackets. In any event, the supercompiler breaks down the process of pattern recognition into its smallest elements, by replacing the pattern expression with contractions (and internally, also with restrictions). Then, the patterns, in contraction form, are compared, and common elements are combined, so that we end up with a graph. Let's say that we have the following definition of a Refal function in metacode B, taken from figure 1:

```
FA 'A' EX = 'B' K/FA/ FX.
   SY EX = SY K/FA/ EX.
         =
```

This function does the following. It takes an expression as its argument, and for each occurrence of the character 'A', replaces it with 'B'. The second sentence says that any other symbol is to be left alone. The third, and last sentence, says that when the function is called with an empty argument, the result of evaluation is nothing. Thus, we have defined a recursive function /FA/. In graph of states format, we could see function /FA/ to be:

```
 1  (/FA/):(:(
 2            (
 3             (*E/1/=(*S/2/*E/1/))
 4               :(
 5                  (
 6                   (*S/2/=(B))[ (B*(/FA/*E/1/))=*E/0/ ]
 7                  )
 8                  +
 9                  (
10                    (*S/2/*(A))[ (*S/2/*(/FA/*E/1/))=*E/0/ ]
11                  )
12                 )
13           )
14           +
15           (
16            (*E/1/=())[ ()=*E/0/ ]
17           )
18         ) )
```

Line 1 gives the name of the function described in the graph (/FA/), the colon and left bracket containing the <f-graph>, and the colon and left bracket surrounding a <branching>. The paired brackets appear at the end of the graph, in line 18. Lines 2 and 13 are the brackets enclosing one <branch>. Line 3 contains a contraction, which breaks the initial argument into one symbol on the left and a contracted expression. Lines 4 and 12 give the brackets surrounding a <branching>, this one at a lower level, within the first <branching>. The brackets in lines 5 and 7 surround one <branch>: line 6. This line says that if *S/2/ (defined in line 3) has the value A, then perform the assignment: *E/0/ becomes the concatenation of the letter B with the result of evaluation of /FA/ on *E/1/ (i.e., the remainder of the original expression after

the first symbol is removed). The brackets in lines 9 and 11 surround the second <branch> in this <branching>, line 10. Line 10 says: if *S/2/ (defined in line 3) is not the letter A, then compose the output expression as whatever the value of *S/2/ is, followed by the result of applying /FA/ to the remainder of the original expression *E/1/. The + in line 8 separates the above two <branching>s. The + in line 14 separates the two higher level <branching>s with the brackets in lines 15 and 17 surrounding the second main branch: line 16. This line says: if the input to the function is the empty expression, then the output is the empty expression.

3.4 THE GRAPH OF STATES AS AN ALGEBRAIC OBJECT

The graph of states of the Refal machine is in fact an algebraic representation of a computer program, written in an abstract form. The supercompiler operates by applying various rules of reduction and driving (i.e., the impressing of a particular case of input to the program onto the "history" of computation represented by the graph) to the graph of states, thus performing optimizations to the original program. The elementary contractions and restrictions presented earlier (see section 3.3) are the basis for the algebraic manipulation rules. For a discussion of the manipulation rules, see [1:100-102].

4. LISA - THE LANGUAGE FOR IMPLEMENTATION BY STACK ASSIGNMENTS

The language LISA has been developed as a universal programming language into which a supercompiled graph of states may be easily transformed. LISA allows description of variables with arbitrary data types, and specifies for each variable the means of accessing and storing it. A LISA program consists essentially of a number of function definitions, each of which corresponds to one function subgraph from the overall graph of states. No mention is made in a LISA program as to how the program is to be implemented -- the LISA program is assumed to be machine independent, representing an abstract LISA machine. The following sections describe the language itself, and the present implementation of a LISA machine in Pascal.

4.1 MAPPING CONCEPTS

It is useful at this point to describe more fully the concepts of mapping. The remainder of this section is excerpted from [1:134-139]:

> The most general principle of mapping of the Refal machine on a target machine (computer) is: to each configuration of the Refal machine there corresponds a control point in the program of the computer, and to each variable in the graph of states of the Refal machine there corresponds a variable in the computer program

together with an information field in the computer memory and an access method to use this information.

The linkage between the Refal machine and the computer is established by the concept of mapped variable. If Vi is a variable, then the mapped variable is

Vi in M

where M is a mapping of this variable, i.e. an object which encodes all directions necessary to have access to the value of the variable. The concrete form of M depends, of course, on convention. The access method used must be encoded, and some specific information must be provided, such as numerical or symbolic word addresses etc. A configuration, in which all variables are mapped, will be referred to as a mapped configuration. E.g., the mapped configuration

C6 (s1 in A23) (e2 in A24, A25)

may signify that when control in the computer is at the point corresponding to configuration C6 of the graph of states of the Refal machine, the value of the variable s1 is stored in word A23 (symbolic address), and the value of e2 is stored in the field beginning at A24 and ending at A25.

To start turning a graph of states into a program for a computer we must somehow map the input configuration. In fact, the mapping of the input configuration should be an integral part of the exact formulation of the job. To define the final program uniquely, one must specify not only the input (initial) configuration, but also the way the input data is stored in the computer. The same is true for the output configuration: the program will vary depending on the output data we choose.

A compilation task must include a program P in Refal, and two mapped configurations: the input configuration C1 and the output configura-

tion C2. We will join these two configurations into the i/o quasisentence:

C1 => C2

E.g., a typical i/o quasisentence might be:

K/L/ GROSS(CAT); ADD(DOG) (ex in R1, R2) => ey in R1, R3

which means that the input string ex should be taken from the field with the boundaries stored in registers R1 and R2, while the boundaries of the output string should be stored in R1 and R3...

... Transformation of a Refal program (or a graph of states of the Refal machine) into a program for the target machine, i.e. the mapping process in the general sense, includes procedures of two types: mapping proper, which refers to variables and configurations, and translation of arcs and other subgraphs of the graph of states (when all of the variables are mapped) into instructions for the target machine. The main unit in the translation process is a translation statement. It consists of two parts separated by a horizontal bar, the top part being an element of the graph of states, and the bottom part being its computer equivalent (translation). The final result of the mapping process, as well as intermediate results and some prerequisites, are translation statements.

For a subgraph with mapped input and output configurations the translation statement is:

<i/o quasisentence>

<corresponding computer program>

If we make a mapping simultaneously with the compilation of the graph of states of the Refal machine, the configurations of i/o quasisentences will be expressed in terms of the original Refal program. If we first compile a graph of states and when this part of the job is finished proceed to map, then configurations in quasisentences will appear in a standard notation (normal form). Translation of external function calls is also performed with the aid of corre-

sponding translation statements. Suppose, e.g., that the target machine has an instruction for addition which in the assembler language (serving as the target language) is written in the form:

ADD, A1, A2 -> A3

(a three-addressed machine). Let us use + as the determiner of the external function performing the operation designated as ADD, and let the format be

K + (N1) (N2) .

where N1 and N2 are the numbers to be added. (We ignore the type of numbers they are and their representation, although the user should of course know this). The translation statement which would allow us to use this external function might be:

K + (e1 in A1) (e2 in A2) => e3 in A3

ADD, A1, A2, -> A3

The essence of the mapping process is: starting with the configurations already mapped, move along the arcs and then map unyet mapped configurations in such a way so as to avoid unnecessary moves of the information in the computer. Contractions are translated into conditional statements and definitions of new variales, whereas assignments are translated into assignments. Decompositions will become procedure calls.

The full state of the Refal machine is not represented, generally, by a configuration, but by a vertical segment, i.e., a composition of a number of configurations (a stack of function calls). The configurations (vertices, to be precise) of a graph of states fall into nonrecurrent, which do not appear in the vertical segments generated by them, and recurrent, which do appear there. Recurrent configurations, in turn, fall into static and dynamic ones. A configuration is static if it appears only on the top of any of the vertical segments generated by it. Otherwise, it is dynamic. Static recurrent configurations correspond

to iteration, and should be programmed as nonrecursive procedures. Dynamic configurations correspond to "recursion proper" and must be programmed as recursive procedures. Analyzing the graph of states, it is easy to break down all active configurations into nonrecurrent, static and dynamic.

The graph of states is a sort of flow chart of the future program, written in a special language. This language is rather abstract and doesn't specify some of the important features of the computer program (the mapping of configurations), so we have some freedom of action for program optimization. Mapping is essentially code generation, with a graph of states as the source program. Methods and techniques of efficient code generation developed by different authors using different source languages can and should be used for mapping.

4.2 THE NEED FOR A UNIVERSAL INTERMEDIATE LANGUAGE

Upon closer inspection of the aforementioned requirements for a successful mapping of a graph of states onto Pascal, and with the additional realization that the Refal supercompiler is intended as a universal optimizing engine for computer algorithms, one may conclude that certain of the aspects of the mapping onto Pascal will be in common with a mapping onto any other language. It is possible to envisage a scheme whereby all the common features of transformation of the graph of states into a program in some other language are performed in one phase (mapping proper), and the actual production of code in the target language is saved for a second phase. This second phase

could be performed by a simple Refal program which operates by applying a set of production rules (the translation statements described in the preceeding section) to the mapped program, in order to generate code in the target language.

Thus we arrive at the need for an intermediate language, which contains all the information necessary to map a graph of states program onto a target language, including the mapping of each variable and function call onto the proper stack locations of the target machine. Such a language has been developed, and has been implemented as part of this project. The language is called LISA, for Language for Implementation by Stack Assignments. It is similar in some ways to the graph of states itself, but contains only mapped variables, and some special LISA commands which could be called compiler directives. The syntax of LISA is described in detail in section 4.3, and the corresponding semantics in section 4.5.

With the establishment of the need for LISA, the scheme for mapping from a graph of states to a target machine can be formalized as taking two steps. They are: 1) transformation of the graph of states into a LISA program; 2) transformation of the LISA program into a working target language program. Both of these func-

tions have been implemented with the data types allowed being just those which exist in Refal programs, i.e., doubly-linked lists and symbols. In addition, basic work has been begun on expanding the present implementation to other data types, and for allowing automatic data typing (see section 4.6.6).

The first step, transformation of the graph of states into LISA, includes: 1) transformation of the graph into a sequential notation; 2) analysis of data typing requirements; and, 3) any optimizations to the original graph. The result, a LISA program, is ready for the second step: translation into the target language. This second step is performed by a simple table-driven code generation scheme, and greatly simplifies the problems of code generation. The LISA program is produced in a simple tree structure which may be easily analyzed by a short Refal program containing the translations for each LISA statement. The retargetting of the second, or LISA compilation phase, is simple to accomplish by hand. In fact, the LISA format makes it quite simple to adjust the code generated to particular programming problems, by recognizing the LISA statements produced, and writing more efficient translation statements. Once a mapping for Pascal is created and tested, the mapping onto another language, say an assembler language, would simply entail the transla-

tion by hand of the translation statements into translation statements for assembler language. This process would have to be performed just once, after which mappings for both languages would exist. Of course, all the background functions of the operating environment would also have to be translated into the new language, but again, this task would need to be performed just once. The process of program development may then be seen as taking three steps: 1) definition of the problem; 2) supercompilation; and, 3) adjustment of data types and code translations, either automatically, as a default, or with human intervention, for fine tuning.

4.3 BNF DEFINITION OF LISA

The following is an extended Backus-Naur Form description of LISA. The extensions to BNF used are square brackets, which indicate an optional entity, and braces, which indicate repetition zero or more times of the entity enclosed. Notice that the stucture is quite similar to that of a graph of states as produced by the Refal supercompiler. The notable difference is that each branch point is preceded by a label, and that all variables are mapped.

```
<LISA-program> ::= <program-comment> <end-label>
     (<internal-label-list>) (<function-id-list>)
     <starting-function-definition> {<function-definition>}
```

```
<end-label> ::= <label>
<internal-label-list> ::= L <label> {<label>}
<function-id-list> ::= C <function-id> {<function-id>}
<function-id> ::= <function-name> |
     <function-name> (<external-function-name>)
<external-function-name> ::= <string>

<starting-function-definition> ::= <function-definition>
<function-definition> ::= <initial-comment> <function-name>
     :( <tree> )

<program-comment> ::= ( <string> )
<initial-comment> ::= ( <string> )
<function-name> ::= <label>
<label> ::= <macrodigit>
<macrodigit> ::= / <digit-string> /

<tree> ::= <c-segment> <walk-end> | <c-segment> <branching> |
            <branching> | <walk-end>

<branching> ::= :( (<branch>) <label> + (<branch>)
     [ <label> + (<branch>) ] )

<branch> ::= <discriminating-contraction> <tree>
<discriminating-contraction> ::= <contraction> | <empty>

<c-segment> ::= <contraction> {<contraction>}
<contraction> ::= ( <mapped-variable> =
     ( <pattern-expression>) )

<walk-end> ::= {<garbage-collection-directive>}
    {<assignment>} <assignment> {<interpreter-directive>}

<assignment> ::= ( (<expression>) = <mapped-variable> )
<compiler-directive> ::= <garbage-collection-directive> |
    <interpreter-directive> | (? <expression> )

<garbage-collection-directive> ::= (?OUT) |
                      (?OUTAV(<offset>)) | (?RV(<offset>))

<interpreter-directive> ::= (?POP) | (?XP(<macrodigit>)) |
          (?BP(<macrodigit>)) | (?X(<label>)) | (?INCSTEP) |
          (?F(<function-name>)) | (?L(<macrodigit>))

<symbol> ::= <object sign> | <compound symbol>
<compound symbol> ::= /<object string>/
<expression> ::= <empty> | <expression><term>
<empty> ::=
<term> ::= <symbol> | (<expression>)

<mapped-variable> ::=
    *M ( <data-type> <status> <role> <offset> )
```

```
<data-type> ::= <term>
<status> ::= A | G | P | R | T
<role> ::= + | -
<offset> ::= <macrodigit>
```

In the above BNF definition, some of the entities are left either undefined, or are not completely defined. <string>, <object-string>, <digit-string>, etc. are defined as usual. <data-type> is purposely left undefined so that the language has the ability to accept various specifications of data types of variables. Of course, in order to implement various other data types, it will be necessary to create the appropriate background functions for a new operating environment in the target language. However, in LISA the operating environment of the target program is invisible. The present implementation accepts just data types E and S, corresponding to the expressions and symbols in interpreted Refal. (The term "interpreted Refal" refers to the current implementation of Refal, where an expression is represented as a doubly-linked list, and a symbol is one element of such a list). The entity <pattern-expression> is left undefined for brevity. In fact, this entity corresponds to the right side of a contraction found in a graph of states, with the restriction that all variables are mapped. Thus, the set of <pattern-expression>'s is a restricted set (see section 3.3).

4.4 EXAMPLE OF A LISA PROGRAM

Before giving a formal description of the semantics of LISA, it is worthwhile to see a sample program in the language, in order to give the reader a sense of what is being presented. A short Refal program was presented as figure 1. The corresponding graph of states, as produced by the Refal supercompiler, was presented as figure 3, in chapter 2.2. The program represented in these figures will be referred to throughout this chapter.

Figure 4 is the LISA translation of the program. In order to make the structure of the program more clear, blanks have been added freely, and the structure brackets surrounding each assignment have been replaced by square brackets. The original LISA program, of course, consists of one string of object symbols. Additionally, each walk segment which is too large to fit on one line has been folded over onto a number of lines, with all lines after the first one indented.

```
 1    (*** MAPGOS1 VERSION OF 10 APR 1983.)
 2    /7/
 3    (L/C//3//4/)
 4    (C/1//2//5/(PROUT)/6/(CARD))
 5    (* /JOB/ = CONFIGURATION NUMBER 1 *)
 6    /1/:(
 7          (?OUT)[ (*(/6/))=*M(EA+/4/) ][ (*M(EA+/4/))=*M(EA-/3/) ]
 8             [ (*(/2/(*M(EA-/3/))))=*M(EA+/2/) ]
 9             [ (*M(EA+/2/))=*M(EA-/1/) ]
10             [ (*(/5/(*M(EA-/1/))))=*M(EA+/0/) ]
11              (?BP(/3/))(?XP(/5/))(?POP)
12        )
13
14    (* /C2/ = CONFIGURATION NUMBER 2 *)
15    /2/:(
16          (
17           (*M(ET-/1/)=(*M(SP-/4/)*M(BP-/3/)))
18           :(
19              (
20               (*M(SP-/4/)=(A))(?RV(3))(?OUT)
21                  [ (*M(EA-/3/))=*M(EA-/1/) ]
22                  [ (*(/2/(*M(EA-/1/))))=*M(EA+/0/) ]
23                  [ (B*M(EA+/0/))=*M(EA+/0/) ]
24                   (?BP(/1/))(?XP(/2/))(?POP)
25              )
26         /3/+
27              (
28               (?RV(4))(?RV(3))(?OUT)
29                  [ (*M(EA-/3/))=*M(EA-/1/) ]
30                  [ (*(/2/(*M(EA-/1/))))=*M(EA+/0/) ]
31                  [ (*M(SA-/4/)*M(S"+/0/))=*M(EA+/0/) ]
32                   (?BP(/1/))(?XP(/2/))(?POP)
33              )
34            )
35          )
36       /4/+
37          (
38           (*M(ET-/1/)=())(?OUT)[ ()=*M(EA+/0/) ](?POP)
39          )
40        )
```

Figure 4: Sample LISA program

4.5 SEMANTICS OF LISA

The definition of an algorithmic language implies the existence of an abstract machine which accepts a program in that language and executes it. The LISA machine can be described in terms of the objects it operates on, the allowed elementary operations, the method of determination of the order of operations, and the environment in which the machine operates.

The basic object on which a LISA machine operates is a data type. Each variable in a LISA program is mapped, so that its data type is fixed. A data type, in the last analysis, corresponds to a field in the computer's memory, specifying only partially the types of operations that may be performed on a particular variable. Each mapped variable has its own data type indicator. This indicator may be used in the code generation phase to determine the encoding of the variable in the information field in the computer. The remaining indicators of a variable's mapping determine the method of access and the location of the particular variable.

The assignment of a data type to a variable is performed by the first phase of the mapping scheme. At present, the only implemented data types are those corresponding to the data types of interpreted Refal. These are

data type S, which implies either one object symbol (character) or a macrodigit (Refal implementation of a non-negative integer), and data type E, which refers to an expression (doubly-linked list structure), in which objects of type S may exist. It is important to remember here that in Refal, parentheses (called structure brackets), are not only linked within an expression with the usual forward and backward links, but also paired parentheses are linked to each other. This is the method by which the tree structure of any Refal expression may be analyzed.

The mapping scheme leaves room for other data types by specifying that the data type indicator of a variable mapping is simply some term. An algebra of data type derivation is being developed, and is discussed further in section 4.6.6.

Data type E is the most general data type. It is the default data type for any variable which can not be reduced to a more restricted format. Data type S is again a general type, although allowing just one element. In interpreted Refal, all elements of type S are treated in the same way, but in a LISA program the handling may be different for macrodigits, characters, special symbols, etc.

In a LISA program there are two basic operations allowed: contraction and assignment. A contraction represents a condition test followed by definitions of new variables (if the condition test succeeds), or by a branch to another point in the program (if it fails). An assignment is just what the name implies: assignment of a value to a variable, as the result of a completed computation.

The general format for a LISA contraction is:

(V=(L))

and the general format for an assignment is:

((E)=V)

where all the V's are mapped, and L and E consist of expressions in which any variables appearing are mapped.

LISA implies a method of execution of a program. The body of a LISA program consists of a number of function definitions. A function definition is written with a tree structure, using Refal structure brackets to delimit the different branches. Each internal branching point in the tree is labelled, and recognition of the appropriate branching destination is easy to implement in the Refal program which generates target language code.

Each function definition describes a process to be performed, which consists of a number of paths or walks through the definition. Every walk starts at the same point in the tree structure, and begins with a number of contractions. A branching (or splitting of execution flow into a number of walks) occurs after every contraction. At the end of each walk, there is a chain of assignments (or walk-end), which describes the formatting of the result of the computation performed by the function. A particular walk-end is reached only upon successful completion of a unique set of the contractions in the beginning of the function definition. A walk-end may contain a number of nested and/or parallel function calls, indicating further processing of the output of the current function.

LISA is a purely functional programming language. That is, a program consists of an initial function call for the LISA machine to evaluate (this represents the task to be performed by the program), and the definition of a number of functions. When the LISA machine is started up, the initial function call is placed on an operating stack, and any input data is loaded into the appropriate locations in the information field of the real computer. Then control is passed to a stack interpreter, a section of the LISA machine which executes the stacked

function calls one by one. Each function is responsible for inserting any output in the proper location in the information field of the computer, and for adding any additional function calls to the operating stack. The called function then passes control back to the stack interpreter, which determines from the operating stack which function is to be called next. Finally, when the operating stack is cleared of all function calls, the LISA machine completes execution.

The environment in which a LISA program executes consists of two fields within a real computer. The program field contains the LISA program, along with the control (operating) stack, the stack of variables, and the stack interpreter. The information field contains the values bound to the variables in the program field. See figure 5, below.

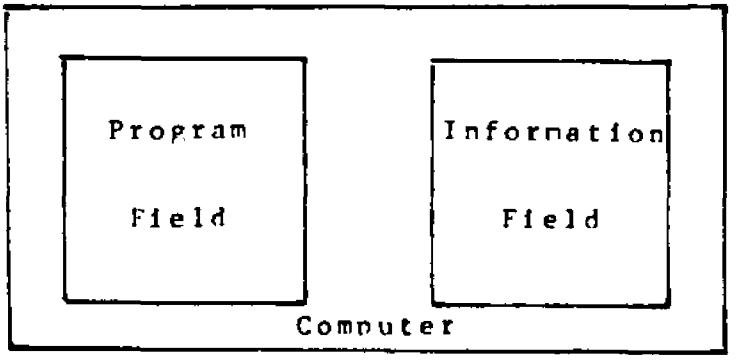

Figure 5: LISA machine environment

The LISA machine implies the existence of two separate stacks in its operating environment. The first is the control stack, which contains function calls. Each function call consists of two parts: one identifies the function to be executed, and the other gives the number of arguments for that function call. The second stack in the LISA environment is the stack of variables. Each level in the stack of variables corresponds to one mapped variable in the LISA program. Such a variable is implemented as a pointer (or set of pointers) to a location or region in the information field of the real computer.

Figure 6 details the structure of the two stacks in the LISA environment. Notice that each of the stacks has a pointer indicating a current level. The pointer in the control stack indicates the current function being executed. The pointer in the stack of variables points to the basis of the current function, i.e., the location of the variable representing the output of the function. All other variables used by the function, whether input variables, or variables defined by the function itself, are referred to by offsets from the pointer in the stack of variables.

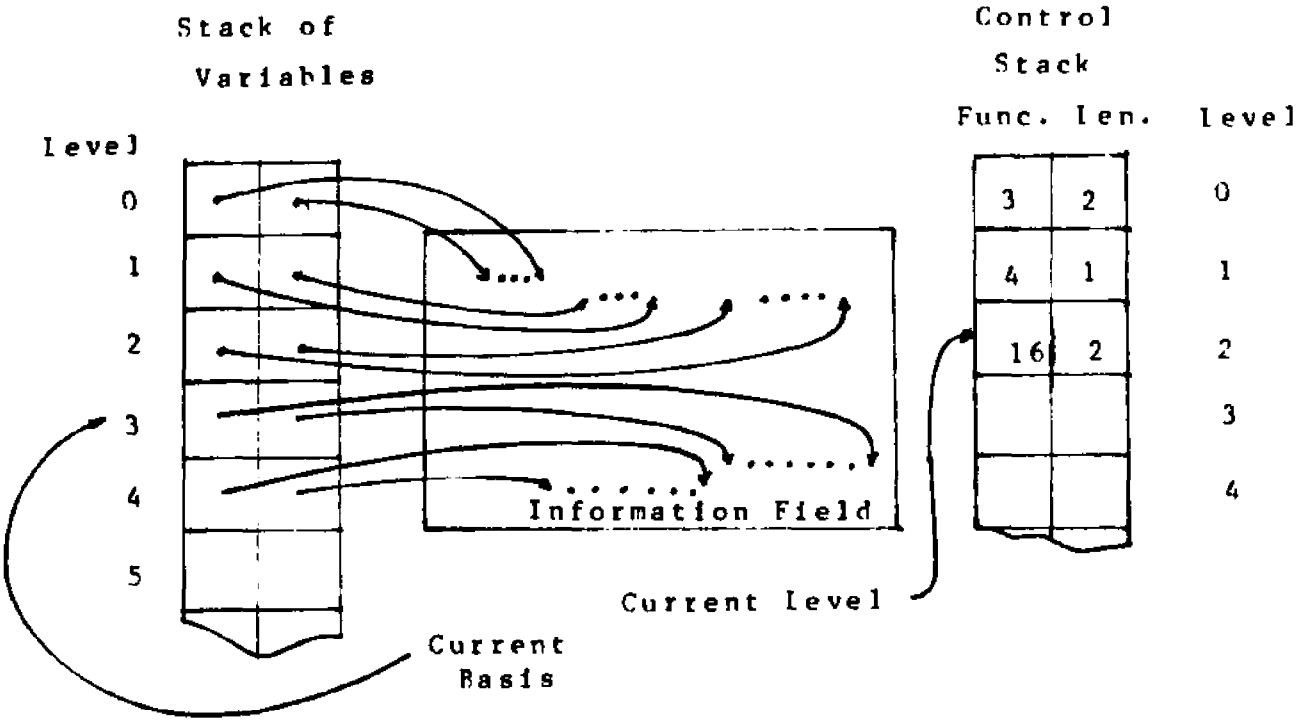

Figure 6: Stack structure in the LISA environment

In figure 6, the stack of variables consists of a number of pointers to sections of the information field of the computer. In the current implementation, these sections are doubly-linked lists. When other data types are implemented, the stack of variables will consist of pointers to other structures within the information field of the computer. In the figure, the current function call is for function number 16. Its length is 2. Correspondingly, the current basis (pointed to by the pointer in the stack of variables), is at level 3 in the stack of variables. This can be found by the sum of the lengths of all function calls further up in the control stack, i.e., 2 + 1 = 3.

Popping of the stacks (upon completion of a function call) is accomplished by first moving the control stack pointer up one level, and then moving the pointer in the stack of variables up by the length of the (now) current function.

The initial function call of the LISA program is placed on the control stack before execution begins. As new function calls are generated, the control stack pointer is always set to the lowest function call on the stack, and control is passed to the stack interpreter. When the pointer in the control stack points to the highest level in the stack, execution is completed.

A function definition in LISA consists of a labelled tree structure, containing contractions and assignments defining the walks through the tree. The branch destinations within the tree are exactly the labels of each branching point. When the condition test of one contraction fails, a branch is taken to the next walk at the same level in the tree structure. If no more branches exist at the same level, the function fails, and execution is halted. This state corresponds to the "recognition impossible" state of the Refal machine. Thus, the tree structure of a function definition determines the order of execution within the function, and the contractions

and assignments determine what operations are to be executed.

The term "LISA statement" refers to contractions, assignments, and LISA compiler directives collectively. These are the statements that are translated into target language code using the translation statements described earlier. The tree structure of a function definition in LISA is not a statement proper, but is used in the generation of target code simply to determine branching destinations, along with the labels at each branch point.

The environment in which a LISA machine operates has been described briefly in this section. A further discussion of the current implementation of the operating environment may be found in chapter 5.4.1. Sections 4.5.1 and 4.5.2, below, describe the objects on which a LISA program operates, and sections 4.5.3, 4.5.4, and 4.5.5, describe the operations performed by a LISA program.

4.5.1 Mapped Variables

A mapped variable in a LISA program is a string which specifies a data type, means of access, and location of a variable. The encoding used in LISA is general to the extent that any name may be given to the data type (within the syntactic constraints described above), but of

course the meaning of any new data type must be built into the resulting target code.

The encoding of a LISA mapped variable is given by:

*M (<data type> <status> <role> <offset>)

In Refal, the above expression will be treated as one object, referring to one mapped variable. As an example, we might have the mapped variable *M(EA+/1/), which refers to a variable of type E at location 1 in the stack of variables, which is the output of some function call.

The four indicators in a variable mapping can be seen as four "dimensions which are orthogonal". Each is needed to fully determine what the fate of the specific variable is to be. The following is a detailed description of the indicators in a variable mapping.

4.5.1.1 Data Type

The data type indicator of a mapped variable has already been described briefly. It remains to show how this indicator may be used for the extension of LISA into various data types other than types E and S of Refal.

Let us suppose that the mapping program determines that a certain variable may best be represented as an array of

characters, as opposed to the more general doubly-linked list structure of an E type variable. We may assign the data type (AC) to mean an array of characters, and data type C to mean a single character. Input variables then may have the forms: *M((AC)T-/1/) and *M(CP-/2/), for example, and we may have a contraction of the form:

(*M((AC)T-/1/) = (*M(CP-/2/) *M((AC)P-/3/)))

The meaning of this contraction is: take the array of characters at location 1, and split off the first character. Assign the value of this character to the variable at location 2, and the remainder of the original array to the variable at location 3. The 'T' in the orginal variable means that the variable is total (i.e., in its original uncontracted form), and the 'P' in the two variables on the right signifies that they are partial. The '-' in all three specifies that each is an input variable.

4.5.1.2 Status

The status indicator of a mapped variable describes either the origin of the variable (for values P, R, and T, described below), or whether or not the value in the information field of the computer may be changed in any way (values A and C, below). Values P and T occur

only in contractions, and values A, G, and R occur only in assignments. The values of the status indicator are defined as:

- A an Actual variable. This implies that the section of the information field pointed to by this variable may be changed in any way required. The inverse status is that of a ghost.
- G a Ghost variable. This indicates that the value in the information field pointed to by this variable may not be changed in any way. Thus, if a ghost variable appears in the left side of an assignment, the value of the variable must be copied, instead of being linked to the surrounding expression. A more detailed discussion of ghost variables may be found in section 4.6.3.
- P a Partial variable. This value applies only to the input variables found in contractions, and indicates that the variable is the partial result of a contraction (i.e., the variable is defined within the walk in some contraction). The inverse status is T, total. This P/T differentiation is useful in the optimization which deletes the definition of local variables which are not used in a function's output. See section 4.6.4 for a further discussion. Only variables with status P are eligible for this optimization.

- R a Replacement variable. Such a variable corresponds to the temporary variable needed when exchanging the values of two variables. Replacement variables are always assigned their values in the beginning of a walk-end, and are always used later in the same walk-end. For example, we may have the assignment:

```
( (*M(ER-/3/)) = *M(ER-/7/) )
```

This assignment saves the value of the R type variable at location 3 in a variable of corresponding type at location 7. The variable at location 7 will be used somewhere later (further to the right) in the walk-end, for example, as *M(ER-/7/). The reason for the replacement is that somewhere between the above assignment and the use of what was originally variable 3, an assignment appears with the variable on the right being at location 3, e.g.,

```
( (...) = *M(.../3/) )
```

where ... indicates an arbitrary expression. In order to prevent the overlaying of the still not used variable at location 3 with some computed value, it is saved at location 7.

- I a Total variable. This value applies only to input variables found in the contractions in the beginning of a walk, and indicates that the variable is still in its uncontracted state, and is an argument of the current function call.

4.5.1.3 Role

The role indicator of a mapped variable relates the usage of the variable to the right-for-left-side substitution which is the main operational unit of the Refal machine. There are two allowable values for this indicator: '-', a reference variable, corresponding to a left side variable in a Refal sentence, and '+', a place-holder variable, corresponding to a function call occurring in the right side of a Refal sentence.

Reference variables are exactly what their name implies: references to specific values in the information field of the computer. This implies that the value already exists, and may be used freely. A reference variable which appears in the left side of an assignment will be used according to the remainder of its mapping (i.e., data type, status, and location). A reference variable in the right side of an assignment signifies simply that the variable is to take as its value the expression in the left side of the assignment.

In the current implementation, reference variables always refer to two locations in the information field of the computer. This is because the basic data type is the doubly-linked list structure of interpreted Refal. The two locations are the end points of a Refal expression. Each variable in the stack of variables is a header record which points to the two ends of the doubly-linked list. An empty expression is indicated by having the forward pointer point back to the header. A variable of type S is indicated by having both pointers point to the same element of a doubly-linked list (i.e., a symbol is treated as a list with one element).

The place-holder variables represent the results of computations which have not yet been completed. When such a variable appears in the left side of an assignment, it means to save a place for composition of the result of a function call within the surrounding expression. For example, in the assignment

```
( (A B *M(EA+/16/) C) = *M(EA-/3/) )
```

the expression on the left is composed of the letters A and B followed by a place-holder and the letter C. The result is to be given to the variable at location 3. This place-holder variable will be used to set a pointer at stack location 16 to the proper place in the expression on

the left. Location 16 refers to the basis of some function call on the stack of variables. The basis is the first location in the stack of variables used by a function call, and always points to such a place-holder. Thus, earlier in the walk-end we will find some assignment

```
( (...) = *M(EA+/16/) )
```

where ... in this case will be some function call. The place-holder variable in the right side of the assignment means that location 16 is the basis of another function call to be added to the stacks (operating and variable). When the value of this function call is computed, it is substituted for the place-holder in the assignment for the variable at stack location 3, above.

4.5.1.4 Offset

Offset as used in the last indicator of a variable mapping refers to the location of a variable. The term is derived from the current implementation: all variables exist in the stack of variables, and are located by an offset from a pointer in that stack. This stack of variables pointer is always set to the basis of the current function by the stack interpreter. Thus, the mapped variable *M(SP-/3/) may be found by looking at the stack location 3 places away from the basis (and the stack pointer).

In other implementations, or with other data types, this indicator may be used to give actual names to variables if necessary. All that is needed is to set some naming convention, such as: all variables of type integer are to be named 'In', where n is some positive integer, like I34723. Then, the code generation program simply appends the character I to the offset to arrive at the variable's name.

4.5.2 Labels

Labels in the PMAIN program (PMAIN is the name of the LISA machine implemented in Pascal) perform two tasks. First, each function is given a label, which corresponds to its number in the graph of states, and which is used by the stack interpreter to call each function by simply transferring control to that label. Second, branch addresses within a function are also given Pascal labels, and serve to control the proper execution of the condition tests within the function. In addition, a label is given to the final 'END.' in the PMAIN program, and a PMAIN procedure called PIFHALT is declared, which simply transfers control to this final statement when execution by the stack interpreter is completed. The label chosen for the HALT function depends on the number of labels used

previously by the translation program and is indicated in the LISA program by the entity <end-label>.

Because of the syntax of Pascal, it is necessary to generate a list of the labels. For example, in the program from which the previous examples were taken, we have a declaration:

```
LABEL 0,3,4,1,2,5,6,7;
```

Some of the labels declared stand for function names, and some stand for internal branch points within those functions. Finally, once we have the list of all function labels, we can produce the stack interpreter:

```
 1      0: BEGIN (* STACK INTERPRETER *)
 2         PIFINCSTEP;
 3         CASE FUNC OF
 4            1: GOTO 1;
 5            2: GOTO 2;
 6            5: 5: BEGIN PRFPROUT; PIFPOP END;
 7            6: 6: BEGIN PRFCARD; PIFPOP END;
 8            7: PIFHALT
 9         END
10      END (* BLOCK 0 *);
```

In this stack interpreter, we have a CASE statement which chooses among the labels 1, 2, 5, 6, and 7 as functions. Label 0 is always the label of the stack interpreter, and in this case label 7 is used to complete execution of PMAIN. Note on line 2 the procedure statement

PIFINCSTEP. This procedure increments the step counter, and if the program is in debugging mode, invokes the debugging functions. Labels 5 and 6 are special cases which represent the calling of external functions corresponding to Refal system functions /PROUT/ and /CARD/.

4.5.3 Contractions

A contraction in LISA signifies a condition test and the definition of local variables if the test succeeds. If the test fails, a branch is taken. The variable in the left part of a LISA contraction is the one the test is to be performed on, and the expression on the right defines the new variables to be given values if the test succeeds.

The set of contractions allowed corresponds to the elementary contractions found in a graph of states. The reader should refer to section 3.3 for the definition of the elementary contractions. LISA contractions are identical, except that each variable in the contraction is mapped, and the variables in the right part never have the same offset as the one in the left (this is so that the partial variables may be accessed without denying access to the original variable).

Suppose we have the following LISA contraction:

```
(*M(ST-/1/)=(*M(SP-/4/)*M(EP-/3/)))
```

This contraction may have been the result, for example, of the mapping of the following graph of states contraction onto LISA:

```
(e   -> s e )
  1      2 1
```

The meaning of the LISA contraction is: take the variable whose stack offset is 1, whose data type is E (expression in the Refal sense), and whose status is T (total, that is, the variable is in its original input form and has not been contracted), and see if it begins with a symbol. If it does, then assign the value of that symbol to the partial (P) variable at stack location 4 (whose type is S -- symbol), and assign the value of the contracted expression to the partial variable at stack offset 3. These local assignments are not assignments in the true sense. Rather, they mean simply to set the pointers at stack offsets 3 and 4 to the appropriate parts of the original value in the computer's information field pointed to by the variable at offset 1. The pointers in variable 1 still point to the original value in the information field. If the test is not met, then a transfer of execution control will be indicated. The transfer destination will be determined by the context of the tree structure in the current function.

Now, the translation of the LISA statement can be made by application of a translation statement. The general format for a translation statement is:

```
        <LISA statement>
------------------------
Corresponding Pascal code
```

The actual translation statement in the code generation program, of course, consists of a Refal pattern on the left, which determines which LISA statement must be translated, and a replacement expression on the right, which generates the Pascal code. The translation of the above LISA statment, for example, will be:

```
 1    (* (ET-1 -> SP-4 EP-3) ELSE  GOTO 4 *)
 2    TA1 := XS(.XP+1.);
 3    IF TA1@.FOLL@.PTAG <> SYM THEN GOTO 4;
 4    WITH XS(.XP+4.)@ DO BEGIN
 5       FOLL := TA1@.FOLL;
 6       PREC := FOLL END;
 7    WITH XS(.XP+3.)@ DO BEGIN
 8       IF TA1@.FOLL@.FOLL = TA1 THEN FOLL := XS(.XP+3.)
 9       ELSE BEGIN FOLL := TA1@.FOLL@.FOLL;
10       PREC := TA1@.PREC END END;
```

Line 1 in the translation is simply a comment, used in debugging the code generation program itself. It provides a method for tracing which LISA statement produced what Pascal code. Line 2 sets the value of variable TA1 (Temporary Address 1) to the address of the header of the expression at stack offset 1. Each expression type variable is stored in PMAIN (the Pascal environment

for executing the translated program) as a doubly-linked ring, with the stack of variables entry being the header for the ring. Thus, the FOLL field of the expression header, IA10.FOLL, points to the first PIECE (or element) in the expression. Line 3 then tests to see if this element is in fact a symbol. If not, a branch is taken, in this case to label 4 in the PMAIN program. Lines 4 through 10 perform the temporary assignments if the condition is true. Lines 4 through 5 assign the value of the first symbol to the header at offset 4. Line 8 checks to see whether the remainder of the original expression, e1, is empty. If so, then the contracted expression is set to empty, in line 8. Otherwise, lines 9 and 10 set the expression header at stack offset 3 to the ends of the contracted expression. This can be used later in the walk, if necessary.

4.5.4 Assignments

An example of a graph of states assignment, its LISA equivalent, and its Pascal translation are given below. Let's say we have the assignment:

```
( (*M(EA-/1/)) = *M(EA+/0/) )
```

This assignment may have been the result of translation into LISA of the following graph of states assignment, or

it may have been generated internally by the translation program when expanding the assignment chain in some walk-end.

```
(e  <- e )
  1     0
```

We can see in the LISA statement that the assignment is to be made to the variable at stack offset 0, which represents the output of a particular configuration. This variable's role is denoted by a +, which denotes output, and its type is E, for expression. The expression to be assigned to this variable is simply another expression, and it is actual (A), not a ghost. That means that we can take the expression in its entirety for the replacement of the function call by its value. Since the variable *M(EA+/0/) represents the value of a function call, the assignment simply means to replace the function call's place-holder with its value. A PMAIN procedure exists which carries out this function, and its name is PIFREPT (all PMAIN function and procedure names begin with a three letter prefix, in this case PIF stands for PMAIN Internal Function). The translation of this LISA statement then is:

```
1    (* (EA-1 <- E:+0) *)
2    PIFREPT(0,1);
```

Line 1 in the translation is again a comment, and line 2 is the Pascal procedure statement which causes replacement of the function call by its value.

4.5.5 Compiler Directives

In addition to the LISA statements which correspond to the contractions and assignments in the original graph of states, statements are added which pertain to the control of execution of the LISA program itself. These compiler directives fall into two categories. They are garbage collection directives, and interpreter directives. The garbage collection directives tell the execution environment when the value in the information field pointed to by a particular variable may be marked for return to the free memory of the computer, and when a particular value is to be saved for further computation. The interpreter directives indicate to the stack interpreter in what way to update the various stack pointers, and, depending on the implementation, may be used to bypass the stack interpreter entirely. A final specification for a LISA compiler directive, (?<expression>), is left undefined, so that future implementations may expand the set of directives, while continuing to use a standardized format. All directives then can be recognized as being one term, where the first symbol inside the enclosing structure brackets is a question mark.

4.5.5.1 Garbage Collection Directives

The most general garbage collection directive is (?OUT). It indicates to the LISA compiler that all of the input variables may be released and added to the free memory of the computer. Of course, in most cases, some of the input variables to a function call will be used in some manner in the output, either totally or partially. For this reason, the directive (?RV<offset>)) may be used to Peserve a Variable before (?OUT) is specified. For example, suppose we have the following walk in a function definition:

```
(*M(ET-/1/)=(*M(SP-/3/)*M(EP-/2/))
...
((*M(SA-/3/))=*M(EA+/0/)
```

The first contraction defines variables at offsets 2 and 3. The ellipsis indicates other contractions and assignments in the walk. Finally, an assignment appears which makes use of the variable at offset 3. Let us assume that the remainder of the original input variable at offset 1 is not used anywhere in the assignment chain. Then the variable at offset 3 must be saved for composition into the output of the function, while the remainder may be returned to the free memory of the computer. Remember that data types E and S in the current implementation are treated as doubly-linked list structures. Then the variable at offset 3 simply points to one element of the list pointed to by the input variable at offset 1. By adding two garbage collection directives, the element pointed

to by variable 3 is removed from the surrounding list structure (the expression pointed to by variable 1), and the remaining list is "sewn up", so that it may be returned to the free memory list. The required directives are:

```
(?RV(/3/)) (?OUT)
```

The first removes variable 3 from the surrounding expression and relinks the expression remaining; the second returns all input variables to the free memory list (in this case, only variable 1).

The condition chain (the part of a walk which contains only contractions) of a walk defines the partial variables in a tree-like structure. For example, suppose we have the walk:

```
(*M(ET-/1/)=(*M(SP-/3/)*M(EP-/2/)))
(*M(EP-/2/)=(*M(SP-/5/)*M(EP-/4/)))
        ...
((*M(EA-/2/)*M(SG-/5/))=*M(EA+/0/))
```

The garbage collection scheme in the mapping program will then produce the two directives:

```
(?RV(/2/)) (?OUT)
```

Notice that no directive is included for variable 5, and that variable 5 is mapped as a ghost in the assignment. This is because variable 5 was defined as part

of a larger expression which is also used in the output (variable 2). The use of a directive (?RV(/5/)) would have simply removed a part of the expression pointed to by variable 2. Since variable 2 is used in its entirety, any use of variable 5 in the output must be made by copying. This is specified by status G in the mapping of variable 5 in the assignment.

The last garbage collection directive, (?OUTAV(<offset>)), is used in the case where an input variable is used in its entirety in the output. In this case it is not enough to simply "reserve" the partial variables used and then release all the remainders of the input variables. The garbage collection scheme recognizes this condition, and instead of adding one (?OUT) statement to the walk-end, adds one (?OUTAV(n)) statement for each input variable which is not used in its entirety.

4.5.5.2 Interpreter Directives

Interpreter directives are used in LISA to indicate updating of the stack pointers, popping of the operating and variable stacks, and if desired, the bypassing of the stack interpreter entirely. The first set of interpreter directives includes (?BP(<macrodigit>)) and (?XP(<macrodigit>)). The names of these directives are de-

rived from the names of the PMAIN variables which represent the Block Pointer (pointer in the control stack), and eXpression Pointer (pointer in the stack of variables, which in this implementation is a stack of expressions in the Refal sense). The directive (?BP(/3/)) then, tells the LISA compiler to insert the appropriate PMAIN commands for adding 3 to the pointer in the control stack, while (?XP(/8/)) indicates that the pointer in the stack of variables should be incremented by 8.

These directives appear only at the very end of a walk in a LISA program. The walk is translated statement by statement from left to right, so that the offsets of variables in the walk are always referred to the current values of the stack pointers. Once a walk is successfully completed, the stack pointers may be updated.

It is interesting to note that the value of the macrodigit in a (?BP(n)) command will always be one more than the number of function calls to be added to the control stack by the particular walk-end, and that the macrodigit in the (?XP(n)) command will always be one more than the cumulative block lengths (the number of stack locations used for input, plus one for the output) of all the function calls to be stacked. This is so that the next interpreter command will cause the stack interpreter to

properly reset the stack pointers for the next function call. The next directive is (?POP), which simply pops both stacks and begins the next function call, by tranferring control to the stack interpreter.

Typically, then, a walk will end with a sequence like:

```
                ...
((...)=*M(EA+/0/))
(?BP(/2/))  (?XP(/5/))  (?POP)
```

If a walk is entirely passive, that is, it does not include any function calls, then the walk will end with only the (?POP) command; the next function to be called is determined by the stack interpreter.

Two more interpreter directives are included in the current set. They are (?X(<label>)), and (?INCSTEP). The first is the equivalent of a GOTO statement, and it is intended for use in the situation where a walk in a function definition is found to be iterative, instead of recursive. In this case it would be unnecessary to change the values of the stack pointers, and only necessary to return to the beginning of the function definition for the next iteration. The label in the (?X(n)) statement then would be the label designating the function. The (?X(<label>)) directive may also be used in the optimization which bypasses the stack interpreter (see sec-

tion 4.6.5). The second directive, (?INCSTEP) is included so that in the above instance, if so desired, the features of PMAIN function INCSTEP, which increments the step counter, and may also be used for debugging aids on a step-by-step basis, may be called without invoking the stack interpreter.

In order to implement the stack interpreter bypassing optimization it is necessary to add directives which set the current function name and length in the control stack. These directives are: (?F(<function-name>)), and (?L(<macrodigit>)).

4.6 OPTIMIZATIONS THROUGH MAPPING

A number of optimizations to the program represented by a graph of states have been identified. Some of these have been implemented, and some are still in the development stage. The following sections describe those optimizations which have been identified so far.

4.6.1 Stack of Variables Optimization

The first optimization of a graph of states through mapping occurs in the implementation of the "information field" of the abstract LISA machine as a stack of variables. This field represents the contents of the computer's memory.

In a Refal machine, this is called the view field, and is set up as one doubly-linked list structure. In the LISA machine, each variable on the stack represents a structure, in this implementation, a doubly-linked list.

In the process of supercompilation, the pattern expression in the left side of each Refal sentence is broken up into a series of condition tests (contractions and restrictions). The result is that a function call in the supercompiled program appears with the format:

*(<config-number> <arg-list>)

where <config-number> is the identifier of the function (or configuration of the Refal machine), and <arg-list> is a Refal list of expressions, defined by:

<arg-list> ::= <empty> | (<expression>) <arg-list>

If a function call with such a format were to be translated back into Refal, then, the Refal system would simply pair up the parentheses surrounding each expression (the actual value to be passed to a particular formal parameter of the function call), and treat the entire list of expressions as one expression. For example, the following Refal function is a translation of one function definition in a graph of states back into Refal:

```
C8  (E1)(S2)(E3)(E4)(S2E5) = (E1)S2E3
  (E1)(S2)(E3)(E4)(S6E5) = K/C8/(E1)(S2)(E3)(E4S6)(E5).
  (E1)(S2)(E3)(E4)((E6)E5) = K/C8/(E1)(S2)(E3)(E4(E6))(E5).
  (E1)(S2)(E3)(E4)() = K/C7/(E4)(E1)(S2E3).
```

Notice that the left side of each sentence consists of 5 sets of paired parentheses, each pair containing some pattern expression. The parentheses serve only to separate one expression from the next. The Refal machine, upon encountering a call for function C8, must go through the process of recognition for each pair of parentheses, even though they are not actually part of the input to the function proper. By implementing the LISA machine as one with an information field which consists of a stack of variables, the recognition process to be performed on the formatting parentheses can be deleted. This means that each pair of parentheses delimits the value to be given to one variable in the stack at the time of invocation of the function.

When a LISA program is compiled then, the formatting parentheses are compiled out. Each LISA function call assumes that the proper number of stack locations have been previously filled by some other function calls. Thus, the recognition process for the first sentence in the preceding example would consist of only two contractions: one which determines if the 5th variable starts with a symbol, and a second which checks to see if the value of

that symbol and the value of variable 2 (also a symbol) are the same. If both contractions succeed, the replacement indicated by the right side (walk-end in the LISA program) is executed.

4.6.2 Iteration in place of Recursion

The mapping from the graph of states to LISA may be used to determine whether a function is recursive or iterative. In the case of an iterative function, the LISA program could be tailored so that the code generated from it, in the target language, makes use of the iterative properties of the function, and prevents recursive function calls. This would greatly reduce the overhead for execution of the function in question.

Suppose we have the following function in Refal:

```
FA 'A' EX = 'B' K/FA/ EX.
 SY EX = SY K/FA/ EX.
 =
```

This function simply scans the argument, which is assumed to consist only of symbols, and replaces every 'A' with a 'B'. Of course, the way the function is written, execution is recursive. Iteration cannot be represented in the graph of states as such, but the translation into LISA can change the function definition to an iterative one. The above function, after supercompilation, and translation back into Refal, may appear as:

```
C2  ('A'E1) = 'B'K/C2/(E1).
 (S2E1) = S2K/C2/(E1).
 () =
```

The corresponding translation into LISA may be as in figure 7, below.

```
 1   /2/:(
 2        (
 3         (*M(ET-/1/)=(*M(SP-/4/)*M(EP-/3/)))
 4         :(
 5            (
 6             (*M(SP-/4/)=(A)) (?RV(3)) (?OUT)
 7                [ (*M(EA-/3/))=*M(EA-/1/) ]
 8                [ (*(/2/(*M(EA-/1/))))=*M(EA+/C/) ]
 9                [ (B*M(EA+/0/))=*M(EA+/0/) ]
10                 (?BP(/1/)) (?XP(/2/)) (?POP)
11            )
12       /3/+
13            (
14             (?RV(4)) (?RV(3)) (?OUT)
15                [ (*M(EA-/3/))=*M(EA-/1/) ]
16                [ (*(/2/(*M(EA-/1/))))=*M(EA+/0/) ]
17                [ (*M(SA-/4/)*M(EA+/0/))=*M(EA+/0/) ]
18                 (?BP(/1/)) (?XP(/2/)) (?POP)
19            )
20          )
21        )
22    /4/+
23        (
24         (*M(ET-/1/)=()) (?OUT)[ ()=*M(EA+/C/) ](?POP)
25        )
26       )
```

Figure 7: Recursive function in LISA

The iterative quality of each walk may be determined by the fact that each one contains at most one active as-

signment, and that assignment is always a call for function 2, the one being studied. In this case, recognition is simple, since each walk deals only with "breaking off" one symbol on the left, or determining if no symbols are left (empty expression). We might modify the LISA program by deleting the active function calls in the first two walks, deleting the interpreter directives in both, and adding the appropriate LISA GOTO statements at the end of each walk, as in figure 8, below.

```
 1    /2/:(
 2          (
 3           (*M(ET-/1/)=(*M(SP-/4/)*M(EP-/3/)))
 4           :(
 5              (
 6               (*M(SP-/4/)=(A)) (?RV(3)) (?OUT)
 7                  [ (*M(EA-/3/))=*M(EA-/1/) ]
 8                  [ (B*M(EA+/0/))=*M(EA+/0/) ]
 9                  (?INCSTEP) (?X(/2/))
10              )
11         /3/+
12              (
13               (?RV(4)) (?RV(3)) (?OUT)
14                  [ (*M(EA-/3/))=*M(EA-/1/) ]
15                  [ (*M(SA-/4/)*M(EA+/0/))=*M(EA+/0/) ]
16                  (?INCSTEP) (?X(/2/))
17              )
18            )
19          )
20     /4/+
21          (
22           (*M(ET-/1/)=()) (?OUT)[ ()=*M(EA+/0/) ](?POP)
23          )
24        )
```

Figure 8: Iterative function in LISA

This optimization has been implemented, and in a short program shows significant execution speedup. See chapter 7.2 for further discussion.

4.6.3 Actual/Ghost optimization

A variable in LISA is a set of pointers to an area within the information field of the computer. The distinction between actual and ghost variables is this: an actual variable points to some part of the information field which may be modified, while a ghost variable points to some part of the information field which may not be modified. This distinction is useful when it is required to perform some nondestructive test on the value in the information field; with a ghost variable this may be done without copying the value (that is, the value of the actual variable).

Figure 9 shows two variables, one at offset 1, whose status is A, and one at offset 17, whose status is G. The variable at offset 1 may be used for composition into some other expression, while the one at offset 17 may only be used for testing. If it is requred to compose the variable at offset 17 into some expression, then the value must be copied.

Consider the following Refal program:

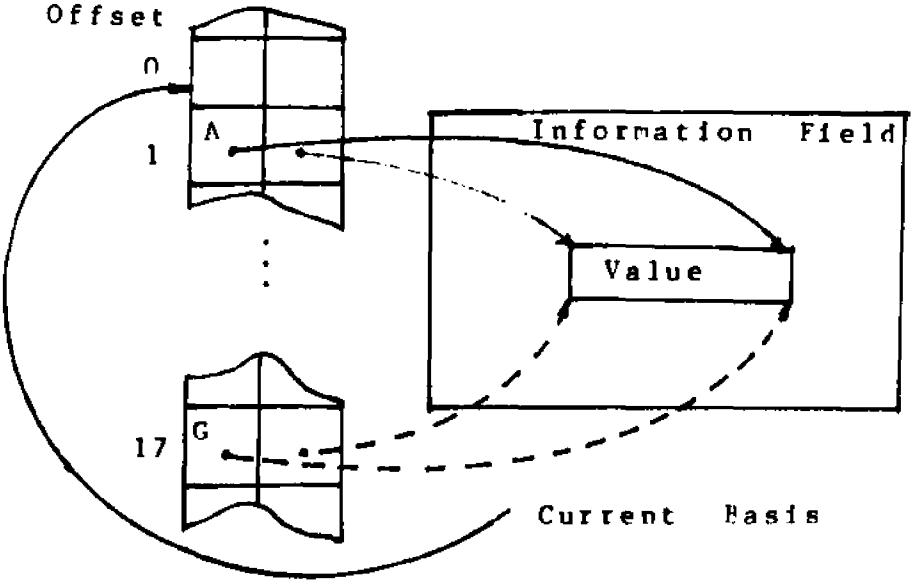

Figure 9: Actual/Ghost mapping

```
1      F0 EX = K/F1/ EX K/F2/ EX..
2      F1 E1 F = K/F3/ E1.
3       E1 T =
4      F2 A EA = I
5       S2 EA = K/F2/ EA.
6       = F
7      F3 ES = ...
```

In line 1, the result of concretization of function /F0/ is seen to produce a copying of its argument, EX. However, investigation of functions /F1/ and /F2/ reveals that no matter what function /F2/ does, it always throws away its argument, which in this case is the same argument which is needed by /F1/. Let the argument to /F2/ be a ghost variable, pointing to the same expression that /F1/ has as part of its input. Then /F2/ can make all the condition tests necessary without modifying the original value in the information field.

In the current implementation, the only use of ghost variables is when copying of a value is necessary. However, the means exist for implementation of this optimization.

4.6.4 Deletion of local variable definition

A contraction in LISA represents a condition test followed by definition of local variables. In many cases, the only part required will be the condition test. In these cases, the definition of the local variables is superfluous, and may be deleted.

In order to determine the need for definition of a local variable, it is necessary to trace the subtree of a function definition following each contraction, determining whether each local variable is used in the output. If not, the contraction in which the local variable is defined may be marked appropriately, for example by creating a new status "X". It should be noted that only variables whose original status is P are eligible for this optimization, since they are the ones defined locally. Variables with status T are the formal parameters to a function call, and as such are already defined, upon entry to a function.

4.6.5 Bypassing the stack interpreter

In the first implementation of LISA, all walks end with a call to the stack interpreter. If a walk ends with no active assignments (i.e., function calls), then it is necessary to do this. However, if there are a number of stacked function calls at the end of a walk, the first implementation sets both stack pointers to one more than the number of locations added to the stacks, and then pops both stacks, and calls the stack interpreter. This is the simplest form of compilation of a LISA program, and produces considerable overhead. It is qute simple instead to set the pointer in the stack of variables to the lowest new basis, and the pointer in the control stack to the lowest new function call, and then to "GOTO" the function indicated, bypassing the popping step. This optimization is easy to perform after the LISA program is already prepared. The result of this optimization can be seen in the chapter on conclusions, chapter 7.2.

4.6.6 Variable data types

As mentioned previously, the current implementation allows only two data types, E and S. However, it is possible to define new data types which are not as general, which may be assigned to certain variables if the

proper conditions are met. If a different data type is assigned to a variable, of course, it will be necessary to create the background functions which deal with the new data type, and the translation statements for producing the target language code from the LISA statments dealing with variables of the new type.

The derivation of data types may be performed in the first mapping phase. At that point, a "state-list" of input and output variables exists. This state list is simply a list of formats for each function, showing the number and general (i.e., type E or S) type of each variable in a call to the function, along with the general type of the output. By applying algebraic rules to the state list and to each variable in the as-yet unmapped graph of states, it is possible to determine whether a specific data type is allowable for each variable, or whether the most general E and S types are necessary.

The derivation of a variable's data type may be performed in the following manner:

1. Collection of requirements. This is a determination of what operations are performed on each variable in the unmapped graph of states. A list of the requirements of these operations is stored for each variable appearing in the graph of states.

2. Deduction of the least general data type allowed for each variable, using the requirements collected in 1) above, along with the data types of the program's input and output (which is part of the input data to the mapping process). This is performed by comparing the requirements for each variable against a list of allowed data types, in order of increasing generality. The occurrence of one requirement for type E for a specific variable, for example, overrides any less general data type specification.

The scheme for derivation of data types described above has been implemented for the two simple data types character and string. A test of a simple Refal program which allows the assignment of these data types to the program's variables is described in chapter 8.

5. TRANSLATION OF GRAPH OF STATES INTO LISA

5.1 OVERVIEW OF THE MAPPING PROCESS

Phase one of the mapping scheme, the translation of the graph of states into a LISA program, demands that a number of transformations be carried out on the graph of states. This section gives a short descrip-

tion of these requirements. More detailed discussions of these processes may be found in sections 5.2 through 5.4.

5.1.1 Modifications to State List

The Refal Supercompiler at present gives two sets of output data. They are the graph of states of a program, and what is called a **state list**. The state list is essentially a list of function call formats with their formal parameters, for each function call in the supercompiled program. It is used in the mapping process in order to determine data types, actual/ghost status, etc.

The state list as received from the supercompiler contains some essential information, and some which is not needed for mapping. It can thus be modified to contain just what is needed, and to add some information which will be used by the mapping process itself, such as for example, the number of arguments in a specific function call, whether or not the function call has been mapped yet, etc.

5.1.2 Labelling and Expansion of Graph of States

The graph of states must be transformed in some way into a format which lends itself more readily to the structure of a conventional programming language. For example, branch points in a graph of states are not labelled; most

conventional languages require labels of some sort. In addition, the graph of states is written in an encoded format, in order to save memory space during supercompilation. For purposes of translation into LISA, and then further translation into a target language, it is convenient to use an expanded format, which specifies each metacode A variable in its entirety. The labelling and expansion processes work in parallel.

5.1.3 Renumbering of Input Variables

Variable indices are redefined in contractions in a graph of states. For example, we may find a contraction:

```
(e   -> s e )
  1      2 1
```

If e1 is used in the left side of an assignment in the <walk> in which it appears, it refers to the contracted value of e1, not the original. For this reason, it becomes necessary to renumber the graph of states, so that each contracted value can be accessed when performing the replacement (or assignment of output values) of the left side of the walk with the right side.

5.1.4 Decomposition of Output Assignment

The graph of states as produced by the supercompiler contains just one assignment at the end of each walk. This assignment may contain active elements (i.e. function calls), or may simply require decomposition into a number of assignments which represent the passing of arguments to higher level function calls. The decomposition process yields a chain of assignments, without necessary regard for mapping each active assignment's formal parameters onto the stack of variables. The active assignments will be replaced again in the expansion process (see section 5.1.5, below), and will finally receive their stack mappings when the variable indices are replaced by stack offsets (see section 5.1.6, below).

5.1.5 Expansion of Active Assignments

Once the single output assignment has been expanded into an assignment chain, each active assignment in the chain must be replaced with a number of passive assignments and compiler directives. The passive assignments represent the passing of actual parameters created during the operation of a configuration to the appropriate stack locations for each particular function call specified as an active assignment. The compiler directives

are LISA statements which stack the function (or configuration) labels on the operating stack of the target program, and tell the target program how many stack locations to reserve for each function. Additional compiler directives are added before each assignment chain to indicate which dynamic variables can be returned to the system.

5.1.6 Replacement of Variable Indices by Stack Offsets

Each variable in the original graph of states has an index which identifies it uniquely within the configuration in which it appears. After the above-mentioned renumbering process, each variable still has a unique index, although it may differ from its original value. In order to place each variable at a specific location in the stack of variables of the executing target program, the indices must be replaced with stack offsets. These offsets must be chosen in such a manner that each variable has a unique location, and so that when the replacement or assignment process occurs, no variables are lost.

5.1.7 Actual/Ghost mapping

A variable may appear more than once in the output chain of assignments. It may also appear as an argument to a

function call which implies that the variable may not be actually needed until after the function call, or maybe not at all, depending on the result of the function call. For this reason, the concept of Ghost variables has been developed. This concept has the potential for great optimization of mapped programs. See section 4.6.3 for an example of actual/ghost mapping.

5.1.8 The Garbage Collection Scheme

As the value of each variable in a LISA program is used, a determination must be made as to whether that value will be needed further or not. If not, the value (i.e., the area in the computer's information field pointed to by the variable) may be returned to the free memory list of the computer for reuse. A number of compiler directives have been developed for this end (see section 4.5.5.1).

5.2 LABELLING OF THE GRAPH OF STATES

The first task of the mapping program is to insert labels into the definition of each function, at each branching point. This in itself is a straightforward process, and can be viewed as the initial transformation of the purely functional format graph of states into a sequential

format. The labels make the (modified) graph sequential in that each branching operation within the graph has a named (i.e., labelled) destination, and so each instruction or statement in the graph may be executed in sequence.

Each function in the graph of states is given a name, which is also a label. A problem arises when dealing with external functions. It is therefore the task of the labelling routine to assign labels for each external function used by the program, and to insert the appropriate instructions into the list of function names so that each external function will be recognized as such when the LISA program is compiled into the target language.

5.3 THE FIRST MAPPING PHASE - /MAP1/

After the initial labelling and reformatting processes are completed, we are ready to begin the mapping proper. The first phase consists of one left to right scan of the graph of each function, with a number of tasks being performed. Function /MAP1/ acts as the main control function for this first phase, and calls a number of specialized functions as each element of the graph is encountered. The necessary arguments for /MAP1/ are: 1) the configuration (i.e., function) number; 2) the maximum stack length of this function (which may be modified in the

course of this first phase); 3) the input block length of the function (i.e., the number of arguments to the function, plus one more stack location for the pointer to the output destination); 4) the state list; 5) a stack of variable renumberings; and finally the graph of this function itself. The function /MAP1/ makes a depth-first scan of the graph, using the multibracket technique detailed in [1:31-37].

The stack of variable renumberings is updated upon encountering each contraction in the graph, and in addition, each time a branch point is entered or exitted. Upon entry to a branch point, a new level is added to this stack. A level is signified by an equal sign, a macrodigit representing the next available index number, and a pair of structure brackets. Each time a contraction is encountered, the renumberings of the variables are added after the equal sign and next available index number for this level as a triple: # I N, where I is the original index of the variable in the graph of states, and N is the new index. The new indices are given out sequentially, starting with the next available index after the number of input variables. At the same time, the contraction itself is renumbered, and a copy of the contraction is added to the list of contractions at this level, in the pair of structure brackets at the end of this stack level. This list

of current contractions will be used later, when /MAP1/ reaches the walk-end, in order to perform conglomeration (see function /CON/, in section 5.3.2.2). Finally, when a branch is completed by /MAP1/, the next branch is entered after 'popping' the stack of renumberings. That is, the last level is removed, in order to allow for the different contractions needed to reach the next branch in the graph.

Suppose that the initial part of a graph of states of a function with one input variable looks like this:

```
:( (
     (*E/1/=(*S/2/*E/1/)) (*E/1/=(*S/3/*E/1/))
                                  :(  ...
```

Then, initially, the stack of renumberings would look like: (=/2/()). The macrodigit /2/ signifies that 2 is the next index available for renumbering. Upon encountering the first branch point (the second right bracket in the first line), a new stack level is added, thus: (=/2/()=/2/()). Notice that intially the next available index is the same as that of the last level. Next, the first contraction is encountered. It is renumbered thus: (*E/1/=(*S/3/*E/2/)). Notice that *E/1/ in the left part is not renumbered at this point, because it represents a total input variable. However, *E/1/ in the right part of the contraction is renumbered, since it represents a

contracted or partial input variable. The stack of renumberings is updated thus:

```
( =/4/#/1//2/#/2//3/ ( (*E/1/=(*S/3/*E/2/)) ) =/2/() )
```

Each equal sign at the highest structure level in the stack represents one level within the graph, and each number sign gives one renumbering. Upon encountering the second contraction, additional renumberings will be added to the highest level. At this point however, *E/1/ in the left part of the second contraction is renumbered, before the renumbering of *E/1/ in the right part is performed, since this entry of *E/1/ represents the already contracted value produced by the first contraction. Therefore, the second contraction is renumbered thus: (*E/2/=(*S/5/*E/4/)), and the stack of renumberings is updated to:

```
( =/6/#/1//4/#/3//5/#/1//2/#/2//3/ (
     (*E/1/=(*S/3/*E/2/)) (*E/2/=(*S/5/*E/4/)) ) =/2/() )
```

Notice that now there are two renumberings in the current stack level for index /1/. Since the Refal system always matches patterns by scanning from left to right, the leftmost renumbering will subsequently be chosen. It is not necessary to remove the earlier renumbering, and its removal would just necessitate an additional scar of the

stack before proceeding to the next step. Therefore, the previous renumbering is left in the stack and ignored.

The actual renumbering of the contractions is performed by function /NXOP/, for Number and eXpand OPerations. The expansion process is necessary because the supercompiler produces contractions in an abbreviated code. Each contraction then is expanded by /NXOP/ into the metacode A format given above, and is renumbered. Restrictions, the other kind of condition, have already been removed by the labelling process before entry to /MAP1/.

Notice that as each new level is passed in the graph of states, a new stack level is added on the left of the stack. This is so that upon exitting from that graph level, the last stack level may be 'popped' simply by removing everything in the stack before the second equal sign. The number signs are necessary for each renumbering, in order to ensure the choice of the correct index as the intial one, and the correct one as the new one.

Upon reaching the first assignment in a walk, /MAP1/ passes control to function /WEP1/, which is the main control function for the first of two passes over the walk-end beginning with this first assignment. /WEP1/ performs the following tasks: 1) renumbering of all in-

put variables appearing in the left sides of the assignments in the walk-end, according to the renumberings accumulated by /MAP1/ (performed by function /NXA/, see section 5.3.2.1); 2) conglomeration of input variables in output expressions, performed by function /CON/; 3) decomposition of the single assignment produced by the supercompiler into a series of stack assignments; 4) calling of the second walk-end processing function, namely, /WEP2/.

Function /WEP2/ takes the walk-end after processing by /WEP1/ and performs the following tasks: 1) creates temporary stack variables when it is necessary to prevent erasure of a variable by overlaying during execution of the resulting program (performed by function /TMP/); 2) updates the maximum stack length of this function; 3) returns to /MAP1/ for processing of the next walk.

5.3.1 Numbering and Expansion of Operations - /NXOP/

The numbering and expansion of conditions (i.e., contractions), as mentioned above, is performed by function /NXOP/. This function is a straightforward selection process, which simply selects the proper contraction type from a list of encodings (corresponding to the codes produced by the supercompiler), and produces the metacode A representa-

tion of the contraction, after renumbering the variables. Function /NXOP/ determines whether or not the variable in the left side of the contraction must be renumbered, and then passes the contraction to an auxiliary function which renumbers the right side of the contraction. The only noteworthy part of this process is the way type 5 contractions (e.g., (*S/1/=(*S/4/))) are handled. Since in this type of contraction, either the variable on the left, the variable on the right, or both, may be input variables, two checks must be made for what must be renumbered, instead of just one.

It should be noted that after renumbering by /NXOP/, the variables in each contraction are indexed, that is, the variable numbers are still not stack offsets. Of course the indices corresponding to total input variables will eventually get offsets identical to their indices, but all partial variables will ultimately get offsets which may be different, depending on the final value of the maximum stack length for this configuration. Thus the input variables are still not yet mapped, but only renumbered to reflect their partial nature. The final mapping of each input variable occurs during processing by function /MAP2/ (see section 5.4).

5.3.2 First pass over the walk-end - /WEP1/

The first pass over the walk-end performs a number of tasks. Remember that at this point, each walk-end consists of just one assignment, as created by the supercompiler program. The purpose of this first pass then consists of:

- Renumbering of input variables appearing in the left side of the single output assignment, to correspond with the renumberings created in this walk by functions /MAP1/ and /NXOP/. This is performed by function /NXA/ (see section 5.3.2.1, below).
- Conglomeration of partial input expressions into their original total expressions, where possible. This is performed by function /CON/ (see section 5.3.2.2, below for a more detailed explanation of this process).
- Decomposition of the single assignment received from the supercompiler program (as modified by the above mentioned functions) into a chain of output assignments, where the right side of each is a fully mapped variable giving a real stack offset (variable location). See section 5.3.2.3, below.
- Passing of the result to the function performing the second pass over the now-expanded walk-end, namely function /WEP2/ (see section 5.3.3, below).

5.3.2.1 Renumbering of variables in the walk-end - /NXA/

The first task performed by the first walk-end processing function is the renumbering of the input variables appearing in the single assignment created by the supercompiler. This process is performed before expansion of the assignment into a stack assignment chain simply for the sake of efficiency. This process is carried out by function /NXA/. The actual arguments for /NXA/ are the stack of renumberings created in /MAP1/ and the walk-end itself. The remainder of the graph of this function is saved outside the range of concretization of this function when /WEP1/ is called, and is restored in the proper multibracket format upon completion of /NXA/ by an auxiliary function called /NXAF/.

Function /NXA/ performs a simple left-to-right scan of the expression in the left part of the single assignment, using the multibracket technique to search all levels of this expression, and replaces all occurrences of renumbered variable indices with the new indices. These indices are simply taken from the renumberings in the stack of variable renumberings created by /MAP1/. For example, suppose that part of the stack of variable renumberings included the following sequence:

```
( =/6/*/1//4/*/3//5/*/1//2/*/2//3/ (
     (*E/1/=(*S/3/*E/2/)) (*E/2/=(*S/5/*E/4/)) ) =/2/() )
```

and also suppose that the single output assignment is:

((*S/2/*S/3/*E/1/ABC*(/6/(*E/1/)(*S/2/))) =*E/0/)

This assignment means: the output of this walk should be composed of variable *S/2/ concatenated with variable *S/3/, concatenated with variable *E/1/, concatenated with the string 'ABC', and finally concatenated with the result of evaluation of function (configuration) 6 on copies of *E/1/ and *S/2/ as its arguments. Function /NXA/ scans left-to-right through the expression in the left part of the assignment, and each time it finds a variable, checks to see if a renumbering exists for that variable. For example, upon encountering *E/1/, /NXA/ recognizes that the last renumbering of this variable is #/1//4/. Thus, *E/1/ will be replaced by *E/4/. Continuing in this fashion, we arrive at the renumbered assignment:

((*S/3/*S/5/*E/4/ABC*(/6/(*E/4/)(*S/3/))) =*E/0/)

Note that there may not be a renumbering for a particular variable in the stack of renumberings. This implies that the variable is a total input variable which was not contracted at all in the walk being processed. In such a case, the variable is left alone. Also note that *E/0/ in the right side of the assignment is left alone, as this variable always represents the output of one walk.

Upon completion of this renumbering scan, control is passed to function /CON/, described below in section 5.3.2.2. At this point, the walk-end still consists of one assignment, but all the variables in the left part of that assignment have been renumbered to correspond with the renumberings in the condition chain leading to this walk-end.

5.3.2.2 Conglomeration - /CON/

At this point, it is possible to search the single assignment for subexpressions which correspond to the right sides of contractions in the condition chain leading to the walk-end. If such subexpressions are found, they can be replaced by the original uncontracted variable in the left side of the contraction in which they were created. This is the reason that each level in the stack of renumberings also includes a list of contractions found at that level. This task of conglomeration is performed by searching the left side of the output assignment for occurrences of the right side of each contraction in the condition chain preceding the walk-end. When a match is found, replacement of the subexpression by the variable from the left side of the contraction is performed. Contractions are back-substituted in this manner in the reverse order from which they occurred in the condition

chain, in order to ensure that all possible conglomerations are performed.

For example, in the example of the previous section, the subexpression *S/5/*E/4/ occurred in the single output assignment. By searching the (renumbered) contractions stored in the stack of variable renumberings, we find the contraction (*E/2/=(*S/5/*E/4/)). We can then replace the subexpression *S/5/*E/4/ with the variable *E/2/, so that the assignment becomes:

((*S/3/*E/2/ABC*(/6/(*E/4/)(*S/3/))) =*E/0/)

Notice that now, we find the subexpression *S/3/*E/2/ in the assignment. This corresponds to the right side of the contraction (*E/1/=(*S/3/*E/2/)). Thus, we can now replace this subexpression with *E/1/, to result in the assignment:

((*E/1/ABC*(/6/(*E/4/)(*S/3/))) =*E/0/)

It is important to understand the significance of the conglomeration process. By replacing subexpressions by their antecedents, the complexity of the output assignment is reduced. However, there is also at this point another meaning. Each contraction in the condition chain represents a condition test followed by the creation of local or partial variables. If after con-

glomeration, the partial variables do not appear in the resulting expression, it is possible to conclude that only the condition test part of the contraction is necessary. It may then be possible to perform an optimization in the code generation phase which recognizes this situation, and deletes the creation of the specific local variables.

Function /CON/ makes use of the renumbered contractions saved in the stack of renumberings. For each walk-end entered, the renumbered contractions will be different, representing a different condition chain followed to arrive at that walk-end. Before /CON/ can perform the conglomeration process, then, an auxiliary function /CONC/ scans the current stack of renumberings and extracts the appropriate contractions for this walk-end.

When function /CON/ is finished with the walk-end, it passes the result to function /DE/, the main decomposition function for the walk-end. Just before this, function /CON/ maps the variable in the right part of the single assignment, since it is already known that this variable is *E/0/, representing the output of the walk-end. The variable is mapped as *M(CA+/0/), which in the current mapping scheme is always the output of a walk.

5.3.2.3 Decomposition - /DE/

The next step in the first pass over the walk-end is the decomposition of the single assignment as produced by the supercompiler into a chain of assignments with mapped variables on the right side. Each mapped variable in this step will represent one stack location, and the indices of each mapped variable will correspond to this stack location. The expression in the left side of the assignment is assumed to have already passed the renumbering and conglomeration processes. Function /DE/ is the formatting function for decomposition, which in turn passes control to function /DEX/, its auxiliary. /DEX/ always looks at the rightmost assignment in the developing assignment chain, and determines whether or not it is necessary to decompose it in any way. If not, the assignment is considered completed up to this point, and is saved with any previously completed assignments, in their original order. If it is necessary to decompose the assignment, control is passed temporarily to one of two auxiliary functions, /PLH/ and /FP/. Upon completion of the processing by either /PLH/ or /FP/, control is returned to /DEX/, for the next assignment. When the assignment chain is completed, it is passed to function /WEP2/ for a second pass, and then, finally, control is returned to /MAP1/.

Of course, upon initial entry to /DE/ and /DEX/, the assignment chain consists of just one assignment. If this assignment requires no decomposition, control is passed to /WEP2/. Otherwise, the single assignment is decomposed, forming the beginnings of an assignment chain. The result of decomposition of that first assignment is then stored, and the newly created assignments are checked again by /DEX/. Processing by /DEX/ continues until all assignments have been decomposed appropriately.

Remember that the single assignment as received by /DE/ and /DEX/ already has a mapped variable on the right. /DE/ then determines whether the single assignment is composite or not. The term composite is used in this context to mean an assignment in which the expression on the left consists of more than one term (i.e., a complex expression in which more than one variable, symbol, or active element appears). If the assignment is composite, then, function /PLH/ is called, to replace active elements with place-holders, and to create new assignments representing the expressions these place-holders replace. Control is then passed to /DEX/ for the remainder of the assignment chain. If the assignment is not composite, meaning it consists of exactly one function call, control is passed to function /FP/, which assigns stack locations and

mappings for the formal parameters in the function call, and creates new assignments which relate the actual to the formal parameters. Control is then passed to /DEX/ for the remainder of the assignment chain.

Once the "initialization" described above is performed by /DS/, function /DEX/ operates on any assignments remaining which have been created previously. /DEX/ recognizes a number of conditions, and responds accordingly. These conditions represent classes of assignments, and they are:

1. Simple assignments which contain just one variable in the left side. These are of the form ((*TI)=*M(DSRO)), where T is the variable type, and I is its index. The left side consists of one mapped variable, with the usual fields. This kind of assignment will have been previously created or passed by functions /PLH/ or /FP/. The result is to simply add the assignment to the chain of completed assignments.
2. Active assignments created previously by /FP/. This kind of assignment takes the form ((*(N<var-list>))=*M(DS-O)), where N is the function number (a macrodigit), and <var-list> is the list of actual parameters for this function call. Note that the role field in the mapped variable on the right has the value '-', which is used by /DEX/ as an

indication that the assignment was created previously by /FP/, and must be assigned an appropriate stack location for its basis (i.e., place-holder for the function's output). In addition, an assignment must be created which relates this basis to the stack location receiving the result of the function call. For example, the assignment:

```
((*(/6/(*E/1/)(*S/2/)))=*M(EA-/4/))
```

might be replaced by two assignments:

```
((*(/6/(*E/1/)(*S/2/))=*M(EA+/10/))
((*M(EA+/10/))=*M(EA-/4/))
```

The first assignment is added to the left end of the list of uncompleted assignments, so that it will be processed later by /FP/. The second assignment is complete, and simply says that the result of evaluation of the function call with basis at stack offset 10 should be linked with the variable at stack offset 4. This linking means that as the target program operates, the function call with basis at stack offset 10 will be executed first, and when execution of the function which has stack offset 4 as one of its arguments is performed, the result from the call of function 6 will be in the proper location.

3. Active assignments with role '+'. These assignments are those created previously by /DEX/ in the process described in the preceding point. When such an assignment is encountered, control is transferred to function /FP/, which produces the assignments relating actual to formal parameters. Upon return to /DEX/, this assignment will have only mapped variables in its <var-list>, and the assignment will be considered completed.
4. Any other assignment. This implies that the assignment has a complex expression on the left, which may contain active elements. The assignment is passed to function /PLH/, which finds any active elements and creates place-holders for them. Once the place-holders are inserted in the appropriate places in the left side of the assignment, the assignment is considered complete. For each place-holder, a new assignment is created, which is added to the list of uncompleted assignments. The new assignments will be processed further by /DEX/.

It should be clear from the above discussion that in addition to simply passing each assignment to the appropriate function, /DEX/ must also assign stack offsets for each new function call added to the resulting assignment chain. Stack offsets for the actual parameters to each func-

tion call are assigned by /FP/ and /PLH/. In order to do this, a number of local variables are used by /DEX/, /PLH/, and /FP/. These are the next available stack location for a basis, and the current number of function calls already stacked. In addition, the current maximum walk-end length is saved, so that it can be compared to the length of the walk-end being processed.

When an assignment of the class described in point 2 above is encountered, /DEX/ assigns the next available location for a basis to this assignment (corresponding to the /10/ in point 2 above), and calculates the next available location for a basis. This calculation is performed by searching the state list for the length of the function in question, and adding this number to the previous basis location.

Finally, when the entire assignment chain is complete, /DEX/ performs one more task before passing control to /WEP2/. This is the determination of whether the walk-end has any active assignments present. The presence of active assignments is indicated by the value of the next available location for a basis, and the number of blocks in the walk-end. A value of zero for both of these local variables indicates that the assignment chain is entirely passive. In this case, the assignment

chain is completed by adding the LISA compiler directive (?POP), which tells the target program to pop the operating stack. If either variable is not zero, the LISA directives for incrementing both the operating stack pointer and the pointer in the stack of variables are added before the pop command. In addition, in this case, the walk-end length is compared with the previous maximum walk-end length, and the maximum of the two is taken.

Functions /FP/ and /PLH/ work in parallel with /DEX/, using the same local variables. They are described in sections 5.3.2.4 and 5.3.2.5, below.

5.3.2.4 Relating Actual to Formal Parameters - /FP/

Function /FP/ is responsible for taking an active assignment found by /DEX/, and creating one assignment for each formal parameter. It uses the offset in the variable in the right part of the assignment to set the offset for the first new assignment by adding one to it. Then, it separates the variable in the right part from the list of actual parameters, and passes control to an auxiliary function /FPA/. No change is made either to the next available location for a block, or to the number of blocks in this walk-end, since it is assumed that the offset of the variable in the right part of the assignment is map-

ped correctly. All assignments created will have a value of '-' for the role field of the mapped variables representing their respective stack locations, so that if this part of the decomposition results in an assignment with exactly one active element in the left part, the role field will be changed to a '+' when /DEX/ assigns a new location for a basis.

/FPA/ then is a function which simply creates one assignment for each formal parameter. Its arguments are: 1) the next parameter's offset; 2) the list of formal parameters for this function, taken from the state list; 3) the mapping of the variable in the right side of the original assignment; 4) the list of actual parameters, taken from the original assignment; 5) the list of completed assignments; 6) a list of dummy arguments for the reconstructed assignment. /FPA/ simply matches each actual parameter with the corresponding formal parameter, and creates an assignment for each such pair. At the same time, the actual parameter in the original assignment is replaced with a dummy argument, thus reconstructing the assignment. Also, as each assignment is created, the next parameter offset is incremented.

As each pairing of actual to formal parameters is made, an auxiliary function, /FPOFF/, is called to re-

place the offset of the formal parameter taken from the state list with the offset to be assigned by /FPA/. Finally, when the lists of parameters are exhausted, /FPA/ reconstructs the original assignment, places it in the list of completed assignments, and returns control to /DEX/.

For example, suppose /FP/ receives the following assignment:

```
((*(/18/(*S/1/)(*E/2/*S/8/)(*(/113/(*E/7/)))))=*M(EA+/0/))
```

Assume also the following state list mappings for functions /18/ and /113/:

```
*M/113/(TYPE)/2/((*M(EA-/1/))=*M(EA+/0/))
*M/18/(C18)/4/((*M(SA-/1/))(*M(EA-/2/))(*M(EA-/3/))=*M(EA+0))
```

In this mapping, function /18/ takes three arguments, represented by formal parameters *M(SA-/1/), *M(EA-/2/), and *M(EA-/3/). Function /113/ is an external function representing the Refal function /TYPE/. It has one argument, *M(EA-/1/).

Function /FPA/ then creates lists of formal and actual parameters for function /18/ from the state list and from the actual assignment. After performing the pairing described above, we arrive at an assignment sub-chain:

```
((*(/113/(*E/7/)))=*M(EA-/3/))
((*E/2/*S/5/)=*M(EA-/2/))
```

((*S/1/)=*M(SA-/1/))

Notice that the first (leftmost) assignment contains just one active element on the left, and a role of '-' in the mapping of the variable on the right. It will subsequently be processed by /DEX/, in order to give it a stack offset for its basis, and then by /FP/, to give an assignment relating the actual parameter of function /113/ (*E/7/) to the formal parameter (as yet unknown). The remaining two assignments should be self-explanatory. These three assignments will be added to the right end of the chain of assignments to be further processed by /DEX/. The first to undergo processing will be the one with the lowest offset.

In addition, the original assignment will take the variables from the right sides of the new assignments, instead of the actual parameters. The result, below, will be added at the left of the chain of completed assignments:

((*(/18/(*M(SA-/1/))(*M(EA-/2/))(*M(EA-/3/))))=*M(EA+/0/))

This assignment (now a LISA statement) will be used in the code generation phase to update the stack pointers in the target program, and to create the operating stack record which simulates a call of function /18/.

Finally, function /FP/ passes its results back to /DEX/.

5.3.2.5 Creating Place Holders for Active Elements - /PLH/

Function /PLH/ is called by /DFX/ whenever an assignment is encountered which has as its left side an expression which is more complex than either one active element or one variable. /PLH/ performs an all-level scan of the expression in the left of the assignment, using the multi-bracket technique. The scan is performed from right to left through the expression, since /PLH/ will assign bases for any active elements it finds. This corresponds to the rule for order of evaluation of expressions in Refal: active expressions are evaluated left-to-right at each level of nesting. /PLH/ assigns locations for bases in ascending order, so that the first active element to be found on the right will be given a lower stack offset than one found later; the blocks (representing function calls) with higher stack offsets will be executed first.

The arguments for function /PLH/ then are: 1) the state list; 2) the next available offset for a basis; 3) the current number of blocks in this walk-end; 4) the expression from the left side of the assignment being scanned; 5) a list (initially empty) for storing any new assignments created. The variable from the right side of the assignment is held outside the range of concretization of /PLH/

by an auxiliary function, for reassembly into the completed assignment when /PLH/ is finished.

Function /PLH/ then simply scans the expression. When it encounters any variable, it simply passes over it. When it finds an active element on the right, the active element is replaced by a place-holder, with the mapping taken from the state list entry corresponding to the function whose call has been found. At the same time, a new assignment is created, with this same place-holder variable as its right side, and the active element as its left side. The new assignment is added to the left end of the list of new assignments, and the two global variables are updated. These are the next available location for a basis, and the number of blocks. The next available location for a basis is calculated by adding the block length of the function call to the previous value; the number of blocks is simply incremented by one.

This function also has the task of replacing the '*V' of metacode A expressions with '*'. The object symbol '*' is changed to '*V' in the metacode conversion; /PLH/ takes responsibility for changing it back.

Finally, /PLH/ passes the modified assignment back to /DEX/ as complete. Any new assignments created by /PLH/ are added to the list of incomplete assignments to be processed by /DEX/ on the left end.

5.3.3 Second pass over the walk-end = /MAP2/

A second pass over the walk-end is performed in the context of /MAP1/. This pass occurs after all the stack locations for output assignments are created. At this point, the walk-end consists of a chain of assignments, with the right side of each being a mapped variable. The offsets for these variables are in descending order (looking at the assignment chain from left to right). The left side of each assignment consists of an unmapped expression.

The developing assignment chain is viewed as a set of sequential assignment statements, which compose the output of a function. Each will ultimately be executed in its turn by the target program. Since each assignment may make use of either part or all of the input variables to the function, and since these input variables exist in the same stack of variables as the output is to exist in, it may happen that some of the stack assignments will overlay an input variable before that variable is composed into the proper output assignment. Therefore it is necessary to check for this condition in the assignment chain, and if it occurs, to create temporary variables with which to save the input variables until they are to be composed into the output. This corresponds to the use of temporary variables when exchanging the values of two storage locations.

The purpose then, of /WEP2/, is to determine exactly whether such a situation has occurred. If it has, an assignment is created which has a replacement variable on the right, and the variable which would have been overlayed on the left. This assignment is placed at the left of the assignment chain, so that it is executed before the undesired overlaying can occur. The search for overlayed variables is carried out by function /TMP/, which is called by /WEP2/.

Once /TMP/ has finished, /WEP2/ determines whether any new assignments have been created. If they have, the new assignments will add to the final length of the walk-end, and the maximum walk-end length may need to be updated. Finally, /WEP2/ passes the walk-end back to /MAP1/, which continues through the graph of the current function.

5.3.3.1 Creation of Temporary Variables - /TMP/

The purpose of function /TMP/ is to determine whether the assignment chain will inadvertently wipe out any needed results before they are used. This may occur since there is no direct relationship between the offsets chosen for stacking the output assignments and the offsets of the input arguments. Thus, a variable at say,

offset 3, might not be used until an assignment is made for the variable at offset 1, but meanwhile an assignment is made to the variable at offset 3. Function /TMP/ creates temporary variables which prevent this from happening.

Function /TMP/ makes one left-to-right scan through the walk-end, and upon encountering each assignment, makes a mark in a list of stack offsets. Once this initial mark is made for a particular offset, if the input variable at the same offset appears further to the left, then it is clear that the variable will be overwritten. In that case, function /TMP/ creates the temporary variable, and an assignment for that variable, and places the assignment at the left (beginning) end of the the walk-end. Additionally, since this assignment will be to a variable in the stack, the maximum stack length needed for the function is updated. Finally, every occurrence of the saved variable further to the right from the point of saving is given the offset of the temporary variable.

The result of function /TMP/ is returned to /MAP1/ via /WEP2/.

5.4 SECOND PHASE OF THE MAPPING = /MAP2/

Function /MAP2/ serves two purposes. The first is to map the input variables appearing in the left sides of assignments, and the second is to insert the proper garbage collection directives in each walk-end. At the point of invocation of /MAP2/, each function graph has all its input variables indexed, but the indices are not stack offsets. The indices are used only to relate the variables appearing in the left side of the graph with those in the right side (walk-ends). The offsets are calculated by adding the maximum stack length of the walk-ends in the graph to the indices. This method insures that input variables will always reside in the stack at locations which will not be overwritten by assignments.

Function /MAP2/ makes one depth-first scan of each function graph, using the multibracket technique. Its arguments are the function graph itself, the maximum length of the output stack (to be added to indices to arrive at the offsets), the state list (which contains the mappings of each variable), and a stack which represents the derivations of each partial variable in the function graph. This stack is added to as each contraction is passed and given its offsets, and expands and contracts similarly to the stack used by function /MAP1/. This

stack is used by function /GC/, which performs the garbage collection strategy upon each walk-end.

The operation of /MAP2/ then is straightforward. Offsets and mappings are given to each input variable in the left side of the function graph, and each walk-end is passed to /GC/, along with the current partial-variable derivation tree. The result of /MAP2/ is the completed LISA function definition, which is passed to the routine which formats it for output. Function /GC/ is described below.

5.4.1 Garbage Collection - /GC/

The purpose of function /GC/ is twofold: 1) to insert the proper garbage collection directives at the beginning of each walk-end, so that any value in the computer's information field which is not needed is restored to free memory, and 2) to perform the actual/ghost mapping for input variables.

The first task is performed by making repeated right-to-left scans of the walk-end, one for each argument of the function. /GC/ scans the left side of each assignment in the walk-end for as yet unmapped variables. Each time one is found, a check is made to see if the variable in question is a derivative of the argument being scanned for.

If so, the variable is mapped with status A, and an indication is made that occurrences of the same variable further to the left must be mapped as ghosts. At the same time, an indication is made in the tree of derivations that the variable in question will be used in the output, and so a directive (?RV(n)) will have to be added to the walk-end. If the variable in question is not a derivative of the the argument being scanned for, it is passed, and will be picked up on a subsequent scan.

When /GC/ completes scanning, a number of indications for (?RV(n)) directives will exist. /GC/ finally determines if any of the variables indicated for reservation are total input variables. If so, then the (?OUTAV(n)) format will be required, and the garbage collection directives are added at the beginning of the walk-end. If not, then the abbreviated (?OUT) format is used.

Function /GC/ is the last function to modify the graph of a function, which may now be properly called a LISA function definition. Any optimizations to the completed graph may be performed after this point (for example, the iteration for recursion optimization). Otherwise the function is passed through /MAP2/ to the output routine of the mapping scheme.

6. OPERATING ENVIRONMENT FOR THE TARGET PROGRAM

Once a graph of states has been mapped onto Pascal, it requires an environment in which to operate. Of course, each configuration (or function) in the graph of states is mapped as a Pascal compound statement. This is essentially the configuration declaration, similar to a procedure declaration in Pascal. However, aside from the configuration declarations themselves, we must have declarations of all the global variables, definitions of the variable and operating stacks, access programs for the various data types, and definitions of internal and external functions. Thus, the final Pascal program will consist of a few separate sections, some of which will appear unchanged in each mapped program, and some of which will be defined by the graph of states of the original program.

In effect, it is necessary to define an abstract machine which will provide the operating environment within the constraints of the target language. For the mapping onto Pascal, this machine will be called PMAIN, for Pascal MAIN program. Included in PMAIN are:

- All the internal functions necessary for accessing each data item in the variable stack. As different data types are mapped, it will be possible to adjust the accessing method for maximum efficiency for each type.

- Declarations of various global variables and types. For example, Refal expressions are mapped with a doubly-linked list structure. Each symbol consists of one piece, which in Pascal is defined as a dynamic record. The type piece is then defined within PMAIN in the fixed portion of the program, and various static pieces are also declared, for use by the accessing procedures.
- Definition of a stack interpreter. This is a section of the program which will determine which function is to be executed at what time. The stack interpreter consists of an operating stack of function calls, and a Pascal CASE statement which reads the current operating stack record and transfers control to the function specified. When a function completes executing, it transfers control back to the stack interpreter. Each function is mapped in such a way that it adds the appropriate records to the operating stack so as to cause execution by the stack interpreter of the correct function on the proper arguments at the appropriate time.

The Pascal language by itself has no stopping function. In order to mimic a step-by-step machine, such as the Refal machine itself, the stack interpreter must be able to determine when execution has been complet-

ed, and then to return control to the operating system. In Pascal it is necessary to create a HALT function, which essentially says, go to the last line in the program and stop. A label is given to the final statement in the program, and when the stack interpreter determines that all function calls have been evaluated (i.e., the operating stack is empty), it transfers control to this label.

The stack interpreter also takes responsibility for all the housekeeping functions, such as recording how many steps the machine has executed, which functions have been executed, and so on. This is useful when running a mapped program in debug mode. In this mode, it is possible to have PMAIN print out the arguments and results of each function evaluation.

- All the external functions relating to the original language of the program before supercompilation, modified to run in the PMAIN environment. For example, if we supercompile a FORTRAN program which deals with floating point numbers, we must provide procedures in PMAIN which will perform the same functions on PMAIN objects as the original program performed on its FORTRAN objects.

Appendix D shows the global declarations section of a PMAIN program, i.e., the lowest level of the background environment. Not included in that appendix are the PMAIN internal functions, and the external functions (corresponding to Refal external functions). Figure 10, below, shows the structure of a PMAIN program.

```
Program Heading
Global Declarations (see appendix D)
PIF (PMAIN Internal Function) Declarations
PDF (PMAIN Debugging Function) Declarations
PRF (Pascal-Refal external Function) Declarations

BEGIN  (main program)
   PMAIN initialization statements
   1: BEGIN
        <starting-function-definition>
      END;
   0: BEGIN
        stack interpreter
      END;

      remaining function definitions

<end-label> : END.
```

Figure 10: Structure of a PMAIN program

7. TRANSLATION OF LISA INTO THE TARGET LANGUAGE - PASCAL

This chapter contains a discussion of the significance of LISA statements, and samples of translations into Pascal. Subsequent sections give descriptions of sample LISA statements.

For the reader's convenience, the TYPE and VAR declaration sections of PMAIN have been included as Appendix D. Variable indentifiers in the Pascal translation statements will then refer to those declarations. The purpose of each target language variable will be explained when necessary.

7.1 RELATION OF LISA ENTITIES TO PMAIN

This section relates the various LISA program entities to the actual Pascal code generated for the PMAIN environment. The entity <LISA-program> gives the format of a program created by the mapping scheme. The purpose of each of the constituents of the program is detailed below:

- <program-comment> is used as an identifier for the program. It may contain information on which supercompiler produced the program, which mapping program version was used, what the name of the original Refal program was, etc. Any string may be used, including the empty string; the comment is enclosed in structure brackets.
- <end-label> is the next available index for a label. In the present implementation, this is used only for the definition of the stopping function

PIFHALT. The label is used to identify the last "END." in the target program; the stopping function is written then as a simple GOTO statement with the label of the last statement as its destination.

- The <internal-label-list> gives all the labels used as branch points within function definitions. It is used along with the <function-id-list> and the <end-label> to produce the LABEL declaration needed in every Pascal program (and will probably be needed for generation of code in other target languages).
- The <function-id-list> is used to create the global label declaration, and also to create the stack interpreter. Each label in this list which is followed by a parenthesized string represents an external function, and is used to produce a line of code such as:

<label> : <label> : BEGIN <external-function> ; PIFPOP END ;

Each label without a following external function name is used to produce one line of code in the stack interpreter, thus:

<label> : GOTO <label> ;

- A <function-definition> is used to define a Pascal compound statement representing a function from the supercompiled program. This is the basic element of a LISA program, and there will usually be

many in each program. The definition of function 1, the starting function for the program, is always specified before any of the others. The remaining function definitions may appear in any order after function 1.

- The <starting-function-definition> is always the first function to be executed, and in the structure of a PMAIN program appears just before the stack interpreter (see figure 10) so that it is executed exactly once before the stack interpreter is invoked for the first time.

7.2 THE CODE GENERATION SCHEME

Sections 4.5.3 and 4.5.4 give examples of LISA to Pascal translation statements. These statements are implemented in the code generation scheme (i.e., LISA compiler) as simple Refal sentences which match the LISA statements against patterns, and produce code according to the pattern matched.

The branching destinations are recognized by the context of each function definition. Remember that in the mapped LISA program, each branch point is labelled. Then, if a label follows any contraction at some level, the branch destination used in the translation process is that

label. If no label follows at the same level, recognition is impossible, and the code for implementing the recognition impossible state is inserted in place of a GOTO statement.

8. EXPERIMENTS IN OPTIMIZATION

As a first test of the mapping scheme, a number of Refal programs were fed to the Supercompiler, and the resulting programs were mapped onto Pascal. The resulting Pascal programs were run side-by-side with the original Refal programs, just to check the result of computation. In all cases, the original Refal programs and the parallel Pascal programs produced identical results. The programs tested ranged from the simple program given as figure 1, to a program which parses arithmetic expressions and produces assembler code for a hypothetical single-addressed machine.

The most important test, of course, is the test of efficiency of the mapped program as compared with the supercompiled program being run under Refal. Unfortunately, at this point it would seem that Pascal may not have been the most appropriate choice for the target language, for a number of reasons, and these preclude a fair comparison. First, at CUNY, the only Pascal system available which has

been able to compile the programs produced by the mapping (for reasons of program size only), is Waterloo Pascal. This Pascal system is not a compiled system, and so any program produced must be interpreted, which adds great inefficiency to execution, beyond the reach of the programmer. Also, because Waterloo Pascal does not allow separate compilation, the compilation phase includes all the time required to compile the background functions needed to execute a PMAIN program. At the same time, Waterloo Pascal does not include any timing functions, so the only means for obtaining any timing information which exist are at the operating system level, such as job statistics when running under MVS, or the TIMES package, when running under VM.

Pascal itself has other drawbacks. The language is very highly typed, which in the stages of development of a program is desireable. However, type and range checking are costly, and it should be assumed that a program produced by the supercompiler includes any such checking necessary. A production Pascal system by rights should contain the means for disabling the type and range checking, but unfortunately Waterloo Pascal is not a production system. Pascal/VS is also available at CUNY, and does give the option for disabling type and range checking, but is simply not capable of compiling programs

as large as the ones produced by the mapping. In order to use Pascal/VS, it would be necessary to completely restructure the mapped programs produced so that each function in the program would become a Pascal procedure, resulting in the loss in efficiency associated with repeated procedure calls.

Additionally, Pascal does not give enough flexibility in the definition of pointer types. Pointers in Pascal are allowed to point to only one type of variable, even though in reality all pointers are basically machine addresses. It is possible to side-step this issue by using record types with free unions, but then there is the added problem of storage mappings: how much space is used up by a free union which consists of 3 different pointer types? One would hope that only the space required for one machine address would be used, but this is not necessarily the case. Also, if a free union is used, a different field identifier must be used for each kind of pointer, and overhead is added by the mechanism which checks for the "invisible" tag-field value. As the implementation of LISA depends strongly on pointer operations, any inefficiency added at this level costs dearly. The problem becomes acute when attempting to implement the variable data-typing described in this work.

It may be concluded then, that comparisons of timing between programs run in interpreted Refal, and those run in mapped Pascal may be meaningless. There are, however, some quantitative comparisons which may be made, namely, comparisons of the effects of the optimizations to the mapped programs. To that end, a simple Refal program was devised with the goal of trying to suppress as much as possible the effect of compilation costs in Waterloo Pascal, while at the same time testing the facilities of the mapping scheme. The program may be found in figure 11, below.

```
JOB  =  K/FORMAT/ () K/TRIM/ K/CARD/ ...
FORMAT ()'#' E1 = K/PROUT/'IMPROPER INPUT FORMAT: #'E1.
 (EN)'#' E1 = K/REP/ K/NUMB/EN. E1.
 (EN) S1 E2 = K/FORMAT/ (EN S1) E2.
TRIM S1 ' ' = K/TRIM/ E1.
 E1 = E1
REP /0/ EX =                                              +
   K/PROUT/'END OF LOOP1:  RESULT = 'K/FA/ K/FB/ EX...
 SN EX =                                                  +
   K/PROUT/ K/FA/ K/FB/ EX... K/REP/ K/SUB/ (SN)/1/. EX.
FA 'A' EX = 'B' K/FA/ EX.
 SY EX = SY K/FA/ EX.
 =
FB 'B' EX = 'C' K/FB/ EX.
 SY EX = SY K/FB/ EX.
 =
=END
```

Figure 11: Program for timing comparisons

The core of this program is function /FA/, as described earlier, and function /FB/, which is identical except that it replaces B's with C's. The program reads one card of data, which starts with a string of digits, followed by a number sign (#), followed by any string of symbols. The digit string is used to control the number of times the main loop in the program is executed. The main loop takes the symbol string read in, and applies first function /FB/, and then /FA/ to the string, and then prints out the result. This operation is repeated the number of times indicated by the digit string from the input. Thus, the symbol string must be copied repeatedly, and the operation of the two string processing functions is repeated as many times. By running the resulting program twice, once with a small number of iterations, and the second time with a large number of iterations, the time per iteration can be approximated, and the compilation time can be removed. Of course, Pascal interpretation time, and range checking time are still included in the resulting figures, but for each level of optimization to the mapped program, a figure which can be compared fairly is arrived at.

Table 1 is a tabulation of the results of running the test program with the current implementation, showing the effects of the various optimizations implemented so

far. The stack of variables optimization is inherent in even the lowest level version (i.e., level 1). Level 2 includes the optimization of bypassing the stack interpreter, and level 3 includes the level 2 optimization, and the replacement of recursive function calls with iteration.

	10 times		400 times		sec. / iter.	
Level	Avg. Virt. CPU (sec.)	% err.	Avg. Virt. CPU (sec.)	% err.	abs.	per cent
1	5.313	1.48	56.559	1.59	0.1314	100
2	5.336	1.49	56.360	1.33	0.1318	99.6
3	5.000	1.07	50.835	1.45	0.1175	89.4

TABLE 1

Comparison of Optimizations by execution time

Table 1 was compiled by taking the supercompiled program, mapping it with three different levels of optimization, and then running each program a number of times. Each experiment was repeated a number of times, in order to arrive at meaningful figures for run time (Avg. Virt.

CPU sec. in the table). The programs were run using Waterloo Pascal under the CUNY VM operating system, with the IBM TIMES package for timing. Repeatability for the experiments was surprisingly poor (it was assumed that "virtual CPU" time was exclusive of paging and so forth, and yet repeating the same experiment a number of times resulted in a wide range of values). In order to reduce the effect of the variability of timing results, any group of experiments which resulted in a percent error of more than 2% was analyzed, and "bad" samples were thrown out. After this process, each of the figures for virtual CPU was comprised of the result of repeating the experiment at least 4 times.

The major columns "10 times" and "400 times" show the results of running the program so that the inner loop was executed the corresponding number of times, along with percent timing error for each group of experiments. The last major column, "seconds per iteration", shows the calculated time required for each iteration, and compares these times for each level of optimization with the lowest level. This calculated figure represents the time needed for each iteration, exclusive of the compilation time of the program, and was arrived at by subtracting the time required for 10 iterations from the time required for 400 iterations, and then by dividing by 400 - 10 = 390. The

percent column shows that the first optimization, that of bypassing the stack interpreter, results in negligible increase in efficiency, but that the second optimization, replacement of recursion by iteration, results in a 10.6% increase in efficiency. Of course, these figures are not absolute; other programs may experience varying degrees of speedup. In particular, the stack interpreter bypassing optimization did not show much speedup in this program, because it only resulted in three actual occurrences of the optimization, which appeared outside the main loop of the program. The iteration for recursion optimization, on the other hand, appeared repeatedly within the main loop, and therefore showed a substantial gain in performance. Additionally, it must be pointed out again that the means of performing these measurements are far from satisfactory, but are the only means currently available on the CUNY system.

Other means of comparison between different levels of optimization are the amount of object code generated for each program, and the number of statements executed when each was run. With these goals in mind, table 2 was compiled. Of course, the cautions that applied for table 1 also apply to this one. The amount of object code generated for each program of course contains all the Pascal type and range checking code, the Pascal interpretation code, and so

forth. Also, the object code generated from the fixed part (i.e., the PMAIN background functions) of each program is included in this number. However, a rough comparison may be obtained. As for the number of statements in each program, it is more useful to isolate the number of statements in the main program section, exclusive of the fixed part. Finally, the number of statements executed by each program is interesting, in that it is the only repeatable measure of execution efficiency. Of course, there is nothing that says what kinds of statements are executed each time, or how much time each takes to execute, but assuming that each PMAIN program uses similar statements (due to the nature of the code generation scheme), this might be the most meaningful measure of comparison, given the tools at hand.

Notice that in table 2, the percentage gain for the first optimization, bypassing of the stack interpreter, is negligible. The reason for this has been given. The gain achieved by replacing recursive function calls with iteration, where possible, shows a 13% gain in efficiency. The figures for statements executed per iteration were arrived at by the same method as for CPU seconds per iteration in table 1, in order to remove the number of statements executed in the PMAIN initialization segment.

Level	Lines of Code in main prog. block	Bytes of Object code generated	Statements Executed	
			per iter.	percent
1	578	61943	2218	100.0
2	582	61911	2209	99.6
3	563	59559	1929	87.0

TABLE 2

Comparison of Optimizations by program size and statements executed

In order to test the variable data-typing scheme, another simple Refal program was devised, in which the inherent data types of the variables are simple characters and strings. The program is similar to the one used in the previous experiments, in that it consists mainly of one function, again called /FA/, which replaces all occurrences of the letter A with the letter B. The difference is that in this case, /FA/ is defined as a purely iterative function of two arguments, so that no doubly-linked list structure is required for the data. The program is shown in figure 12, below.

The program in figure 12 was supercompiled, and then mapped using the level 3 mapping described earlier in this section, and then mapped again using level 4, which derives

```
JOB = K/PROUT/ K/FA/() (K/CARD/.)..
FA (E1) ('A' E2) = K/FA/ (E1 'B') (E2).
   (E1) (S2 E3) = K/FA/ (E1 S2) (E3).
   (E1) () = E1
=END
```

Figure 12: Iterative program for testing data-typing

data types more specific than E and S. The mapped LISA programs are shown in figures 13 and 14, repectively.

```
 1    /2/:(
 2         (
 3          (*M(ET-/2/)=(*M(SP-/5/)*M(EP-/4/)))
 4          :(
 5            (
 6             (*M(SP-/5/)=(A))(?RV(/4/))(?OUTAV(/2/))
 7                [(*M(EA-/4/))=*M(EA-/2/)]
 8                [(*M(EA-/1/)B)=*M(EA-/1/)]
 9                (?INCSTEP)(?X(/2/))
10            )
11          /3/+
12            (
13             (?RV(/5/))(?RV(/4/))(?OUTAV(/2/))
14                [(*M(EA-/4/))=*M(EA-/2/)]
15                [(*M(EA-/1/)*M(SA-/5/))=*M(EA-/1/)]
16                (?INCSTEP)(?X(/2/))
17            )
18           )
19         )
20       /4/+
21         (
22          (*M(ET-/2/)=())(?OUTAV(/2/))
23             [(*M(EA-/1/))=*M(EA+/0/)](?POP)
24         )
25        )
```

Figure 13: Level 3 mapping of iterative function

```
 1    /2/:(
 2          (
 3           (*M((AC)T-/2/)=(*N((C)P-/5/)*M((AC)P-/4/)))
 4           :(
 5              (
 6               (*M((C)P-/5/)=(A))[ (*M((AC)A-/4/))=*M((AC)A-/2/) ]
 7                  [ (*M((AC)A-/1/)B)=*M((AC)A-/1/) ]
 8                  (?INCSTEP) (?X(/2/))
 9              )
10           /3/+
11              (
12               [ (*M((AC)A-/4/))=*M((AC)A-/2/) ]
13                  [ (*M((AC)A-/1/)*M((C)A-/5/))=*M((AC)A-/1/) ]
14                  (?INCSTEP) (?X(/2/))
15              )
16            )
17          )
18       /4/+
19          (
20           (*M((AC)T-/2/)=())[ (*M((AC)A-/1/))=*M((AC)A+/0/) ](?POP)
21          )
22        )
```

Figure 14: Level 4 mapping of iterative function

Notice that the differences between the functions shown in figures 13 and 14 lie in the data types of the variables, and in the absence of garbage collection directives in the latter. In the level 4 mapping, all variables have been mapped with data types (C) and (AC), which stand for character and array of characters (string), respectively. The structure of the function, however, is the same at both mapping levels.

In order to compare execution speeds of the two mappings, it was necessary to redefine PMAIN so that it would work

with the new data types. The same machine structure was retained, but the PMAIN functions which operate on the new data types were added. Tables 3 and 4, below, detail the results of the comparison.

	Statements Executed		Execution Time	
Level	absolute	percent	absolute	percent
3	1685	100	0.054	100
4	601	35.7	0.032	59.3

TABLE 3

Comparison of Data-typing optimization by execution time

As seen in table 3, execution time was speeded up by 30.7% by mapping the program onto the character and string data types. The number of statements executed was reduced by 64.3%. Table 4, below, compares program lengths. Again, substantial reductions were obtained by this optimization.

Level	Bytes of Object code generated	Lines of Source code generated
3	29789	741
4	6522	166

TABLE 4

Comparison of Data-typing optimization by program size

9. CONCLUSIONS

A method has been presented for mapping the graph of states of the Refal machine onto a Von Neumann type sequential language. This method paves the way for implementing a supercompiler system which produces efficient programs for real computers. The development of this method has resulted in:

1. The definition of an intermediate language, which allows the specification of arbitrary data types for its variables. The language, LISA, has a simple tree structure, which makes it easy to translate into code in an implemented sequential language, and is written in an algebraic form which simplifies the manipulations necessary to perform optimizations.

2. The implementation of a method for translating the abstract program represented by the graph of states into the intermediate language. The method includes a number of optimizations to the original program, and the potential for further optimizations, which have been outlined.
3. The implementation of a scheme which generates code in the target language, Pascal, from the LISA program. The code generation process may be easily retargetted for other languages, either high or low-level (i.e., assemblers).
4. The implementation of a LISA machine in Pascal. This includes all the internal functions and definitions needed to run the mapped program in Pascal, along with the definitions of the external functions available in Refal.

Appendix A

REFAL SYSTEM STANDARD EXTERNAL FUNCTIONS

The following is a list of the standard functions and their meanings and syntax:

/CARD/. Data for programs may be read from card images using the built in procedure /CARD/. The result of concretization of K/CARD/. is the 80 object symbols on the next card. At the first call the first card is read, at the second call the second card is read, etc. When function /CARD/ finds no more cards, the result of replacement is the empty expression. Calling function /CARD/ once more will result in an abnormal stop.

/PRINT/. The result of concretization of K/PRINT/ EX., where EX is any expression, will be the printing of expression EX, and EX will be left in the view-field in the place of K/PRINT/ EX. All compound symbols will be printed in quotes, and object symbols and structure brackets will be printed as characters.

/PROUT/. Function /PROUT/ works the same as /PRINT/, except that the empty expression is left in its place in the view-field.

/BR/, /DG/, /CP/, /RP/. These are functions used for "saving" intermediate values during execution. The implementation of the Refal machine has a third field, called the stock, in which we can "bury" (/BR/) and "dig" up (/DG/) saved values. For example, if we have a term K/BR/'X=APPLE'. in the view-field, the Refal machine will store the value 'APPLE' under the name 'X', and remove the term from the view-field. If we subsequently have a term K/DG/'X'. in the view-field, the Refal machine will look in the stock, find out that the value of 'X' is 'APPLE', replace the term with 'APPLE' and remove the entry from the stock. If we bury two values under the name 'X', then when we call the dig function, one value will be returned each time, in the reverse order from which they where buried. If function /DG/ finds no entry under the name it is called with, it returns the empty expression. Function /CP/ works similarly to /DG/, except that it does not remove the entry from the stock (i.e. /CP/

- copy). Similarly, function /RP/, when called, adds a value to the stock, just as /BR/ does, but if it finds that the stock already contains a value under the name it is called with, it replaces that value with the new one.

/ADD/, /SUB/, /MUL/, /DR/. These are the functions for integer arithmetic. All are called in the same format: K/ADD/(N1)N2. where /ADD/ can be replaced by the appropriate determiner for addition, subtraction, multiplication, and integer division, respectively, and N1 and N2 are macrodigits. The result of concretization of the first three will be a macrodigit, and for division, the result will be of the form Q(R), where Q is the quotient, and R is the remainder. Division by zero results in an abnormal stop.

/NUMB/, /SYMB/. Function /NUMB/ converts an string of object symbol digits into a macrodigit, i.e. a term K/NUMB/'123'. will place the macrodigit /123/ in the view-field in its place. Function /SYMB/ works in the opposite direction: K/SYMB/ /123/. will leave the object string '123' in the view-field.

/TYPE/. This function is used for determining what type of element is found in the first position in the expression it is acting on. Its format is K/TYPE/ EX. where EX is any expression. It will return an expression of the form T EX, where T identifies the type of the first element as:

T	First element of EX
'F'	compound symbol-label
'N'	compound symbol-number (macrodigit)
'L'	object symbol-letter
'D'	object symbol-digit
'B'	left structure bracket
'*'	empty
'O'	other object sign

/FIRST/, /LAST/. Functions /FIRST/ and /LAST/ serve to break off a part of a fixed length from an expression. A call to them has the following format: K/FIRST/ N EX. or K/LAST/ N EX. where N is a macrodigit, and EX is any expression. These functions break down the expression into two parts, i.e. the expression EX into E1E2, in such a way that the following condition holds: for /FIRST/ E1 consists of N terms, and for /LAST/ E2 consists of N terms. If it is possible to break EX down in such a way (i.e. EX consists of at least N terms), then the result will have the following format: (E1)E2 for /FIRST/,

and E1(E2) for /LAST/. If the number of terms in EX is less than N, then the output of /FIRST/ is '*'EX, and the output of /LAST/ is EX'*'.

/LENGW/, /LENGR/. The result of concretization of K/LENGW/ EX. or K/LENGR/ EX., where EX is any expression, has the format N EX, where N is a macrodigit. For /LENGW/ N represents the number of terms in expression EX, and for /LENGP/ N is the real length of the expression, i.e. the number of symbols and brackets in EX.

/MULTE/. This function simply multiplies the expression it is operating on. If we write K/MULTE/ N EX., where N is a macrodigit and EX is any expression, the result of concretization will be expression EX reproduced N times.

Appendix B

REPRESENTATIONS AND METACODES

(Excerpted from [1:12-14])

To write in Refal Algorithms dealing with algorithms written in Refal itself we have to represent sentences by object expressions, therefore, we need a special code for this purpose. It will be called Metacode A. We further need a code to input Refal programs into a computer, which will be called Metacode B. It is convenient to represent object signs in Refal by bytes in a computer, and it is convenient to treat each byte as an object sign. Since in I/O operations we are dealing, after all, with sequences of bytes, Refal sentences, and all possible Refal objects, for that matter, must be represented in metacode B by object strings (strings of object signs). In metacode A, Refal objects will be represented as object expressions -- for there is no need to destroy their tree structure.

So, metacode A is a mapping of the set of all Refal objects (that is programs and expressions) on the set of all object expressions. Metacode transformation will be designated by adding an asterisk as a superscript to the designation of a Refal object. If Z is a Refal object, Z* is its metacode-A transformation, Z** its double transformation, etc. Natuarally, the metacode transformation and the reverse must be unique, but there is no need to require that each object expression could be interpreted as the metacode of some Refal object. It would be convenient if the metacode of an object expression were always identical to the expression itself, but this is, obviously, impossible because of the required uniqueness of the inverse transformation. (Indeed, let E be an expression, which is not an object expression. Then E* is an object expression. If its metacode transformation E** must be identical to E*, then the inverse metacode transformation, when applied to E*, must give both E and E*, which do not coincide by the definition of E.) Nevertheless, it is desireable to define metacode A in such a way that the subset of those object expressions E0 for which E0* is not equal to E0 is minimized.

We define a metacode A by the following rules.

- The metacode of a sequence of objects is the sequence of the metacodes of these objects. The metacode of (E), where E is an expression, is (E*). This rule also applies to specifiers.

- The asterisk is a special symbol. Its metacode is *V. All the other symbols are transformed by the metacode into themselves.

- A variable type sign v is transformed into *v. For example, the metacode of sx is *SX. Restriction: The asterisk cannot by used as a variable index.

- An expression K E . is transformed into *(E*)[5].

- A sentence with the left side L and the right side R is transformed into *((L*) = R*)[6].

As an example consider the following program:

```
k/RPM/ e  + e  => k/RPM1/ e . - k/RPM/ e .
        1    2              1             2

k/RPM/ e       => k/RPM1/ e .
        1                 1

k/RPM1/ e (e )e  => e  (k/RPM/ e .) k/RPM1/ e .
          1  2  3     1           2            3

k/RPM1/ e        => e
         1          1
```

which describes a function, replacing the symbol + by the symbol - on all levels of parenthesis structure. In the metacode A it will become the following expression:

```
*((/RPM/ *E1 + *E2) = *(/RPM/ *E1) - *(/RPM1/ *E2))
*((/RPM/ *E1)       = *(/RPM1/ *E1))
*((/RPM1/*E1(*E2)*E3) = *E1(*(/RPM/ *E2)) *(/RPM1/*E3))
*((/RPM1/*E1)         = *E1)
```

---

[5] This rule has been modified. The original metacode transformation was *K(E*). However, it was found that the "K" was unnecessary. The subsequent example has been modified to reflect this change.

[6] Here too, the metacode has been changed: the reversion indicator and comments have been deleted.

We shall not describe metacode B here (it may vary with implementation), we will only give an illustration[7]. This is how the above program will appear on the programming form:

```
RPM E1 '+' E2 = K/RPM1/E1. '-' K/RPM/E2.
    E1 = K/RPM/E1.
RPM1 E1(E2)E3 = E1 (K/RPM/E2.) K/RPM/E3.
     E1 = E1
```

---

[7] The metacode transformation given is in fact that found in the CUNY/UCC implementation of REFAL.

Appendix C

DEFINITION OF BASIC REFAL

(Excerpted from [1:7-11])

1. Syntax

A considerable part of the syntax will be described in the Backus Normal Form.

1.1 Signs

```
<sign> ::= <specific sign> | <object sign>
<specific sign> ::= / | <bracket> | <variable type sign>
<bracket> ::= <structure bracket> | <concretization bracket>
<structure bracket> ::= ( | )
<concretization bracket> ::= K | . | =
<variable type sign> ::= S | W | E
```

<Object sign> is any character (represented by a byte in the computer). To distinguish object signs from those characters that are used to represent special Refal signs object signs are embraced in quotes, e.g. 'A' or ')'. A quote as a single object sign is represented by a pair of quotes '' .

An <object string> is a non-empty sequence of object signs. It is put into quotes as a whole, e.g. 'ED+35/5' .

1.2 Symbols and Expressions

```
<symbol> ::= <object sign> | <compound symbol>
<compound symbol> ::= /<object string>/
<expression> ::= <empty> | <expression><term>
<empty> ::=
<term> ::=
    <symbol> | <variable> | (<expression>) | K<expression>.
<variable> ::= <simple variable> | <specified variable>
<simple variable> ::= <variable type sign><index>
<index> ::= <object sign>
<specified variable> ::= S<specifier><index>
<specifier> ::= (<object string>) | <compound symbol>
```

A pattern expression is an expression, which does not contain concretization signs (but generally contains variables). A workable expression is an expression, which does not contain variables (but generally contains concretization signs). An object expression is an expression, which contains neither concretization signs nor variables.

1.3 Sentences and Programs.

The following syntax definitions are given for the ABSTRACT Refal machine, which is used below to formally define the semantics of Refal. The way a program is written for computer input is described in Part II .

```
<sentence> ::=
     #<comment><reversion indicator><left side><right side>
<comment> ::= <object string> | <empty>
<reversion indicator> ::= <empty> | (R)
<left side> ::= K<pattern expression>=
<program> ::= <empty> | <program><sentence>
```

No sentence can contain variables with identical indices but different type signs. The right side of a sentence can contain only those variables appearing in its left side. Specifiers in right sides are omitted.

By the range of a concretization sign K in an expression we mean the subexpression bounded by this sign and the concretization point "." paired with it. We call the leftmost sign K with no other signs K in its range the leading sign K.

2. Syntactical Recognition

2.1 We say that an object expression EO can be syntactically recognized as a pattern expression Ep, if the variables in Ep can be replaced - observing the rules listed below - by such expressions, called their values, that Ep becomes identical to EO. The rules are as follows.

2.1.1 A variable of the form Sx, Wx, or Ex, where x is the index, can take as a value any symbol, term, and expression, respectively.

2.1.2 A variable of the form S(P)x, where P is an object string, can take as a value any symbol, which enters P. Variables S/SIGN/x and S/COMP/x take as values object signs and compound symbols respectively. A variable of the form SDx, where D is a compound symbol different from those two, is equivalent to a variable S(P)x, where P is the result of concretization of KD. .

2.1.3 All entries of the same variable, i.e. those with the same index, must be replaced by the same value.

2.2 If there are several alternative ways of assigning values to the variables, the ambiguity is resolved in one of the following two ways, which will be called recognition from left to right, and from right to left. If recognition from left to right (from right to left) takes place, then of all alternatives the one in which the leftmost (rightmost) expression variable Ep takes the shortest value is chosen. If this does not resolve the ambiguity, the analogous selection is made with respect to the second from the left (right) expression variable, etc.

2.3 To recognize a term K E0 . as a left side K Ep = means to recognize E0 as Ep.

3. The Refal Machine

The Refal machine is an abstract device which executes algorithms written in Refal. It consists of two potentially infinite stores, which are called the memory-field and the view-field, and a processor. At every moment in time the memory-field contains a finite sequence of sentences, and the view-field contains a workable expression.

The Refal machine works in steps. Having fulfilled a step, the machine proceeds to execute the next one, provided that the former has not led to a normal or abnormal stop. Execution of the step begins with the search for the leading sign K in the view-field. If there is no sign K, the Refal machine comes to a normal stop. On finding the leading sign K the Refal machine examines the term which begins with it; it is called the active term, and we say that the starting sign K has become active.

3.1 If the active term is K/BR/N'='E., where N and E are expressions, the machine writes down a new sentence #K/DG/ N = E into the memory field, putting it before the first sentence. The active term is removed from the view-field, and the step is completed.

3.2 If the active term is K/DC/ N., the Refal machine finds the first sentence in the memory-field of the form #K/DG/ N = E with the same N, removes it from the memory-field and substitutes E for the active term, thus finishing the step. If there is no such sentence, the active term is merely removed.

3.3 In other cases the Refal machine compares the active term with the consecutive sentences in the memory-field, beginning with the first one, searching for an applicable sentence, by which we mean such a sentence that the active term can be recognized as its left side. Recognition is performed from left to right if the reversion indicator is empty, and from right to left if it is (R). Having found the first applicable sentence, the Refal machine copies its right side, replacing the variables by the values they have taken in the process of recognition. The workable expression thus formed is substituted for the active term, and the step is finished. If there is no applicable sentence, an abnormal stop occurs.

4. External Functions

In real implementations of Refal, as distinct from the abstract Refal machine described above, one more action is taken at each step before using the sentences: the examination of whether the active term is or is not an external function call. By external we mean those functions which are not described in Refal. Some symbols must be specified in every implementation as external function determiners. If the active term has the form KFE., where F is such a determiner, control goes to a program (or whatever) that performs the concretization. It may result in the replacement of the active term by some workable expression, and may produce any effect in the environment. After it is over, the current step is finished and control goes back to the Refal machine.

The functions which provide input/output facilities clearly must be external. In all implementations, a function /PRINT/ is available, which is defined so that when a term K/PRINT/E. becomes active, the expression E is printed and the term is transformed into E. Another function, /PROUT/, prints arguments and deletes the active term.

We do not introduce into the formal description of Refal the concept of number, but in the implementations it is possible to code positive integer numbers in a certain range (e.g. for IBM/370 up to 2 exp(31) - 1) as compound symbols of a special kind (called a macrodigit). The arithmetic operations on them are performed with the aid of the appropriate external functions. In the process of recognition, a macrodigit will be recognized as one symbol variable, without regard to the number of real digits involved.

A compound symbol which enters a symbol variable as a specifier may also represent an external function.

Appendix D

TYPE AND VAR DECLARATIONS FROM PMAIN

This appendix has been included as a reference in describing Pascal code generated when a graph of states is mapped onto Pascal. Appropriate comments have been added to the code. The declarations have been extracted from a program actually produced by the current mapping. Note that the data types inherent in this mapping are primarily REFAL expressions.

```
CONST STACKDEPTH = 100 (* DEPTH OF OPERATING STACK *);
      MAXNUMBER=16777215 (* = 2 TO 24 MINUS ONE *);

TYPE NUMBERTYPE = 0..MAXNUMBER;
     ADDR = @PIECE;
     PIECETYPE = (SYM, PAR, PLH, HEAD);
     SYMPARTYPE = (CP, NP, WP, LPAR, RPAR);
     SYMTYPE = CP..WP;
     PARTYPE = LPAR..RPAR;
     PIECE = PACKED RECORD
        PREC, FOLL : ADDR;
        CASE PTAG : PIECETYPE OF
           SYM : (CTAG: SYMTYPE; CODE: NUMBERTYPE);
           PAR : (BTAG: PARTYPE; PAIR: ADDR);
           PLH : ();
           HEAD : ()
        END;
     XSRANGE = 0..STACKDEPTH;
     BSRANGE = 0..STACKDEPTH;
     FLABELRANGE = 1..115;
     BLOCKHEADER = PACKED RECORD
        F : FLABELRANGE      (* FUNCTION NAMES *);
        L : BSRANGE          (* BLOCK LENGTHS  *)
        END;

VAR
    XS : ARRAY (.XSRANGE.) OF ADDR  (* EXPRESSION STACK *);
    XP : XSRANGE (* EXPRESSION STACK POINTER *);
    BS : ARRAY (.BSRANGE.) OF BLOCKHEADER ;
    BP : BSRANGE (* BLOCK HEADER STACK POINTER *);
    (* AUX. POINTERS FOR BUILDING EXPRESSIONS *)
    B : ADDR (* POINTER THAT MOVES THROUGH EXPRESSIONS *);
    B0 : ADDR (* ANCHOR PIECE, INITIALIZED THROUGH CREATION *);
    B1,B2 : ADDR (* LEFT AND RIGHT ENDS OF NEW EXPRESSION *);
```

```
TPLH : ADDR (* POINTER TO TAKER/PLACE HOLDER *);
ADFREE : ADDR (* BEGINNING OF FREE MEMORY *);
FUNC : FLABELRANGE (* CURRENT FUNCTION NUMBER *);
BLEN : XSRANGE (* CURRENT BLOCK LENGTH *);
ABR : ADDR (* POINTER TO LEFT BRACKET CHAIN FOR PAIRING *);
(* AUXILIARY POINTERS FOR EXPRESSION MANIPULATIONS *)
TA0,TA1,TA2,TA3,TA4,TA5,TA6,TA7,TA8,TA9 : ADDR;
TA10,TA11,TA12,TA13,TA14,TA15,TA16,TA17,TA18,TA19 : ADDR;
TA0F,TA1F,TA2F,TA3F,TA4F,TA5F,TA6F,TA7F,TA8F,TA9F : ADDR;
TA10F,TA11F,TA12F,TA13F,TA14F : ADDR;
TA15F,TA16F,TA17F,TA18F,TA19F : ADDR;
TA0P,TA1P,TA2P,TA3P,TA4P,TA5P,TA6P,TA7P,TA8P,TA9P : ADDR;
TA10P,TA11P,TA12P,TA13P,TA14P : ADDR;
TA15P,TA16P,TA17P,TA18P,TA19P : ADDR;
STEP : INTEGER  (* STEP COUNTER *);
DIGITS, LETTERS : SET OF CHAR;
```

Appendix E

GLOSSARY

Active Element — A part of an expression found in an assignment, which represents a function call. An active element takes the form: *(N <var-list>), where N is a macrodigit giving the function number, and <var-list> is a Refal list of actual parameters for this function call. Thus, we may have an active element: *(/6/(*E/4/)(*S/3/)). Equivalently, at the level of graph of states, this active element can be seen as a pair of concretization brackets for function C6: K/C6/ (E4) (S3). .

Assignment — An element of a graph of states which represents the composition of an expression for output, upon successful completion of a walk through the graph. In LISA programs, one assignment is generated for each stack location, thus directing the various results of execution of a function to the next function to operate on each. In a graph of states, an assignment takes the form: ((E)=V), where E is any expression in metacode A, and V is a metacode A variable which is assigned the value of expression E. In LISA, an assignment takes the same format, except that all variables are mapped.

Assignment Chain — A series of assignments which constitutes the end of a walk through a graph. The assignment chain represents the right side of a Refal sentence, i.e., the replacement part of a production rule.

Basis — The initial stack location of a block. When a particular function is executing, its own basis always has relative stack location

zero, i.e., the pointer in the stack of variables points to this basis. Each function call to be created as a result of this function's evaluation will be given its own basis, relative to the basis of the executing function. The basis always points to a place holder which is the destination of the result of evaluation of the function being executed.

Block — A set of sequential locations on the stack of variables which represent the input and output of one function call. When a function is selected by the stack interpreter for execution, the pointer in the stack of variables is set to point to the location of the place holder variable for this function (offset = 0 within this block, i.e., the basis), and the succeeding levels in the stack represent the input variables for the function.

Compiler Directive — A LISA statement which gives the code generation program information needed for setting up the control structure of the translated program. LISA compiler directives include commands for updating the stack pointers, popping the operating stack, restoring dynamic storage to the system, and also include any target language code generated by the mapping program itself, such as the function declarations which call external function. Each LISA compiler directive is represented as one term, in which the first symbol is a question mark, e.g., (?POP).

Condition — A term in a graph of states or LISA program which constitutes a conditional test and branch, i.e., a contraction or a restriction. See Condition Chain.

Condition Chain — The series of conditions in the beginning of one walk through a graph. The condition chain can be seen as corresponding to the left side of a Refal sentence, i.e., a pattern expression. If the pattern is not

matched (all conditions in the chain are not met), a branch is taken to the next walk, or, if none remain in the graph, recognition is declared impossible, and execution is halted.

Conglomeration — The process of replacement of subexpressions in an assignment by their antecedent variables, as created in the condition chain preceding the assignment. This process significantly reduces the complexity of output expressions. See section 5.3.2.2 for an example.

Contraction — An element of a graph of states which represents a condition test followed by the creation of local variables. If the condition test is successful, the local variables are created, and execution continues on the same walk within the graph of the function. If the test is not successful, execution either continues on the next branch within the graph, or execution is halted if no other branches remain in the graph (i.e., recognition is impossible).

Data Type — Part of the specification of a mapped variable. At present, this indicator can take the following values: S - Symbol in a doubly linked list, and E - Expression in a doubly linked list.

Graph of States — A representation of a computer program as produced by a Refal Supercompiler. A configuration in a graph of states represents one function. Each variable in a graph of states is originally unmapped, and may only have the data types S and E, corresponding to symbols and expressions in Refal.

Input Variable — A variable which is one of the arguments to a configuration (function). See also Total and Partial Input Variables in this glossary.

LISA	The Language for Implementation by Stack Assignments. This language was developed as an intermediary between the mapping and code generation stages of translation of a graph of states into a working computer program. A LISA program contains all the information necessary to create each function definition in the target language, and gives the mapping for each variable.
Local Variable	A variable which is created as the result of a contraction. That is, a variable which first appears in the right side of a contraction. Such a variable may itself be contracted in subsequent contractions, and may appear as part of an output expression.
Mapped Variable	A specification of a variable within a mapped program, which denotes all the information necessary to locate, access, and operate on the data contained in the variable. In the present mapping scheme, a mapped variable is represented as: *M(D S R O), where D is the data type, S is the status, R is the role, and O is the offset (location in the stack).
Offset	Part of the specification of a mapped variable. This indicator always takes an integer value in the present mapping, and represents the location of the variable. Since the present mapping employs a stack of variables, this location is represented as a stack offset, hence the name.
Operating Stack	A PMAIN global array which contains one record for each function call. Each record consists of a field specifying the function name (i.e., configuration number), and a field which defines the stack length of the configuration.
Operation	Pseudonym for Condition.

Output Variable	A variable which appears in the right side of an assignment. Such a variable directs the output of a function call to a particular location (stack offset).
Partial Input Variable	A variable which comprises part of an input variable. Thus, a partial input variable is a result of contraction of a total input variable.
Place-holder	An element of an output expression which holds a place for the value of a not yet computed function call. In LISP mappings, place holders are denoted by a '+' as the role indicator. A place holder variable appearing in the right side of an assignment then gives the destination for the result of a function call, while a place holder in an output expression (left side of an assignment) actually holds the place for the same value until evaluation is completed.
PMAIN	The name of the operating environment for programs in the Pascal language produced by the mapping and code generation schemes. PMAIN includes all the internal operating functions needed to execute the Pascal program, and all the external function definitions required by the program itself.
Recognition Impossible	A state of the Refal machine which means that the actual argument to a particular function did not correspond to any of the pattern expressions in the left sides of the sentences defining the function. When such a state occurs, execution is abnormally terminated. A similar condition exists in PMAIN, when none of the walks through a function definition result in successful fulfillment of all the conditions in those walks.
Refal	The REcursive Functions Algorithmical Language, invented by Prof. V. F. Turchin. See appendix 1 for a basic definition of the language.

Referenced Variable	A variable which references a specific value. A referenced variable is indicated by a '-' in the role field. All input variables are referenced variables, as are variables which appear in the right sides of assignments when the value of the variable can be directly computed by the assignment (i.e., no function calls or place holders exist in the expression). Referenced variables appearing in the left sides of assignments refer to variables created in the condition chain of the current walk.
Restriction	An element of a graph of states which represents a negative condition test (see Contraction). Restrictions are removed from a graph of states before mapping, since the transformation of the program from functional to sequential representation obviates their presence.
Role	Part of the specification of a mapped variable. This indicator can take either the value '-', which signifies a referenced variable, or '+', which signifies a place holder variable.
Stack Assignment	In LISA, a statement corresponding to an assignment in a graph of states. All variables in a stack assignment are mapped, and each variable has one field which gives the stack offset, i.e., the location in the stack of variables where the particular variable may be found. The variable on the right side of a stack assignment has an offset which gives the destination for the mapped expression in the left side.
Stack Interpreter	A section of a PMAIN program whose purpose is to simulate execution of successive function calls on the operating stack.
Stack of Variables	A PMAIN global variable which consists of an array of pointers to variables operated on by the program.

Stack Length — The number of stack locations needed to assemble the output of one walk in a graph. Note that this stack length may be different in each walk in a particular graph, depending on the number of function calls found in the each walk. The stack length in the current implementation always corresponds to the number of assignments in a particular assignment chain. See Block, above.

State List — A global variable of the mapping process, which shows the format of each function call, along with the mapping of each input and output variable.

Status — Part of the specification of a mapped variable. At present, this indicator can take the following values: T - Total input variable, and P - Partial input variable, both of which appear only in the condition chain of a walk; and A - Actual variable, G - Ghost variable, and R - Replacement variable, which appear only in the assignment chain of a walk.

Supercompiler — A program written in Refal, which performs certain transformations on another program written in Refal. For more details, see [1].

Total Input Variable — A variable which is one of the arguments to a configuration (function). The term total is applied to differentiate between the arguments themselves, and the partial, or contracted variables, which are results of contraction of the arguments. Total input variables can be viewed as being the formal parameters within a function definition.

Walk — A path through a graph of one function, comprised of a condition chain followed by an assignment chain. Each graph may have a number of walks, since in general, a graph has a tree structure.

Walk-End	That part of a walk which consists only of unconditional assignments, i.e., the assignment chain. In LISA, the walk-end may also include compiler directives.

REFERENCES

[1] Turchin, V.F. The Language REFAL, the Theory of Compilation, and Metasystem Analysis. Courant Institute Report #20, New York, 1980.

[2] Turchin, V.F., Nirenberg, R.M., Turchin, D.V., Experiments with a Supercompiler, Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, Pittsburgh, August 1982.

[3] Turchin, V.F., A Supercompiler System based on the Language REFAL, Proposal for Renewal of National Science Foundation Grant #MCS-8007565, September 1982.

[4] Turchin, V.F., Implementation of Programming Languages through the Supercompiler System, Internal Proposal for Research, The City College of New York, 1982.

[5] REFAL User's Guide, A section from the book [1:7-9], also available at CUNY online as 'WYL.CC.PUB.REFMAN'

[6] REFAL User's Manual, translated from the Russian.

[7] Turchin, V.F., A Supercompiler System based on the Language Refal, SIGPLAN Notices, 14, 2 (February 1979), pp. 46-54.

[8] Winograd, T., Beyond Programming Languages, Comm. ACM, 22, 391 (July 1979).

[9] [Basic REFAL and its Implementation on Computers], Bazisnyi REFAL i yego realizatsiya no vychislitelnykh mashinakh, GOSSTROY SSSR, TsNIPIASS, Moscow, 1977. (The authors are not indicated in the book. In fact they are: Khoroshevsky V. F., Klimov And. V., Klimov Ark. V., Krasovsky A. G., Romanenko S. A., Shchenkov I. B., Turchin V. F.)

[10] Darlington, J. and Feather, M., A Tranformational Approach to Modification, Materials of IFIP Working Group 2.1 meeting, April 1979.

[11] Burstall, R.M., and Darlington, J., A transformation System for Developing Recursive Programs, J. ACM 24, 1, Jan 1977, pp. 44-67.

[12] Kennedy, K., and Schwartz, J., An introduction to the Set Theoretical Language SETL, Comp. and Maths. with Appls., Vol. 1, pp. 97- 119, Pergamon Press 1975.

[13] Aho, Alfred V., Translator Writing Systems: Where do they now stand? IEEE Computer, Vol. 13, No. 8, Aug. 1980, pp. 9-14.

[14] Graham, Susan L., Table-Driven Code Generation, IEEE Computer, Vol. 13, No. 8, Aug. 1980, pp. 25-34.

[15] Ganapathi, M., and Fischer, C.N., Bibliography on Automated Retargetable Code Generation, SIGPLAN Notices, Vol. 16, No. 10, Oct. 1981, pp. 9-12.

[16] Ganapathi, M., and Fischer, C.N., Automated Compiler Code Generation and Reusable Machine-Dependent Optimization -- A Revised Bibliography, SIGPLAN Notices, Vol. 18, No. 4, Apr. 1983, pp. 27-34.