THE UNIVERSITY *of* EDINBURGH

A SEMANTIC APPROACH TO AUTOMATIC PROGRAM IMPROVEMENT

by

John Darlington

Ph.D. Thesis

University of Edinburgh

1972

ABSTRACT

The programs that are easiest to write and understand are often not the most efficient. This thesis gives methods of converting programs of the former type to those of the latter type; this involves converting definitions of algorithms given as recursion equations using high level primitives into lower level flow chart programs.

The main activities involved are recursion removal (c.f. Strong), loop elimination, and the overwriting of shared structures. We have concentrated on the semantics, rather than the syntax, of the programs we are transforming and we have used techniques developed in work done on proving the correctness of programs.

The transformations are done in a hierarchical manner and can be regarded as compiling a program defined in a structured manner (Dijkstra) to produce an efficient low level program that simulates it.

We describe the implementation of a system that allows the user to specify algorithms in a simple set language and converts them to flow chart programs in either a bitstring or list processing language. Both of these lower languages allow the sharing of structures. The principles are applicable to other domains and we describe how our system can be applied more generally.

ACKNOWLEDGEMENTS

Particular thanks are due to my supervisor, Rod Burstall, not only for his constant encouragement and assistance throughout this research but also for introducing me to the ideas of abstract programming and program semantics.

The influence of Cooper, Dijkstra, Milner and Minsky will be obvious from the thesis.

The POP-2 language developed at the Department of Machine Intelligence by Burstall, Collins and Popplestone greatly assisted the implementation of the ideas in this thesis and the sets package also developed at the Department of Machine Intelligence by Ambler and Burstall provided a useful starting point for the thesis.

I would like to thank all my colleagues in the Department of Machine Intelligence for many helpful conversations, in particular I am indebted to Bruce Anderson, Michael Gordon, Nigel Parker, Gordon Plotkin, Robin Popplestone, Rodney Topor and Sylvia Weir.

More than the usual thanks are due to the typist, Eleanor Kerse, not only for her patience and skill but for her active interest in the thesis. Thanks also to Margaret Pithie for printing and compiling this thesis.

Finally I would like to thank the Science Research Council for providing the financial support that made this work possible.

CONTENTS

Chapter 1 Introduction

Chapter 2 Outline

2.1 Introduction
2.2 Philosophy
2.3 Basic Method
2.4 Outline Description

2.4.1 Languages and Domains

2.4.1.1 Source Languages
2.4.1.2 Target Languages
2.4.1.3 Intermediate Languages

2.4.2 Outline of Compiler

2.4.2.1 Removal of Recursion
2.4.2.2 Choice of Representation

2.4.2.2.1 Transformation Extraction
2.4.2.2.2 Planning
2.4.2.2.3 Projection
2.4.2.2.4 Program Writing

2.4.2.3 Example

2.4.2.3.1 Transformation Extraction
2.4.2.3.2 Processing the Transformation
2.4.2.3.3 Implementation in L-0
2.4.2.3.4 Implementation in B-0

2.4.2.4 Destructive Implementation

Chapter 3 Implementation

3.1 Removal of Recursion

3.1.1 Introduction
3.1.2 Source and Target Languages

3.1.2.1 Source Language

3.1.2.1.2 Semantics

3.1.2.2 Target Language

3.1.2.2.1 Target Language Syntax
3.1.2.2.2 Target Semantics

3.1.3 Translation Process

3.1.3.1 Pre-processing
3.1.3.2 Structure Recognising

3.1.3.2.1/

3.1.3.2.1 Matching Algorithm
3.1.3.3 Production of Translation
3.1.3.3.1 Syntax of Tests
3.1.3.4 Examples
3.2 Compilation
3.2.1 Language Descriptions
3.2.1.1 S-1 Syntax
3.2.1.2 S-1 Semantics
3.2.1.3 L-0 Syntax
3.2.1.4 L-1 Syntax
3.2.1.5 L-0, L-1 Semantics
3.2.1.6 B-0 Syntax
3.2.1.7 B-0 Semantics
3.2.2 Compilation Example
3.2.2.1 Extracting the Transformation
3.2.2.2 Simplification
3.2.2.3 Planning
3.2.2.3.1 Production of Alternative Forms
3.2.2.3.2 Evaluation of Alternatives
3.2.2.4 Projection
3.2.2.4.1 Projection into B-0
3.2.2.4.2 Projection into L-0
3.2.2.5 Shrinking
3.2.2.6 Linking
3.2.2.7 Program Writing
3.2.2.7.1 Code Production
3.2.2.7.2 Code Expansion
3.2.2.8 Examples
3.2.3 Destructive Implementation of L-0 Programs
3.2.3.1 Extraction of Transformation
3.2.3.2 Optimisation of Transformation
3.2.3.3 Projection
3.2.3.4 Program Writing
3.2.3.4.1 Examples and Remarks
3.2.3.5 Example

3.3/

3.3 Algebraic Manipulation Routines

3.3.1 Specification
3.3.2 Produceallforms

Chapter 4 Theory of Computation and Computer Science

4.1 Programming and Machine Design
4.2 Theory of Computation

4.2.1 Languages and Meta-Languages
4.2.2 Proofs about Programs
4.2.3 Program Writing

Chapter 5 Extension to General Systems. Extensible Compilers

5.1 Improvements and Suggestions
5.2 Language Extension

5.2.1 Semantic Apparatus
5.2.2 The Generalised Compiler
5.2.3 Example. A transformational compiler for a matrix system.

References

Appendix I The Matching Algorithm

Appendix II The Function Codefor

CHAPTER 1. INTRODUCTION

Here are two versions of one program.

```
reverse(x1) = if null(x1) then nil
                          else append(reverse(tl(x1)),hd(x1)) *

reverse(x1) = nil->ans;
              while not(null(x1))
                        do [tl(x1)->temp; ans->tl(x1); x1->ans;
                                  temp->x1];
```

One is clear and abstract the other tortuous and efficient. Given the first as a definition a competent programmer should be able to produce the second. Can we get the computer to do this?

Again, in fairly abstract terms a definition of set union is

```
union(x,y) = if nullset(x) then y else
                  consset(choose(x),union(minus(choose(x),x),y)).
```

If/

---

* The primitives that we use are based on the POP-2 language, Burstall, Collins and Popplestone (1971). The main features to note are that X->Y is equivalent to the Algol Y:=X, hd is the LISP car, tl is the LISP cdr, append adds an element to the end of a list, and intsub(n,s) selects the nth element of a string s. Nullset, consset, choose and minus are primitives of a set language that we define later, nullset is true if the set is empty, consset adds an element to a set, choose picks an element from a set and minus removes a given element from a set.

If we represent sets of integers as bitstrings in the computer there is an efficient but unnatural implementation for this function, viz.

```
length(x)->l; 1->n;
        while n≤l  do [if intsub(n,x) = 1 then 1->intsub(n,y);
                                    1+n->n];
```

In the following chapters (2 and 3) we first outline and then give details of a system and its implementation that can achieve these transitions. In the above examples the first version is written in a simple language and the second in a more extensive and efficient language with more machine oriented primitives. Thus we can regard the change as a sort of compilation. We might call it 'transformational compilation' since the structure of the program is transformed during compilation.

Chapters 4 and 5 give our views on (i) The relevance of this activity to high level languages, program proving and automatic program writing and (ii) How our transformational compiler could be extended to a wide variety of domains.

CHAPTER 2

2.1 Introduction

In this chapter we present an outline of a scheme for a transformational compiler that will attempt the efficient compilation of programs stated in a mathematical user oriented language for a limited domain. We will describe the implementation of these ideas in more detail in the following chapter.

2.2 Philosophy

The basic approach combines two philosophies of programming and compilation given by Minsky and Dijkstra.

In his Turing Lecture Minsky (1970) said that there is too much preoccupation with form (or syntax) in present day compilers and not enough attention is paid to computational content. We believe that there is a growing divergence between the program structures that enable programmers to clearly and concisely specify computations and structures that are suitable for efficient implementation on present day computers. Further we believe that form based compilation of these structures will be inefficient and traditional optimisation techniques insignificant. We have attempted to produce a compiler that knows a lot more about the programs it is implementing, particularly their semantics, and has a limited inferential capability that allows it to attempt the implementation of high level programs in a more intelligent manner.

The other aim of our approach is basically to implement in our compiler techniques for program production first advocated by Dijkstra (1970) in his works on structured programming. We will not give a full description of the technique of structured programming as it is a widely known and accepted technique. For our purposes, however, we note that it has a natural implementation in programming languages with a well-developed procedure (function) mechanism, for example in LISP. Here, if one is faced with the task of writing a program to perform a given task using certain basic primitives with known semantics one can adopt the following top down method. Assume you have enough useful high level primitives to make the writing of the program trivial and write a program (or function) using these primitives. You are then faced with the task of realising these primitives, which you do by the same process, until all the last functions written use only the originally given primitives. This method applies whether we are thinking of purely functional languages or ones with imperative procedures. Thus if we view a program as an expression (which may be other than purely functional) in certain primitives the process is tree-like, with primitives being realised as further expressions in lower more machine oriented languages.

2.3 Basic Method

Our compiler, which is for a simple set language, works on this structured pattern. The user is presented with certain high level set primitives and for a given task produces the top portion of the tree. The compiler can complete the tree as it has structured definitions/

definitions of the given primitives in terms finally of the target or machine primitives. Each level in these structured definitions corresponds to a well-identified domain. The task is to flatten the tree and produce if possible a single expression in the target primitives that performs the task specified by the topmost function, given appropriate data domain mappings to encode the inputs and decode the outputs i.e. representations.

Note that we already differ from normal compilers for functional languages where the tree structure is retained after compilation and accommodated to the linear nature of the machine by means of interpretive run time systems performing function entry and exit.

The way that we proceed is top down, rewriting each expression into another expression. However we do not proceed by the normal method of replacing each primitive by its definition (in-line code introduction) followed by optimisation. We compile as much as possible via the semantics of the procedures. At a given level we examine as much as possible the semantic content of the given program or function and then reimplement this in the most appropriate manner in the next level. This consists of four stages.

(i) We extract the semantic content or transformation induced by the program.

(ii) In a planning or lookahead stage we consider the future implementation of this transformation and using knowledge appropriate to this domain adjust it so that it is still effectively the same transformation but is in a form more suitable for future implementation.

(iii)/

(iii) We then project this transformation into the next (lower) domain. By this we mean produce a transformation in the next domain which represents the previous transformation under the appropriate domain mappings.

(iv) Our compiler has certain automatic program writing ability that enables it to take this projected semantic transformation and produce a program in the lower domain that performs this transformation and is appropriate to that domain, a task similar to that proposed for the heuristic compiler of Simon (1963).

Note that our 'optimisations' are basically performed in (ii) and consist of manipulations in a clearer semantic domain and not manipulations of program text.

Each primitive at a given level in the tree has two faces as far as our compiler is concerned. It has a semantic face that allows the content of functions written using these primitives to be extracted and it has an implementation face, allowing the compiler to know about the future (lower down the tree) implementation of primitives, and it is thus able to rearrange the transformations in such a way as to facilitate optimising interactions in future implementations. The main optimising interactions are:-

(i) Linking, where two primitives have a common loop structure and may be merged if they occur in certain combinations.

(ii)/

(ii) Repeated expression elimination.

(iii) Construction reduction (doing less constructing of data structures by using assignment into structures and sharing). This can be thought of as compile time garbage collection.

We thus rewrite our programs to facilitate these optimisations. The stages are facilitated if we can keep the changes between stages as small as possible. Only as much structure as is needed to give some point to the step should be introduced so that we effectively deal with one language feature at a time.

It is usual in producing a program to have a choice of several ways of implementing primitives. Often the crucial choices are the ones of choosing representations for the data objects. It is a merit of structured programming that these decisions are separated from the task of producing the basic algorithm. Our compiler shares these advantages. It offers the user a choice of two representations for the abstract data objects. At present the choice is left to the user although the structure and information is there to allow the compiler to make the decision.

One must also distinguish between the functions on the tree defined by the user and those known to the compiler. For the latter we have information about their semantics and implementation. For the former, one would need to discover facts about their semantics and implementation. Our compiler can do this only to a limited extent.

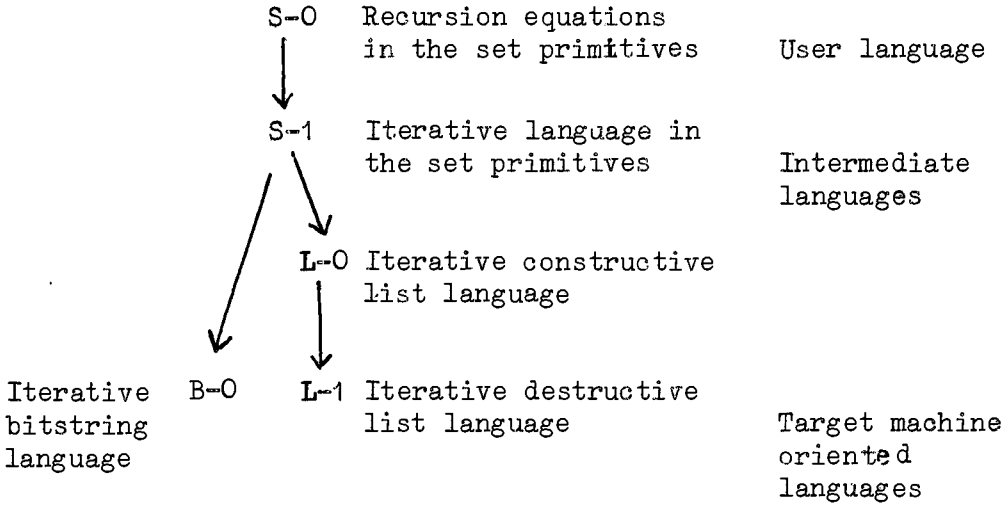
2.4/

2.4 Outline Description

We will now describe in outline how our transformational compiler works. In order to make our discussions more concrete we will describe it by showing how it works on an example. This entails introducing the language and the semantic apparatus used for dealing with programs written in these languages in this chapter though we do so informally and only to the extent needed to describe our methods. A fuller description and semantic definition will be given in the following chapter describing the implementation and the program writing algorithms.

2.4.1 Languages and Domains

To give the reader a guide we present here a summary of the relationships between the languages we are about to introduce



The abstract language that we choose for our compiler was one that would help people who wish to write programs involving manipulation of/

of sets of integers. Although this is not an area that is very rich in natural and useful algorithms the area does have an abstract structure and the sets have two competing representations, viz. as lists or as bitstrings.

2.4.1.1 User Language

The language in which the user presents his algorithms to the compiler is a purely functional recursive language.

As a pilot project we have chosen to implement a simple set language following Ambler and Burstall (1971) who developed a set package to allow the user to write programs using set primitives and offered a choice of representations. Much of the work applies to other systems and we discuss this in chapter 5.

The input consists of sets of mutually recursive equations in the set primitives, union, subtract, nullset, consset, choose and minus. These have the following informal semantics which we will give more formally in 3.2.1.

union(s1,s2) = s1 $\cup$ s2

subtract(s1,s2) = s1/s2

consset(x,s) = {x} $\cup$ s

minus(x,s) = s/{x}

nullset(s) = s=$\emptyset$

choose(s) = x such that x$\in$s

and s1 = s2 => choose(s1) = choose(s2)

The only one worthy of remark is choose, which is an arbitrary but deterministic function.

For/

For example, a program that defines union in terms of choose, minus and consset consists of the single recursive equation

$$\text{union}(s1,s2) = \begin{cases} \text{nullset}(s1) \longrightarrow s2 \\ \text{not}(\text{nullset}(s1)) \longrightarrow \text{consset}(\text{choose}(s1), \\ \qquad\qquad \text{union}(\text{minus}(\text{choose}(s1),s1),s2)) \end{cases}$$

We will call this language S-0.

2.4.1.2 Target Languages

The target languages in which we wish to implement these algorithms we will call L-1 and B-0. L-1 is basically an iterative, non-functional list programming language using the usual list primitives and B-0 is an iterative, non-functional language using bitstrings, one dimensional boolean arrays.

We avoid the use of unrestricted goto's to make our programs more easily understandable and amenable to manipulation and optimisation.

Iteration in both these languages is taken care of by the use of 'canned' loops, namely while statements and applists (in L-0 and L-1).

Applist applies a function to every member of a list and is defined as

```
Applist(x1,f) = while not(null(x1))
                    do [f(hd(x1)); tl(x1)->x1];
```

The results of Ashcroft and Manna (1971) show that we lose no power in avoiding unrestricted goto's.

In/

In both these languages (B-0 and L-1) destructive assignment is allowed, that is, assignment to parts of structures, instead of only allowing assignment to variables. Two examples will serve to give the flavour of these languages.

(i) A program in L-1 to change the nth element of a list l to x is simply

```
l->lcopy;
while n>1 do [n-1->n; tl(lcopy)->lcopy];
x->hd(lcopy)
```

(ii) A program in B-0 to concatenate two bitstrings.

```
Length(s1)->n1; Length(s2)->n2;
init(n1+n2)->s3;
1->n; while n≤n1 do [intsub(n,s1)->intsub(n,s3); n+1->n]
1->n; while n≤n2 do [intsub(n,s2)->intsub(n+n1,s3); n+1->n]
```

As we are using lists and bitstrings to represent sets we must define the representation relationship.

A list L represents a set S iff each member of S occurs just once in L and L has no other elements, e.g. (1,2,3) represents {3,2,1} but (1,2,3,1) does not.

A bitstring B represents a set S of integers iff the x'th component of B is 1 whenever $x \in S$ and 0 whenever $x \notin S$.

2.4.1.3 Intermediate Languages

In order to give the compilation process structure we introduce intermediate languages.

Our first task is to remove as much recursion as possible. This is represented as a transition between S-0 and a language we call S-1. S-1 has the same basic primitives as S-0 but it has assignment (to variables only) and its iterative structure is the same as L-1 or B-0. Thus the previous example of a definition in S-0 of union in terms of consset, choose and minus becomes in S-1.

```
while not(nullset(s1)) do [choose(s1)->x; minus(x,s1)->s1
                                    consset(x,s2)->s2]
```

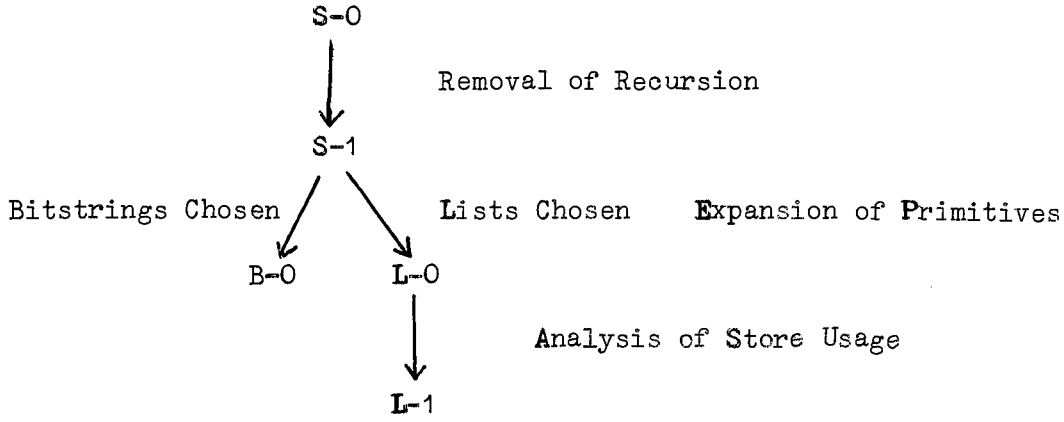
The next stage down the hierarchy concerns the expansion of the set primitives (e.g. union,choose) in terms of the list or bitstring primitives. It is here that a choice concerning the representation is made by the user. For bitstrings the translation is straight to B-0. If lists are chosen the translation is first to a program in a language L-0 which is similar to L-1 but is purely constructive in that it only allows assignment to variables and not to structure.

The previous example of a program written in L-1 to replace the nth element of a list l by x would have to be written thus in L-0.

```
nil/
```

```
nil->res;
applist(l,lambda z; if n=1 then cons(x,res)->res
                           else cons(z,res)->res
                           n-1->n)
rev(res)->l;
```

Assuming rev is a primitive for the moment (which it is not in L-0). The final stage, if lists have been chosen as a representation, is the translation of the program from L-0 to L-1. The transitions can be summarised as



The steps S-0->S-1, S-1->B-0 and S-1->L-0 are the steps where the main flattening of the functional tree takes place.
[By a completely flattened program at a given level we understand an iterative (non-recursive program) composed entirely of primitives of that level.]

The removal of recursion S-0=>S-1 attempts to convert a tree of mutually recursive equations to a single iterative program. The steps S-1=>B-0, and S-1=>L-0 involve looking at the semantics of this program, manipulating this transformation according to set theoretic laws/

laws and rewriting it in such a way that when the primitives are expanded into B-O or L-O primitives the most beneficial interactions occur enabling loops to be linked.

The step L-O=>L-1 involves no expansion of primitives. It involves looking at the way the program uses store and attempting to economise on its use of store, by looking after many of the responsibilities usually allocated to the garbage collector at run time and rewriting the program to remove as many of these responsibilities as possible.

2.4.2 Outline of Compiler

We are now in a position to describe these steps in some detail. This will be done mainly through the use of an example. We show how the compiler deals with the definition of union illustrated earlier

$$\text{union}(s1,s2) = \begin{cases} \text{nullset}(s1) \longrightarrow s2 \\ \text{not}(\text{nullset}(s1)) \longrightarrow \text{consset}(\text{choose}(s1), \\ \qquad \text{union}(\text{minus}(\text{choose}(s1),s1),s2)) \end{cases}$$

2.4.2.1 Removal of Recursion

We do not implement a general method of recursion translation such as those proposed by Strong (1970) and Garland and Luckham (1971). We find that many recursive forms often met in practical/

practical programming fall outside the scope of these methods. As these methods are based on schemas they do not make use of any of the properties of the primitives such as associativity or commutativity and any tree-like recursion resulting from polyadic functions, for example the factorial function, is often only translatable because of certain properties of its primitives. Conversely the schemas for which these general methods are most impressive, viz. highly inter-related monadic schemas, are very little met in practical programming. Instead, we concentrate on bringing as much semantic information as possible to bear on the translation. The system classifies the input according to its structural form by matching it with one of a repertoire of stored schemas eaching having a known iterative translation. It then looks at the semantic properties of the primitives making up the function to see if a translation is possible.

This approach is not rigid in that one can vary the matching filter to classify more functions in one structural form and then do more work in the second phase, which performs a semantically based translation. It is a matter for empirical testing whether an adequate number of functions can be handled by a manageable number of patterns. We have not been concerned with the completeness of our algorithm, although it will translate many functions that complete algorithms in the schemata sense will be unable to. We visualise that its use would be in partnership with a general complete algorithm of the kind proposed by Strong.

The/

The first task on being given a set of recursion equations is to partition it into sets of mutually recursive equations. These can be translated independently and then recombined.

In the case of our example this partitioning is trivial as there is only one equation.

Our matching process classifies this equation into the class associated with the **dyadic** functions with a structure similar to factorial. Generally this form is untranslatable, but as consset has the property that

consset(α,consset(β , γ)) = consset(β ,consset(α , γ))

our program manages to produce the translation,

```
s2->Ans;
 while not(nullset(s1))
   do [choose(s1)->t; consset(t,Ans)->Ans;
       minus(t,s1)->s1]
```

2.4.2.2 Choice of Representation, Implementation in Lower Level-Description

The basic process we describe in this section is used for all transitions between language levels except for the recursion translation stage (S-0 to S-1). Different transitions however emphasise different aspects of this process and will be described separately.

2.4.2.2.1/

2.4.2.2.1 S-1=>B-O, S-1=>L-O Transformation Extraction

The approach to compilation we adopt owes much to Milner's work on Simulation, Milner (1971). Given a piece of program text (say in S-1) we first analyse the transformation induced by this program text on a symbolic state vector. A state vector gives a value for every variable occurring in the program. It therefore defines a state the program could be in. A symbolic state vector is one in which the values given to variables are names instead of actual values. We extract the transformation from a piece of program text by evaluating its effect on the symbolic vector, producing a pair of before and after state descriptions.

2.4.2.2.2 Planning

It is here that the choice of the next lower language is made, at present by the user. This stage is the planning or optimisation stage. We will describe it as occurring before the next stage, projection, but as we will see later the separation is not critical. This stage consists of rearranging the state descriptions so that they will be optimally implemented in the next lower language chosen. This involves rearranging the state descriptions using algebraic laws appropriate to the higher level, but the evaluation function that measures the optimality of the rearranged transformations is couched in terms of the future implementation proposed for these primitives. Thus though the state descriptions manipulated are in terms/

terms of the higher level primitives and it is their algebraic laws that we utilise, they may be thought of as shorthand for their proposed expansion.

This stage can also be truly called a planning stage as the search for an optimal implementation is done in a simplified space, and proposed solutions may be sub-optimal although the computation required to find them is much less than would be needed to find an optimal solution.

2.4.2.2.3 Projection

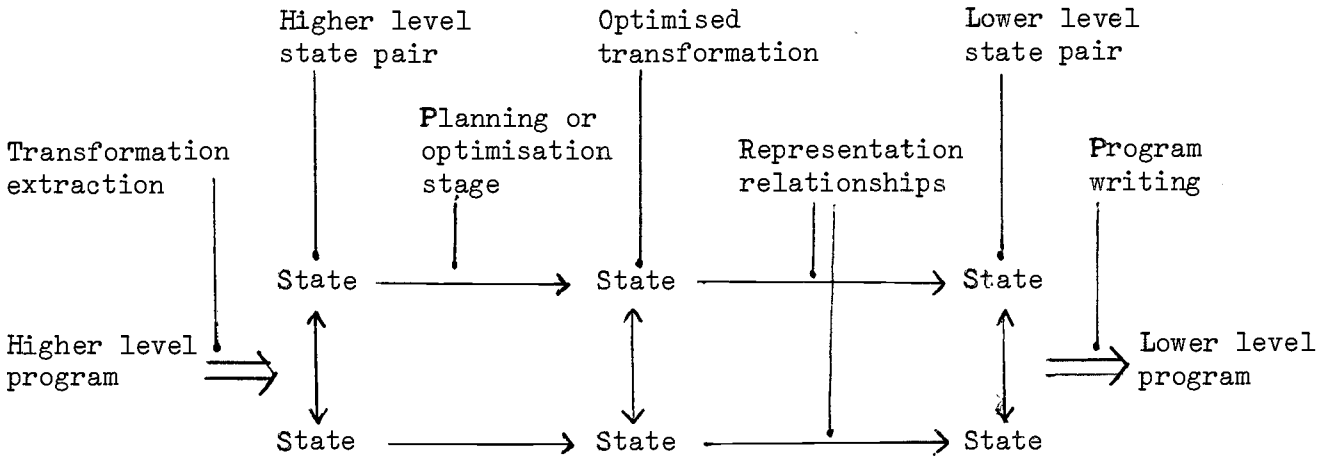
The result of the previous stage is a pair of symbolic state vectors representing the transformation extracted from the program text. We now project these symbolic state vectors into the lower domain chosen using the representation relationship.

2.4.2.2.4 Program Writing

Given this pair of lower level state descriptions we then automatically write a program to perform this transition in the lower level language as efficiently as possible.

Thus/

Thus to summarise



The higher level transformation is given in the form of a program, re-expressed as a pair of symbolic state descriptions. This transformation is rearranged in the planning stage and then translated in the projection stage into a corresponding pair of lower level state descriptions. From these the lower level program is produced. Ideally we would like to be able to take for our pieces of program text arbitrary programs. Then this process would be devoid of any form-directed compilation and able to take a program at any given level purely as a specification and re-interpret this entirely. This of course requires the ability to extract the semantics of whole programs, which we are unable to do. Our basic unit for the process is a single while statement. Generally we are unable to alter whiles introduced by the recursion translation process and the overall iterative structure introduced there remains to the end. However, there are several special cases that our compiler can recognise where it can alter this structure.

2.4.2.3/

2.4.2.3 Implementation in Lower Level-Example

Our first task is to segment the program into sections to be dealt with independently. A section is usually a while statement together with any piece of the program text following the while statement up to the next while statement or end of program.

Our example divides into two sections

```
        s2->ans
and     while not(nullset(s1)) do [choose(s1)->t; consset(t,ans)->ans;
                                   minus(t,s1)->s1]
```

Our compiler can do nothing for the first section which remains as it is. On looking at the second section, it generally divides it up into three parts: the predicate of the while statement, its body and the rest of the section. In this example only the first two are present.

2.4.2.3.1 Transformation Extraction

We now investigate the semantic transformations involved. The semantic apparatus that we use will be described in detail in the following chapter, chapter 3. At this level (S-1) the semantics of our program is not unnaturally expressed in set-theoretic terms. We require the variables in our programs to be typed, so that we know which variables can take sets as values and those that take objects as values. Persuasive reasons have been advanced for typed/

typed languages. We do not enter this controversy here but note that typing makes translation easier.

Thus in our example s1, s2 and ans would be of type set and t of type object. We invent symbolic names for our semantic objects and set up an initial state vector thus

```
 s1     s2     ans    t
 ↓      ↓      ↓      ↓
set1   set2   set3   ob1
```

Our semantic transformation is basically derived as expressions over the set and object names. However we also have the ability to give symbolic names to members of specific sets. Thus if we attempt to choose an element from a set we can give it a specific symbolic name. **Each set has associated with it a membership list which is a list of** those object names which are definitely in the given set (this is discovered from the program text).

The transformation is extracted from the program text basically by interpreting the state vector through the text. If we encounter a 'choose' we attempt to discover whether the set already has any designated members. If it has, we choose the first one to be the value of this choose, if not we invent one and choose it.
To illustrate this process we will first annotate points in our example program thus.

```
      A                       B                   C                           D
while | not(nullset(s1)) do [ | choose(s1)->t;    | consset(t,ans)->ans;      |
                                                  E
                                minus(t,s1)->s1   | ]
```

We/

We may wish to combine some of the work done in evaluating the predicate expression controlling the execution of the **while statement** and evaluating the other two (in this case one) bodies of the section, thus we include the predicate value in our interpretation by giving it a variable denoted **PV**. When rewriting the program we insist that PV must be evaluated before any assignment to program variables takes place. Any values that may be used again later are stored as temporary values. The process of interpreting the state vector thus produces these intermediate stages.

A

s1	s2	ans	t	PV
↓	↓	↓	↓	↓
set1	set2	set3	ob1	ob2

Membership list of set1,set2,set3 = nil.

B

s1	s2	ans	t	PV
↓	↓	↓	↓	↓
set1	set2	set3	ob1	not(nullset(set1))

C

s1	s2	ans	t	PV
↓	↓	↓	↓	↓
set1	set2	set3	ob3	not(nullset(set1))

and the membership list of set1 becomes [ob3]

D

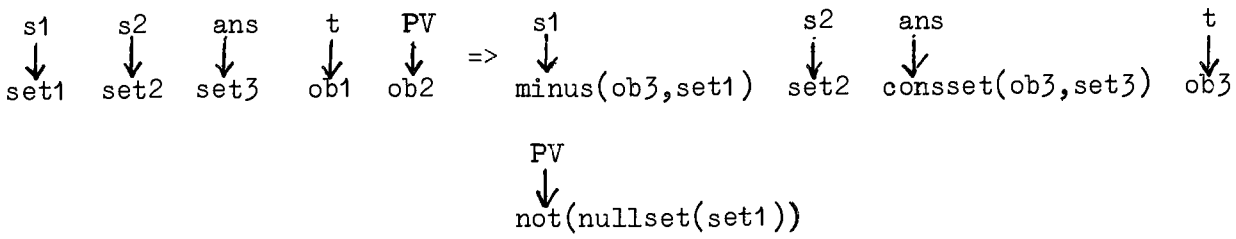
s1	s2	ans	t	PV
↓	↓	↓	↓	↓
set1	set2	consset(ob3,set3)	ob3	not(nullset(set1))

E

s1	s2	ans	t	PV
↓	↓	↓	↓	↓
minus(ob3,set1)	set2	consset(ob3,set3)	ob3	not(nullset(set1))

Thus the overall transition is

s1/



with property lists

set1 - [ob3]

set2 - nil

set3 - nil

2.4.2.3.2 Processing the Transformation

We are now in a position to consider the implementation of these primitives into the next level L-0 or B-0. We therefore give the general definition of these primitives in the two languages:-

B-0

```
nullset(s) = true->res; length(s)->l; 1->n;
             while n≤l do
               [if intsub(n,s) = 1 then false->res; n+1->n]
               res

choose(s) = length(s)->l; 1->n;
             while n≤l do
               [if intsub(n,s) = 1 then n->res; n+1->n]
               res

minus/
```

```
minus(x,s) = 0->intsub(x,s);s

consset(x,s) = 1->intsub(x,s);s

union(s1,s2) = copy(s1)->res;
          length(s2)->l; 1->n;
          while n≤l
            do [if intsub(n,s2) = 1 then
                1->intsub(n,res); n+1->n]
            res

subtract(s1,s2) = copy(s1)->res;
          length(s2)->l; 1->n;
          while n≤l
            do [if intsub(n,s2) = 1 then 0->intsub(n,res);
                                          n+1->n]
            res

member(x,s1) = intsub(x,s1) = 1
```

L-0

```
nullset(s) = null(s)
choose(s) = hd(s)
consset(x,s) = if member(x,s) then s else cons(x,s)
minus(x,s) = nil->res;
             applist(s, λ y; if not(y=x) then
                             cons(x,res)->res)
             res
```

union/

```
union(s1,s2) = s2->res;
               applist(s1, λ x; if not(member(x,s2)) then
                                    cons(x,res)->res)
               res
subtract(s1,s2) = nil->res;
               applist(s1, λ x; if not(member(x,s2)) then
                                    cons(x,res)->res)
               res
```

The next step is to attempt to simplify some of the set expressions. We note that in B-O nullset and choose have to be interpreted iteratively, similarly consset and minus in L-O. We try to avoid this. We note that if we know something about the membership of the sets in question this is not always necessary. We have two ways of avoiding this (i) calculating whether specific symbolic objects are definitely in or definitely not in a specific set and (ii) ordering the known members of sets so that when they are projected into lists they will be accessible without iteration.

(1) We define two functions in the meta-language, defon, defnoton.

defon, defnoton: expressions in setnames->lists of members which tell us what members are known to be in or known to be not in given expressions of type set. They are defined inductively over expressions thus:-

defon/

defon($\mathcal{E}$) = cases $\mathcal{E}$: setname: membership list of $\mathcal{E}$ *

nilset: $\emptyset$

union($\mathcal{E}_1, \mathcal{E}_2$): defon($\mathcal{E}_1$) $\cup$ defon($\mathcal{E}_2$)

subtract($\mathcal{E}_1, \mathcal{E}_2$): defon($\mathcal{E}_1$) $\wedge$ defnoton($\mathcal{E}_2$)

consset($\mathcal{E}_1, \mathcal{E}_2$): $\{\mathcal{E}_1\} \cup$ defon($\mathcal{E}_2$)

minus($\mathcal{E}_1, \mathcal{E}_2$): defon($\mathcal{E}_2$)/$\{\mathcal{E}_1\}$

defnoton($\mathcal{E}$) = cases $\mathcal{E}$: setname: $\emptyset$

nilset: $\emptyset$

union($\mathcal{E}_1, \mathcal{E}_2$): defnoton($\mathcal{E}_1$) $\wedge$ defnoton($\mathcal{E}_2$)

subtract($\mathcal{E}_1, \mathcal{E}_2$): (defnoton($\mathcal{E}_1$) $\cup$ defon($\mathcal{E}_2$))
$\cup$ (defon($\mathcal{E}_1$) $\wedge$ defnoton($\mathcal{E}_2$))

consset($\mathcal{E}_1, \mathcal{E}_2$): defnoton($\mathcal{E}_2$)/$\{\mathcal{E}_1\}$

minus($\mathcal{E}_1, \mathcal{E}_2$): $\{\mathcal{E}_1\} \cup$ defnoton($\mathcal{E}_2$)

Given these functions we can make the following simplifications.

consset (ob1, $\mathcal{E}$)

(i) if ob1$\in$defon($\mathcal{E}$) then $\mathcal{E}$

(ii) if ob1$\in$defnoton($\mathcal{E}$) then in L-O this can be simplified to

cons(ob1, $\mathcal{E}$)

minus

---

* The cases notation that we will use for describing algorithms was developed from the switch on type introduced into CPL by M. Richards (1967), by R.M. Burstall (1969). As well as being a conditional switch it allows structures to be decomposed into their constituents.

minus (ob1, $\mathcal{E}$)

(i) if ob1ϵdefon($\mathcal{E}$) then in L-0 we may be able to arrange it so that this is tl($\mathcal{E}$)

(ii) if ob1ϵdefnoton($\mathcal{E}$) then $\mathcal{E}$

The first case for consset and the second case for minus are not likely to occur. As these relations are discoverable from the program text, they would also be apparent to a programmer. They may occur after recursion translation.

The next stage depends on which language is chosen. It is at this stage that a choice of representation would have to be made. We could make the choice by evaluating the transformation in terms of how much work it involves in each of the competing domains. These would be upper bounds as they would not be able to take into account any optimisations we may be able to perform later.

For this small example the planning or manipulation stage does not alter the state transformations in any of the two implementations. We therefore reserve its detailed description until we have an example where it has a significant effect. (3.2.2.3).

2.4.2.3.3 Implementation in L-0

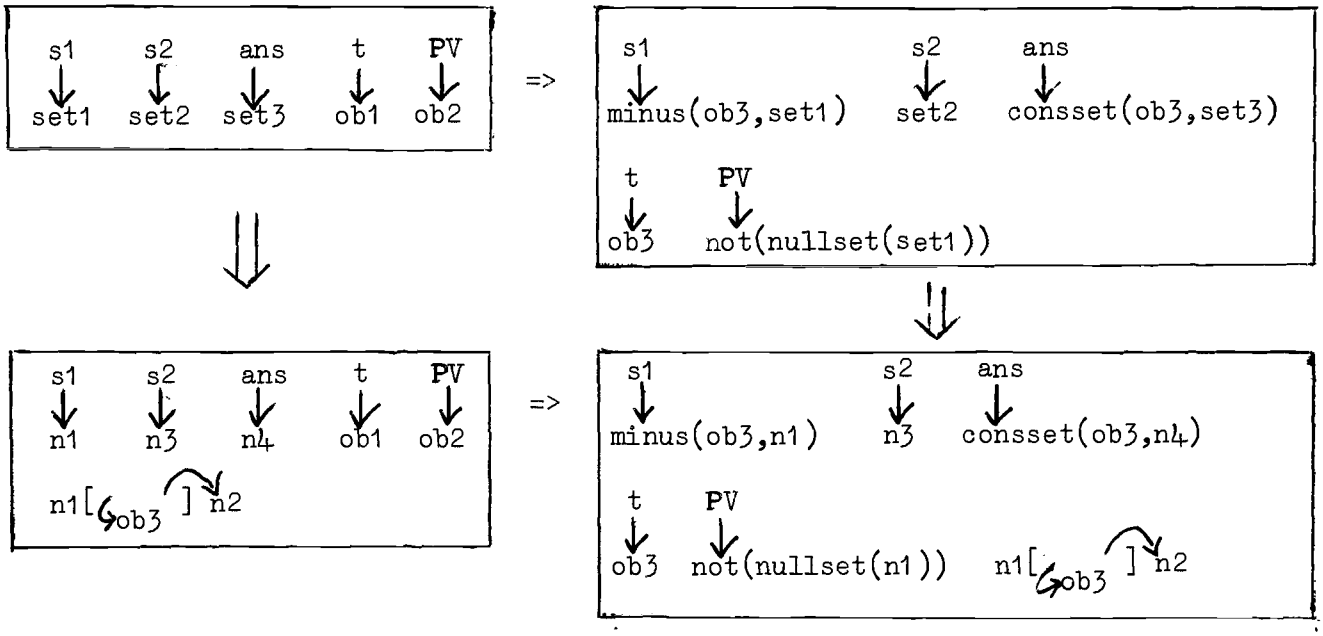
If we choose L-0 the next task is to project the transition into the lower domain. We must therefore briefly explain our semantic apparatus for this domain. Transformations are again expressed as pairs of state vectors. In the place of sets we have lists./

lists. Lists are represented symbolically as chained links of named nodes. Each list is given just enough structure to enable us to describe the transformation. Variables can again take as values free expressions over symbolic objects or nodes. The main work of the projection is in attempting to arrange the nodes of the list (representing the members of the set) in such a way as to be able to implement them easily.

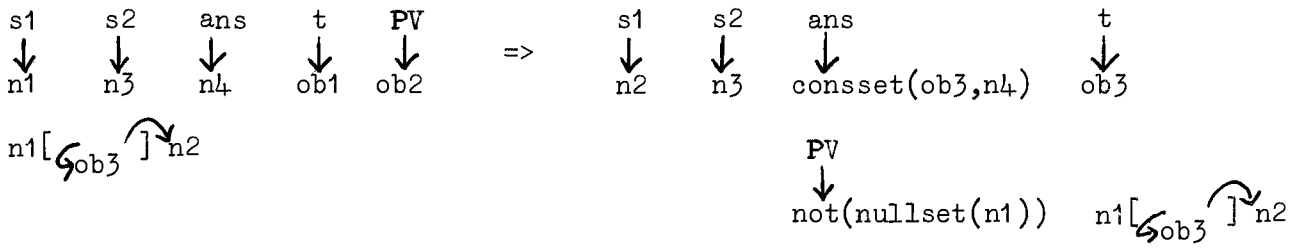
As there is no possible linking of the iterative structure of the primitives in this example we will leave its description for the section on the implementation in B-O where it is of importance.

In this example one set has only one named member so the projection is easily done. The symbolic set set1 is represented by the symbolic list n1[$\curvearrowright$ob3$\curvearrowright$]n2 where n1 is the name of the first node whose head points to ob3 and whose tail points to the node n2. n2 can be thought of as the rest of the list whose structure does not enter into our computation.

Thus our projection is

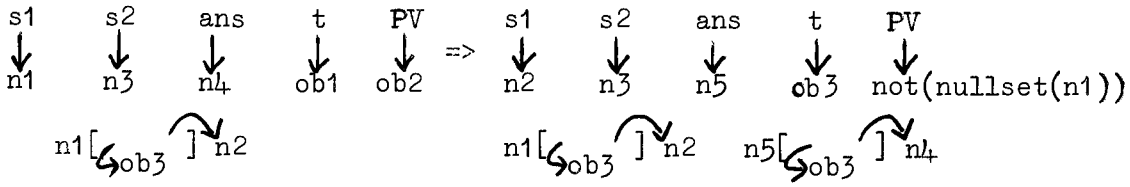


where minus, consset, not and nullset should now be thought of in the lower domain as shorthand for their definition in this domain. The program can now see that it can simplify the minus(ob3,n1) into n2. Thus our final transition is



The task now is to write the program to perform the above transformation and to expand the primitives.

We would like to make a small change here that will enable us to better illustrate a later mechanism. We were unable to reduce the consset(ob3,n4) as we can derive no knowledge of the membership of set3. However we could have given our compiler a slight interactive capacity so that when it saw a possible optimisation it could ask the user questions. Feldman (1972) points out that this capacity will be essential. Thus if the user knew that the sets were always to be disjoint we could further simplify the transition to



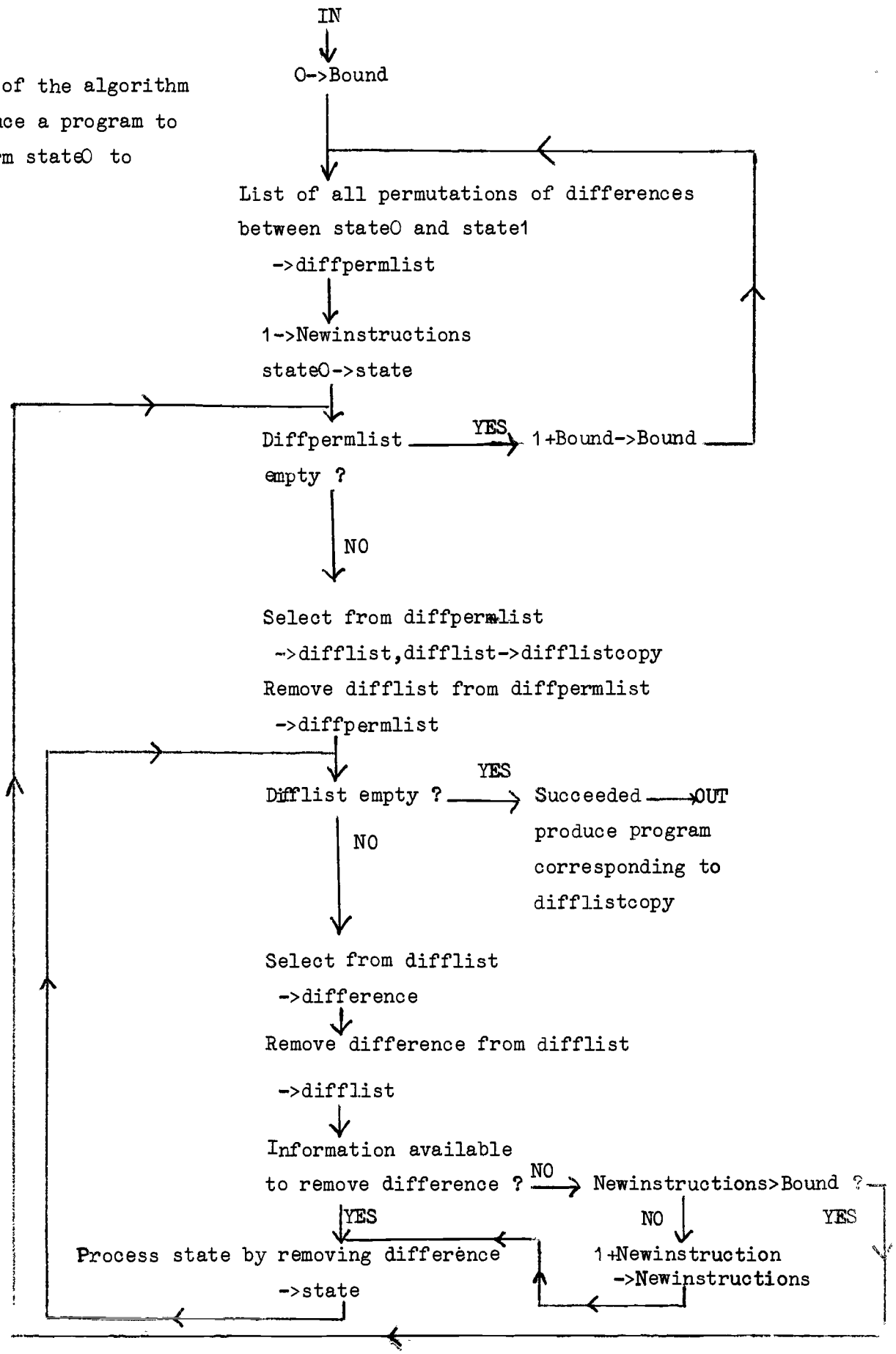
where n5 is the name of the new node introduced by cons(ob3,n4).
The task is now to produce a program from the state transformation.

Our/

Our basic method is given in outline flowchart form overleaf. It will be described more fully in the later chapter on implementation. We first calculate a set of differences between the states. A difference occurs when a variable has differing values in the two states or a new node is introduced. [At this stage the list structure can only be altered by adding new nodes.] There exist ways of removing these differences by assignment instructions, provided that certain information, in this case the information required to calculate the value of the left hand side of the assignment, is available. Our program attempts to order the differences in such a way that they can be removed without any necessary information being lost. If there exists such an order our program produces the program corresponding to that order which is guaranteed to be optimal, as each difference requires at least one assignment statement to remove it. If this is not possible our program allows itself one extra instruction, in order to remember information that would otherwise get lost, and again attempts to order the differences. The program continues in this way until either a program is found or an effort bound is exceeded. One restriction is that the difference corresponding to 'PV' in the transformation must be removed first. Any calculation that is done in the course of this calculation that will be useful later is stored in temporary variables until after the PV difference has been removed. On being given the above state transformation our program produces the following order of differences

(i)/

Outline of the algorithm to produce a program to transform state0 to state1



(i) PV (ii) t (iii) s1 (iv) n5 (v) ans

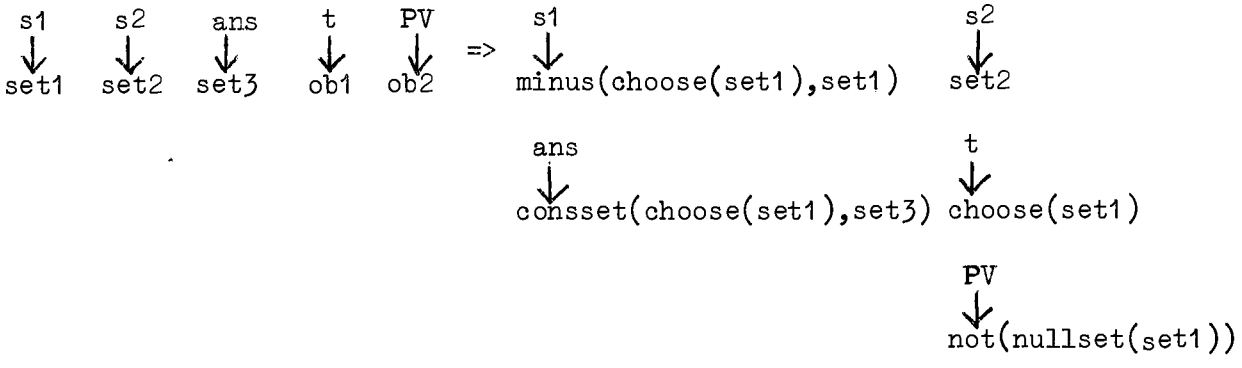
and produces the following program

while not(null(s1)) do [hd(s1)->t; tl(s1)->s1; cons(t,ans)->ans]

2.4.2.3.4 Implementation in B-O (S-1 => B-O)

We now back up and show how our compiler implements the transition in B-O.

Here none of the investigation of the discoverable memberships of sets is useful as in B-O both consset and minus are implemented as single operations, so we do not bother to investigate set membership or simplify choose, minus or consset, thus the transition is



What is important is that choose and nullset have similar iterative structure in this implementation and can be linked. An important point to note before we proceed is that the combining of the loops of nullset and choose is done as it were using the S-1 primitives [with a knowledge of their future implementations] and then expanding them into an already optimised form, not expanding the code and then optimising.

An/

An improvement would be to have several competing implementations of S-1 primitives within a single representation choice and allow the compiler to choose the best one for this particular purpose, or to have the compiler write or modify implementations according to its need. Our compiler has none of these capacities.

The same program writing mechanism was used for the S-1 => L-0 transition is again used. In this transition, though, iterative linking plays an important part, so we shall describe it in some detail. During the calculation of differences the program notes if any of the expressions it needs to calculate can be linked iteratively. Basically they can be linked if and only if they are in the same iterative group (as nullset and choose are), they have the same iterative argument, and one does not occur as a subexpression of the other. The program has an evaluation mechanism that attempts to evaluate whether the effort saved in doing things on the same loop is worthwhile. In this case it is, and it links nullset(set1) and choose(set1). As nullset(set1) is part of the 'PV' difference no assignment can take place until after 'PV' has been calculated.

Our program therefore produces

```
while [length(s1)->l; 1->n; true->res1;
        while n≤l do [if intsub(n,s1) = 1 then[false->res1; n->res2;]
                      n+1->n]
    not(res1)] do [res2->t; 0->intsub(t,s1); 1->intsub(t,ans)]
```

Generally/

Generally that would be as far as our compiler goes on this branch (B-0). As we said earlier the compiler has no general ability to alter the structure of a complete while after introduction. Usually those whiles that are introduced by the recursion translation remain to the end. However we have added to our compiler the ability to recognise and deal with two special cases

viz. while nullset($\mathcal{E}$) do A

and while not(nullset($\mathcal{E}$)) do A.

As the second case is applicable to our example we choose that to explain.

If it is discoverable that performing A once removes exactly one element from $\mathcal{E}$, this form can be rewritten as

```
length(ℰ)->L; 1->n; while n≤L do [A; 1+n->n]
```

Applying this to the transition our compiler gets as its final B-0 program.

```
length(s1)->l; 1->n;
  while n≤l do [if intsub(n,s1) = 1
                then [n->t; 1->intsub(t,ans); 0->intsub(t,s1)]
                n+1->n]
```

2.4.2.4 Destructive Implementation L-0 => L-1

There/

There remains one step left to describe. The program produced by projection from S-1 to L-0 was

```
s2->ans;
while not(null(s1)) do [hd(s1)->t; tl(s1)->s1; cons(t,ans)->ans]
```

We now describe how our compiler re-implements this in L-1 which is a destructive list language (i.e. one that allows assignment to parts of structure, as in X->hd(Y)).

This part of our program concentrates on analysing the store usage as there are no primitives that expand further.

Again the only section of the above program that our compiler changes is

```
                          2                    3                4                    5 6
while not(null(s1))! do [hd(s1)->t; ! tl(s1)->s1; ! cons(t,ans)->ans!]!
```

The starting state can be described symbolically as

```
s1    s2    ans    t
↓     ↓     ↓      ↓
n1    n3    n4     ob1

n1->n2              n3.    n4.
↓
ob3
```

We here present our descriptions of the structure of the front portions of our symbolic lists in pictorial form. How much structure we actually need at the beginning of our symbolic lists to adequately describe the transformation is discovered by starting off with a node representing/

representing each list and whenever we access some of this structure (which must be on the front of the list) backing up, adding as much structure as is needed and restarting. Some of this information could have been deduced from the fact that not(null(s1)) is true on this branch.

The following states are derived by interpreting our symbolic state on the above program

At	2	s1	s2	ans	t	n1->n2, n3, n4
		↓	↓	↓	↓	↓
		n1	n3	n4	ob1	ob3
	3	s1	s2	ans	t	n1->n2, n3, n4
		↓	↓	↓	↓	↓
		n1	n3	n4	ob3	ob3
	4	s1	s2	ans	t	n1->n2, n3, n4
		↓	↓	↓	↓	↓
		n2	n3	n4	ob3	ob3
	5	s1	s2	ans	t	n1->n2, n3, n5->n4
		↓	↓	↓	↓	↓
		n2	n3	n5	ob3	ob3 ob3
and at	6	s1	s2	ans	t	n1->n2, n3, n4
		↓	↓	↓	↓	↓
		n1	n3	n4	ob1	ob3

So the only effective transition we need to implement is

s1	s2	ans	t	n1->n2, n3, n4
↓	↓	↓	↓	↓
n1	n3	n4	ob1	ob3

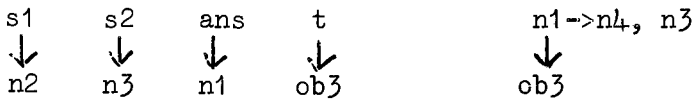
to/

to

We now project this transformation into the lower domain, in this case **L-0=>L-1**. The semantic apparatus we use for the two domains is the same and so the projection is the identity.

The optimisation of the transition we now perform is to try and economise on store usage by stopping the introduction of new cells. Our program notes that n1 is not accessed in any way at the end (as far as this piece of program is concerned) and that its hd has the same value as our new node n5, thus an equivalent transformation involving less rearranging (fewer differences).

to



Our compiler now writes the program to realise this transition. The method is as before, calculating differences and attempting to find an order in which they can be removed. In this case differences are more complex as they may involve rearranging of hd, tl pointers/

pointers and our compiler has to check that side-effects are not losing information. Our program outputs the following program to perform the above transformation.

```
ans->t;
s1->ans;
tl(s1)->s1;
t->tl(ans);
hd(ans)->t;
```

which when plugged back in the earlier program gives

```
s2->ans;
while not(null(s1)) do [ans->t; s1->ans; tl(s1)->s1; t->tl(ans);
                                                    hd(ans)->t]
```

Summary

We have implemented the recursive definition

$$\text{union}(s1,s2) = \begin{cases} \text{nullset}(s1) \longrightarrow s2 \\ \\ \text{not}(\text{nullset}(s1)) \longrightarrow \text{consset}(\text{choose}(s1), \\ \qquad\qquad \text{union}(\text{minus}(\text{choose}(s1),s1),s2)) \end{cases}$$

in two programs viz.

s2/

```
s2->ans;
length(s1)->l; 1->n;
  while n≤l do [if intsub(n,s1) = 1
                then[n->t; 1->intsub(t,ans); 0->intsub(t,s1)]
                n+1->n]
```

and

```
s2->ans;
while not(null(s1)) do [ans->t; s1->ans; tl(s1)->s1; t->tl(ans);
                        hd(ans)->t]
```

The order of complexity of both of these implementations is n where n = length of s1.

If we had implemented our function by the normal interpretive compiler we would have had two program viz.

$$\text{union}(s1,s2) = \begin{cases} \text{nullset}(s1) \longrightarrow s2 \\ \\ \text{not}(\text{nullset}(s1)) \longrightarrow \text{consset}(\text{choose}(s1),\text{union}(\text{minus}(\text{choose}(s1),s1),s2)) \end{cases}$$

```
where nullset(s) = True->res; length(s)->L; 1->n;
                   while n≤L do
                   [if intsub(n,s) = 1 then false->res; n+1->n]
                   res
```

where/

```
where  consset(x,s1) = 1->intsub(x,s1); s1
where  minus(x,s1) = 0->intsub(x,s1); s1
where  choose(s)  = Length(s)->l; 1->n;
                    while n≤l do
                    [if intsub(n,s) = 1 then n->res; n+1->n]
                    res
```

A program of the order of complexity n^2

and

$$\text{union}(s1,s2) = \begin{cases} \text{nullset}(s1) \longrightarrow s2 \\ \text{not}(\text{nullset}(s1)) \longrightarrow \text{consset}(\text{choose}(s1),\text{union}(\text{minus}(\text{choose}(s1),s1),s2)) \end{cases}$$

```
where  nullset(s1) = null(s1)
where  consset(x,s1) = if member(x,s1) then s1 else cons(x,s1)
where  minus(x,s1) = nil->res;
                     applist(s1, λ y; if not(y=x) then cons(x,res)->res)
                     res
where  choose(s1) = hd(s1)
```

A program of the order n^3.

CHAPTER 3. IMPLEMENTATION

In this chapter we describe how we have implemented the ideas described in chapter 2.

3.1 Removal of Recursion

3.1.1 In this section we deal with how, whenever possible, we translate sets of recursion equations to iterative programs. It has long been held desirable to remove some recursion on compilation in order to discard some of the inefficiencies often inherent in the use of a stack. There has been some theoretical investigation of when and how it is possible to translate recursive schemas into equivalent iterative schemas e.g. Strong (1970) although as far as we know no actual compiler makes use of any of these ideas, although BBN LISP, Teitelman et. al. (1971) removes very trivial recursions at compilation. These methods usually are developed for free or uninterpreted schemas and thus rely on syntactic criteria for classifying schema and syntactic translation algorithms for converting suitable schema to equivalent iterative schema (i.e. ones that agree over a given range of interpretations of the basic symbols). Not all schemas are translatable and not all are translatable into iterative schemas of comparable efficiency. We, of course, are interested only in translations that maintain or improve efficiency, but our approach differs in several other respects. Firstly our inputs will be fully interpreted functions and not schemas, that is, the/

the function letters occurring will have known denotations; by solely dealing with schema translatability results we will be restricting ourselves unnecessarily. Also most of the positive translatability results have been confined to monadic schemas or linear recursive schemas and are unable to deal with K-fold recursions, Peter (1951), for $K \geq 2$. We feel that many of the forms often met in practical programming thus fall outside the scope of these methods, as often when it is natural to use recursion it is because the underlying data is tree-like (lists) or the algorithm involves polyadic functions. The schemas on which these methods seem most impressive are highly interrelated monadic or linear schemas. These are met very little in practical programming.

The translations that we achieve are of two types.

(i) Where the computation sequence of the iterative program produced is a rearrangement of the computation sequence of the recursive program but contains the same number of computation steps. In these cases we save the time and storage overheads and inefficiencies associated with the various stacking mechanisms and function entry and exit. The factorial function is an example here (see 3.1.3.4).

(ii) In other cases, for example the Fibonacci function (3.1.3.4), the tree grown by the recursive calls contains redundancies in that the same values are calculated at separate nodes. Our translation process attempts to recognise these cases and produce an iterative program whose computation sequence is shorter and contains no redundancies/

redundancies, these savings being on top of removing the stacking overheads.

The system that we propose and have implemented consists basically of a 'big switch' according to the structural form of the equations input followed by the investigation of the semantic properties of the primitives making up the function in an attempt to produce a translation. This method owes much to the Planner philosophy, Hewitt, C. (1971), in that it consists of a matching phase that attempts to match the input with a set of target patterns. Associated with each target pattern is a set of rules or procedures that investigate the semantic properties of the values produced by the matching process and produce a translation if one is possible. The work of Cooper (1966) provided valuable insights here. We do not aim at a theoretically complete translation algorithm although of course the method could be backed by a general method for the sake of completeness.

3.1.2 Source and Target Languages

We now define the problem. We do not restrict ourselves to the task of translating S-0 programs into S-1 programs but generally state the problem of translating recursion equations into iterative programs given that we know certain basic semantic properties of the function primitives making up the recursion equations.

3.1.2.1 Source Language

Most/

Most of the following definitions for the recursion equations follow Strong (1970).

For the source language, which we shall call, S-T, we have an alphabet B of base symbols, an alphabet P of predicate symbols, an alphabet X of argument symbols and an alphabet F of function letters. Each element $f \in F$ of rank n has associated with it a collection of n argument symbols $xf_1,\ldots,xf_n$ abbreviated to $\overline{xf}$. However, to re-emphasise that we are dealing with functions and not uninterpreted schemas, each $p \in P$ and $b \in B$ has associated with it a specific function and we are given certain properties of these functions.

A <u>tree</u> is an element of B of rank 0 or an element of X or an expression $a(t_1,\ldots,t_n)$ where a is an element of B, P or F of rank n and $t_1,\ldots,t_n$ are trees.

A <u>branched recursion equation</u> is of the form

$$f(\overline{xf}) = \begin{cases} p_1 \longrightarrow t_1 \\ \vdots \quad\quad \vdots \\ p_m \longrightarrow t_m \end{cases}$$

where f is an element of F, $p_1,\ldots,p_m$ are trees with symbols from $X \cup B \cup P$ whose topmost node is a symbol from P and $t_1,\ldots,t_m$ are trees with symbols from $X \cup B \cup P \cup F$.
Note that we do not allow recursive calls to occur in the predicate trees p_i. This is not an essential restriction for us but it simplifies the exposition. We also require that the p_i for a particular recursion equation be exhaustive and mutually exclusive.

A/

A branched recursion set is a system of branched recursion equations with distinct left hand sides. One of the left hand function letters is distinguished as the name of the branched recursion set.

3.1.2.1.2 Semantics

The semantics of branched recursion sets are straightforward.

Consider a given domain D. A specific recursion equation set $E(f,\overline{xf})$ with name f of rank n defines a partial function $D^n \to D$.

Each base function $b \in B$ of rank m defines a partial function $D^m \to D$. A computation of $E(f,\overline{xf})$ for a particular value of $\overline{xf} = \langle d_1,\ldots,d_n \rangle \in D^n$, is defined as follows. The computation is a sequence of trees $t_0,\ldots,t_k$ where $t_0 = f(d_1,\ldots,d_n)$ and t_{i+1} is derived from t_i by the following rule. Select the leftmost innermost sub-tree whose root $\in F$, say $g(g_1,\ldots,g_m)$ substitute $g_1,\ldots,g_m$ in place of $xg_1,\ldots,xg_m$ throughout the schema $g(\overline{xg})$, select the line whose predicate evaluates* to true, if this is p_{g_i} then t_{g_i} replaces $g(g_1,\ldots,g_m)$ in t_i to produce t_{i+1}. This process is continued until a t_i is reached that contains only elements from $B \cup D$. The fixed interpretation of the base functions means that this defines a unique element/

---

* The evaluation of the predicate trees is identical to the above process. The $p \in P$ have fixed interpretations.

element in D which is the result of the computation. If there is no such t_i the computation is said not to terminate.

3.1.2.2 Target Language

The target language into which we attempt to translate the recursion equation sets is a simple flow chart language. The main feature is that we do not allow an unrestricted flow chart shape, but only those flow charts that can be represented by while statements. We do this tc make our tasks of further manipulating the program easier. The results of Ashcroft and Manna (1971) mean that we do not lose any chance of translatability by so doing.

We have the same sets B of basic functions and P of predicate functions and a (possibly) augmented set X of variables.

3.1.2.2.1 Target Language Syntax

A program is a sequence of statements.

A statement is either an assignment statement
or an if then else statement
or a while statement.

An assignment statement has a left hand side and a variable.

A left hand side is a function tree i.e. a tree with members of X or members of B of rank 0 at its tip and members of B at its nodes.

A/

A while statement has a predicate tree i.e. a tree with members of X or members of B of rank 0 at its tip and members of B or of P at its nodes and a member of P at its root, and a program.

An if then else statement has a predicate tree, a true branch and a false branch.

A true branch is a program.

A false branch is a program.

For example a program to calculate factorial of n.

```
1->ans;
1->x2;
while not(x2>n) do [mult(x2,ans)->ans; x2+1->x2]
```

Throughout this thesis programs in the target language will represent functions and will therefore be single valued. For convenience the result of these programs will be stored in a variable called either res or ans. We do not make this part of the syntax of the language though.

We do not give a concrete syntax for our language. Whenever the abstract structure of a program is not clear from the written text we will insert extra separators or **delimitors** such as [ ] to make it clear.

3.1.2.2.2 Target Semantics

We do not give a detailed semantics for our target language programs deeming them to be sufficiently obvious.

3.1.3 Translation Process

The input to the process is a recursion equation set.

3.1.3.1 Pre-processing

Our first task is to extract all maximal mutual recursive sets from these recursion equations. A recursion equation set E is said to be a mutual recursion set if for each pair of function letters (f,g) occurring in E there is a finite sequence of function letters $f = g_0, g_1, \ldots, g_i, \ldots, g_n = g$ such that for i between 0 and n-1, g_i appears in the tree of a rule of g_{i+1}.

The problem of extracting maximal mutual recursive sets is identically that of extracting maximal circuits from the di-graph derived from the schemas by drawing up the calling graph. As there exist many algorithms to do this in the literature we shall not discuss it further, Berge and Ghouila-Houri (1965).

Maximal mutual recursion sets correspond to sets of equations closed under the possibility of recursive re-entry, and function letters (members of F) not being names of equations in a mutual recursion set can be regarded as basic functions (members of B) as far as the translation of this mutual recursive set is concerned. Thus we can translate each maximal mutual recursion set independently, and recombine/

recombine after the translation. The recombination is trivial as long as we take care to standardise apart variables used in the translation.

For example the schema $\mathcal{E}\ (f,\overline{xf})$

$$f(\overline{xf}) = \begin{cases} \ldots\ g(\overline{xf}) \\ \ldots\ f(\overline{xf}) \end{cases}$$

$$g(\overline{xg}) = \begin{cases} \ldots\ g(\overline{xg}) \\ \ldots\ h(\overline{xg}) \end{cases}$$

$$h(\overline{xh}) = \begin{cases} \ldots\ h(\overline{xh}) \\ \ldots\ i(\overline{xh}) \end{cases}$$

$$i(\overline{xi}) = \{\ \ldots\ g(\overline{xi})$$

where no other $f \in \mathbf{F}$ occurs in any tree of any rule of f, g, h or i, has the following associated graph

and the maximal mutual recursive sets are two, viz.

$\{f\}$ and $\{g,h,i\}$

For recursion equations that are not part of any mutual recursion set the translation is immediate, viz.

f/

$f(\overline{xf}) = \begin{cases} p_1 \longrightarrow t_1 \\ \vdots \quad\quad \vdots \\ p_m \longrightarrow t_m \end{cases}$ where $t_1,\cdots,t_m$ are trees not containing any $f \in F$.

becomes if p_1 then t_1->xf'

else if p_2 then t_2->xf

⋮

else if p_m then t_m->xf

Having extracted the mutual recursive sets the remaining part of the translation can be divided into two parts, 1) recognising the structure of the input set and 2) using semantic information about the primitives to try and affect a translation.

3.1.3.2 Structure Recognising

This is achieved by a matching algorithm that matches the input set against pattern sets embodying particular recursive forms. For example the factorial function

$$fact(x) = \begin{cases} not(zero(x)) \longrightarrow mult(x,fact(minus(x,1))) \\ zero(x) \longrightarrow 1 \end{cases}$$

has recursive form typified by

$$f(xf) = \begin{cases} a \longrightarrow b \\ c \longrightarrow h(d,f(e)) \end{cases}$$

as has the function to reverse a list.

Rev/

$$Rev(xl) = \begin{cases} null(xl) \longrightarrow nil \\ not(null(xl)) \longrightarrow append(Rev(tl(xl)),hd(xl)) \end{cases}$$

Our matching algorithm has the ability to perform functional abstraction and we are able to characterise a wide variety of equations by one structural pattern.

3.1.3.2.1 Matching Algorithm

The following abstract structure is assumed.

A circuit is a set of recursion equations.

A recursion equation has a left hand side and a right hand side.

A left hand side has a name and list of variable names.

A right hand side is a set of rules.

A rule has a predicate tree and a tree.

A function findmatchingcircuit: circuit $\times 2^{circuit}$ $\times$m-list->m-list is provided. Findmatchingcircuit(actualcircuit,posscircuitset,nil) searches through the set of possible circuit patterns posscircuitset and if it finds a circuit pattern that matches with the input circuit, actualcircuit, it succeeds producing a matchlist of elements of the chosen pattern with fragments of the input circuit.

The/

The full algorithm to achieve this is given in appendix I. The correctness criterion of this function is fairly simple to specify. However if one wanted to prove this function it is debatable whether this is what one would wish to prove as an important point of the algorithm is the dynamic way the backtracking organises the search for a successful match and we feel that any proof should take account of this.

The correctness criterion defines a successful conclusion to the search in terms of two functions match and matchtree which are defined further in appendix I. There they are functions that either succeed or fail, here they are regarded as predicates.

The conditions required for an element of the pattern to match with a fragment of a function are given fully in appendix I. Basically there are three types of elements in the pattern

(i) Atomic nodes. These can match with any tree as long as the tree does not contain any potentially recursive calls.

(ii) Recursive nodes. These must match directly with recursive nodes (i.e. names of equations) in the tree.

(iii) Compound nodes. These can match with functions abstracted from the tree having the same number of arguments and which do not contain recursive nodes.

$\exists$ matchlist/

$\exists$ matchlist $\exists$ circuit(circuit $\in$ possircuitset

$\wedge \forall$ schema(schema$\in$circuit

$\supset \exists$ schema1(schema1$\in$actualcircuit

$\wedge$ matchschema(schema,schema1,matchlist)

$\wedge \forall$ sch1,sch2((sch1$\in$actualcircuit $\wedge$ sch2$\in$actualcircuit

$\wedge$ matchschema(schema,sch1,matchlist)

$\wedge$ matchschema(schema,sch2,matchlist))

$\supset$ sch1 = sch2))))

where matchschema(schema,schema1,matchlist) is

match(nameof(schema),nameof(schema1),matchlist)

$\wedge$ match(varsof(schema),varsof(schema1),matchlist)

$\wedge \forall$ rule(rule$\in$schema

$\supset \exists$ rule1(rule1$\in$schema1

$\wedge$ matchtree(predicatetreeof(rule)

,predicatetreeof(rule1)

,matchlist)

$\wedge$ matchtree(treeof(rule),treeof(rule1),matchlist))

$\wedge \forall$ r1, r2((r1$\in$schema1 $\wedge$ r2$\in$schema1

$\wedge$ matchtree(predicatetreeof(rule),

predicatetreeof(r1),matchlist)

$\wedge$ matchtree(predicatetreeof(rule),

predicatetreeof(r2),matchlist))

$\supset$ r1 = r2))

3.1.3.3/

3.1.3.3 Production of Translation

The result of the first stage of the process is either a fail indicating that the input does not correspond to any structure within the present repertoire or a successful match indicating that the input has a structure that we can investigate further, and a matching of variables of the pattern to expressions and primitives within the function.

In the latter case findmatchingcircuit produces a list associating elements in the successful pattern with fragments of the input.

Associated with each structural form are procedures that look for special properties of the function and its primitives to try and find a translation. This could of course be an arbitrary piece of program text, and in the limit we could have a preliminary matching process that always succeeds and a complete algorithm i.e. Strong (1970) in this section. As a first order approximation we have implemented a simple system where only the properties of the primitives comprising the input function are considered. Each pattern has associated with it a piece of program which is run if the match succeeds with this pattern. Each such program consists of a series of tests and patterns.

3.1.3.3.1 Syntax of Tests

A program is a list of statements.

A/

A statement is <test program> translation is <program pattern>.

A test program is a list of tests.

A test is either a transformation test

or an oracle

or a function.

A program pattern is a target language program as defined earlier (3.1.2.2.1), except that the function trees may have as tips or nodes letters from the associated circuit pattern.

We use the word oracle for a device shown to us by R.M. Burstall by which the system interrogates the user to fill in gaps in its knowledge. At present the system has no ability to discover whether a piece of program text matched with a pattern element has an inverse or to construct such an inverse. This ability is often needed in producing translations. So we use an oracle to type out the text and ask the user if he can produce an inverse. If the user says no this test fails, otherwise if all the other tests succeed we ask the user to type in the inverse and incorporate this into the translation.

A transformation test is used to discover whether input functions have certain semantic properties needed to achieve a translation.

A transformation test is lefthandside==>righthandside.

Lefthandside and righthandside are both trees composed of elements from the associated pattern. The program substitutes in these/

these trees the values associated with the elements in the matchlist and then uses the algebraic manipulation routine (3.3) to see if it can transform the lefthandside to the righthandside using the rules giving the semantics of the primitive functions. If it can do so the test succeeds. If the effort bound is exceeded without a transformation being achieved the test fails.

A function can be any arbitrary function that either succeeds or fails. The basic one is freeof(x1,x2) which succeeds if the expression associated with x2 in matchlist does not contain as a subexpression the expression associated with x1 in matchlist. Fails otherwise.

The whole test program is run by taking each statement in turn and running each of the tests, if one test fails the next statement is tried. If all tests succeed the corresponding program pattern is taken. Occurrences of schema pattern letters are replaced by their values from matchlist and the expanded pattern is the translation. If all statements fail to produce a translation the fail is propogated back and either another match produced with the pattern or if there are no further matches the attempted match with this pattern fails and the fail propogates further back and the next pattern is attempted. If there are no further patterns the whole translation fails.

The derivation of the translation patterns was done in a manner similar to that used by Cooper (1966). This concentrates on the operational/

operational aspect of the recursion equation by writing down a computation sequence and seeing if any properties of the primitives would allow us to transform this computation sequence to one that corresponds to an iterative program. Again following Cooper we could use this method to show that our transformations are correct. We also suggest an alternative method. Given a recursion equation schema R1 and an iterative schema I1 which we wish to prove equivalent we first translate I1 into a recursion equation R2 by any of the well-known methods and then attempt to prove R1 and R2 equivalent using only the properties of the primitives asked for in the translation test.

3.1.3.4 Examples

We now give a list of patterns and translations with a list of some of the examples that they have translated.

A basic pattern is

SCHEMA/

1. SCHEMA 1

pattern

$$f(x) = \begin{cases} a \longrightarrow b \\ c \longrightarrow h(d, f(e)) \end{cases}$$

Translation program

```
<freeof(x,h); h(x1,h(x2,x3)) ==> h(h(x1,x2),x3)>
      translation is <if a then b->ans
                      else[d->ans; e->x;
                      while c do [h(ans,d)->ans; e->x]
                      h(ans,b)->ans]>
<freeof(x,h); h(x1,h(x2,x3)) ==> h(x2,h(x1,x3))>
      translation is < b->ans; while c do
                      [h(d,ans)->ans; e->x]>
<freeof(x,h); hasinverse(e); hasuniquepoint(a)>
      translation is < b->ans; x->xsave;uniquepointof(a)->x;
            while not(x=xsave) do [inverseof(e)(x)->x;
                                   h(d,ans)->ans]>
```

hasinverse(x) is an oracle which asks if the text associated with x has an inverse

hasuniquepoint(x) is an oracle which asks if there is a unique value that satisfies the predicate associated with x.

inverseof and uniquepointof are oracles which ask the user to type these values in.

On inputting the factorial function

fact/

$$fact(n) = \begin{cases} zero(n) \longrightarrow 1 \\ not(zero(n)) \longrightarrow mult(n,fact(minus(n,1))) \end{cases}$$

the program succeeds in matching it with the above pattern in just one way viz.

a = zero(n); b = 1; c = not(zero(n)); d = n;

h = λx1,x2; mult(x1,x2); e = minus(n,1)

f = fact; x = n.

Running this match on the first test succeeds. Freeof(x,h) succeeds because λ x1,x2; mult(x1,x2) does not have n as a free variable. On substituting into the transformation we get mult(x1,mult(x2,x3)) ==> mult(mult(x1,x2),x3). This transformation is then presented to the expression transformation routine (see 3.3) which succeeds in making the transformation. This fact is given as a basic theorem for the mult function.

Thus one translation produced is

```
if zero(n) then 1->ans
    else [n->ans; minus(n,1)->n; while not(zero(n)) do [mult(ans,n)->ans;
                                                        minus(n,1)->n]
                                 mult(ans,1)->ans]
```

The second test program also succeeds although in this case the transformation requires some deduction. We have to prove mult(x1,mult(x2,x3)) => mult(x2,mult(x1,x3)).

Given/

Given the rules

1 mult(x,y) = mult(y,x)

and 2 mult(x,mult(y,z)) = mult((x,y),z)

the algebraic manipulation routine goes through the following deduction

mult(x1,mult(x2,x3)) Starting Expression

=> mult(mult(x1,x2),x3) by 2

=> mult(mult(x2,x1),x3) by 1

=> mult(x2,mult(x1,x3)) by 2

Giving the translation

```
1->ans; while not(zero(n)) do [mult(n,ans)->ans; n-1->n]
```

The third test also succeeds the dialogue with the user going:

```
: Has minus(n,1) a unique inverse
: Yes

: Has zero(n) a unique point
: Yes

: Type in the inverse of minus(n,1) please
: Plus(n,1)

: Type in the unique point such that zero(n)
: 0
```

The program produced in this case is

1/

```
1->ans; n->xsave; 0->n;
   while not(n=xsave) do [plus(n,1)->n; mult(n,ans)->ans]
```

A more complicated example, a function for reversing lists at the top level.

$$\text{Rev(xl)} = \begin{cases} \text{not(null(xl))} \longrightarrow \text{concat(Rev(tl(xl)), cons(hd(xl),nil))} \\ \text{null(xl)} \longrightarrow \text{nil} \end{cases}$$

matches with the factorial pattern in the following ways

A.
```
a = null(xl)
b = nil
c = not(null(xl))
e = tl(xl)
f = Rev
x = xl
h = λ x1,x2; concat(x2,x1)
d = cons(hd(xl),nil)
```

B. a, b, c, e, f, x — As in **A.**
```
h = λ x1,x2; concat(x2,cons(x1,nil))
d = hd(xl)
```

C. a, b, c, e, f, x — As in **A.**
```
h = λ x1,x2; concat(x2,cons(hd(xl),x1))
d = nil
```

D. a, b, c, e, f, x — As in **A.**
```
h = λ x1,x2; concat(x2,cons(hd(x1),nil)
d = xl
```

The/

The program succeeds in just one case in making a translation. Taking match A together with the first condition requires the transformation

concat(concat(x3,x2),x1) => concat(x3,concat(x2,x1)) which

succeeds being again a basic property.

Thus the translation produced is

```
if null(x1) then nil->ans
     else [cons(hd(x1),nil)->ans; tl(x1)->x1;
           while not(null(x1))
              do [concat(cons(hd(x1),nil),ans)->ans;
                  tl(x1)->x1]
           concat(nil,ans)->ans]
```

All the other matches fail on all tests.

2. SCHEMA 2

$$f(x1,x2) = \begin{cases} a \longrightarrow b \\ c \longrightarrow h(d,f(e1,e2)) \end{cases}$$

Translation program

```
<freeof(x1,h); freeof(x2,h); h(x1,h(x2,x3)) ==> h(x2,h(x1,x3))>
        translation is < b->ans; while c do [h(d,ans)->ans; e1->x1save;
                                             e2->x2; x1save->x1]>
```

3./

3. SCHEMA 3

$$f(x) = \begin{cases} a \longrightarrow b \\ c \longrightarrow h(f(d1),f(d2)) \end{cases}$$

Translation program

<freeof(x,h); d1 ==> d2; hasinverse(d1); hasuniquepoint(a)> *

translation is < b->ans; x->xsave; uniquepointof(a)->x;

while not(x=xsave) do [inverseof(d1)(x)->x; h(ans,ans)->ans]>

<freeof(x,h); d1 ==> d2{d2/x}†; h(x1,h(x2,x3)) ==> h(x2,h(x1,x3))>

translation is < b->y1; b->y2; while c do [h(y1,y2)->ans;

y2->y1; ans->y2;d1->x]>

<freeof(x,h); d1 ==> d2; h(x1,h(x2,x3)) ==> h(x2,h(x1,x3))>

translation is < b->ans; while c do [h(ans,ans)->ans; d1->x]>

4./

---

* d1 ==> d2 means that whatever matches with d1 is transformable to whatever matches with d2. We need not state the other case i.e. d2 ==> d1 as this will be catered for by the matching algorithm. Similarly with the second condition.

† By e1{e2/e3} we mean substitute e3 for e2 throughout e1.

4. Iterative schemas viz.

SCHEMA 4

$$f(x) = \begin{cases} a \longrightarrow b \\ c \longrightarrow h(f(d)) \end{cases}$$

Translation program

<freeof(x,h); hasinverse(d); hasuniquepoint(a)>

translation is < b->ans; x->xsave;

uniquepointof(a)->x;

while not(x=xsave)

do [inverseof(d)->x; h(ans)->ans]>

<h ==> λx;x>

translation is < while c do [d->x] b->ans>

SCHEMA 5

$$f(x,y) = \begin{cases} a \longrightarrow b \\ c \longrightarrow h(f(d1,d2)) \end{cases}$$

Translation program

<h ==> λx;x>

translation is < while c do [d1->xsave; d2->y; xsave->x]

b->ans>

There/

There are of course further translations if one allows oneself counter augmented flowcharts, Strong (1970). We do not introduce this complication here.

These are sufficient to achieve the following translations

1. $$Fib(n) = \begin{cases} or(equal(n,1),equal(n,0)) \longrightarrow 1 \\ not(or(equal(n,1),equal(n,0))) \longrightarrow \\ \qquad plus(Fib(minus(n,1)),Fib(minus(n,2))) \end{cases}$$

```
1->y1; 1->y2; while not(or(equal(n,1),equal(n,2)))
                    do [plus(y1,y2)->ans; y2->y1; ans->y2; minus(n,1)->n]
```

By schema 3 test 2

2. A function defining union in terms of consset

$$union(x,y) = \begin{cases} nullset(x) \longrightarrow y \\ not(nullset(x)) \longrightarrow consset(choose(x),union(minus(choose(x),x),y)) \end{cases}$$

```
y->ans;
while not(nullset(x)) do [consset(choose(x),ans)->ans;
                          minus(choose(x),x)->x]
```

By schema 2

3. A function to compute the Boolean (power set) of a set is

Bool/

$$\text{Bool}(S) = \begin{cases} \text{nullset}(S) \longrightarrow \text{nilset} \\ \text{not}(\text{nullset}(S)) \longrightarrow \text{union}(\text{union}(\text{bool}(S/\{\text{choose}(S)\}), \\ \qquad \{\{\text{choose}(S)\}\}), g(\text{bool}(S/\{\text{choose}(S)\}), \\ \qquad\qquad \text{choose}(S))) \end{cases}$$

$$g(S,x) = \begin{cases} \text{nullset}(S) \longrightarrow \text{nilset} \\ \text{not}(\text{nullset}(S)) \longrightarrow \text{consset}(\text{consset}(x,\{\text{choose}(S)\}), g(S/\{\text{choose}(S)\},x)) \end{cases}$$

This divides first into two separate tasks corresponding to the two separate equations. The first equation matches with schema 3 giving the following match

a = nullset(S) b = nilset

c = not(nullset(S)) f = bool x = S

d1 = S/{choose(S)} d2 = S/{choose(S)}

h = λ x1,x2; union(union(x1,{{choose(S)}},g(x2,choose(S))),g(x2,choose(S)))

All tests fail.

The second equation matches with schema 2, and the test succeeds h being matched with λ x1,x2; consset(x1,x2)

and consset(x1,consset(x2,x3)) ==> consset(x2,consset(x1,x3)) is provable. This gives the translation

```
nilset->ans;
  while not(nullset(S))
     do [consset(consset(x,{choose(S)}),ans)->ans;
         S/{choose(S)}->x1save; x->x; x1save->S]
```

4./

4. A program to compute 2^n

$$\text{twon}(n) = \begin{cases} \text{zero}(n) \longrightarrow 1 \\ \text{not}(\text{zero}(n)) \longrightarrow \text{mult}(2, \text{twon}(\text{minus}(n,1))) \end{cases}$$

matches with schema 1 and translates in 3 ways

(i)
```
if zero(n) then 1->ans;
        else [2->ans; minus(n,1)->n;
              while not(zero(n))
                 do [mult(ans,2)->ans; minus(n,1)->n]
              mult(ans,1)->ans]
```

(ii)
```
1->ans; while not(zero(n)) do [mult(2,ans)->ans; minus(n,1)->n]
```

(iii)
```
1->ans; n->xsave; 0->n;
        while not(n=xsave) do [plus(n,1)->n; mult(2,ans)->ans]
```

where plus(n,1) is the user supplied inverse to minus(n,1) and 0 is the unique n such that zero(n).

5. A function that takes a finite partial function represented as a set of pairs and applies it to all members of a set

$$\text{apply}(f,s) = \begin{cases} \text{nullset}(f) \longrightarrow \text{nilset} \\ \text{not}(\text{nullset}(f)) \longrightarrow \text{union}(g(\text{choose}(f),s), \text{apply}(f/\{\text{choose}(f)\},s)) \end{cases}$$

$$g(p,s) = \begin{cases} \text{nullset}(s) \longrightarrow \text{nilset} \\ \text{and}(\text{not}(\text{nullset}(s)), \text{equal}(\text{choose}(s), \text{front}(p))) \longrightarrow \{\text{back}(p)\} \\ \text{and}(\text{not}(\text{nullset}(s)), \text{not}(\text{equal}(\text{choose}(s), \text{front}(p)))) \longrightarrow g(p, s/\{\text{choose}(s)\}) \end{cases}$$

This/

This again splits into two problems.

The first equation (treating g as basic) translates to

```
nilset->ans;
   while not(nullset(f))
      do [union(g(choose(f),s),ans)->ans;
          f/{choose(f)}->x1save; s->s; x1save->f]
```

By schema 2.

The second equation illustrates a point about generality. At the present level of simplicity our implementation would fail to translate this, but it would be easy to add a preprocessor which effectively generalises rules that contain no recursive call into one rule i.e. in this case a function of the form

$$f(x,y) = \begin{cases} a \longrightarrow b \\ c \longrightarrow d \\ e \longrightarrow h(f(g1,g2)) \end{cases}$$

would be preprocessed into the form

$$f(x,y) = \begin{cases} a \vee c \longrightarrow \text{if } a \text{ then } b \text{ else } d \\ e \longrightarrow h(f(g1,g2)) \end{cases}$$

which is then translatable giving

```
while and(not(nullset(s)),not(equal(choose(s),front(p))))
   do [p->xsave; s/{choose(s)}->s; xsave->p]
[if nullset(s) then nilset else {back(p)}]
                              ->ans
```

By schema 5.

3.2 Compilation.

We are now in a position to describe in some detail our implementation of the compilation process outlined in chapter 2. To recapitulate our overall task is the implementation of recursion equations written in the language S-0 as iterative programs in B-0 or L-1. The first stage S-0=>S-1 is achieved by the recursion translation process previously described. This process is not restricted to any particular set of function primitives, the semantics of the primitives being given as transformation rules. The basic process involved in the transformations S-1=>L-0, S-1=>B-0 and L-0=>L-1 could be applied to any compilation and we have attempted to separate the subject dependent information from the main processes in order to make it easier to specialise the compiler to other domains. To make our discussions more concrete we will describe them in terms of the specific domains of S-1, B-0, L-0 and L-1 and indicate later how the differing aspects are separated [chapter 5].

Towards this end we will have to describe the syntax and semantics of the languages S-1, B-0, L-0 and L-1.

3.2.1 Description of S-1, B-0, L-0 and L-1

3.2.1.1 S-1 Syntax

A program is a sequence of statements.

A statement is either an assignment statement e.g. x->y

or an ifthenelse statement e.g. if P then A1 else A2

or a while statement e.g. while P do [A]

An/

An assignment statement has a left hand side and a variable.

A left hand side is a function tree which has members of X or members of B of rank 0 at its tips and members of B at its nodes.

A while statement has a predicate tree and a program.

An ifthenelse statement has a predicate tree, a true branch and a false branch.

A predicate tree is a tree with members of X or members of B of rank 0 at its tips, members of B or of P at its nodes and a member of P at its root.

A true branch is a program.

A false branch is a program.

$B = B_0 \cup B_1 \cup B_2$

where B_0 = {nilset} $\cup$ names for integers

B_1 = {choose}

B_2 = {union,subtract,consset,minus}

$P = P_1 \cup P_2$

where P_1 = {not,nullset}

P_2 = {and,or,equal}

3.2.1.2 S-1 Semantics

We describe as much semantic apparatus as is utilised in our implementation. We define the semantics by giving a mathematical meaning to the state of a computation of a program and showing how certain constructs modify the state. Our meta-language is set-theoretic.

The state of a computation of a program P written in S-1 is given by a pair

$$S = \langle V, C \rangle$$

where V = {Variables occurring in P}

and $C = V \times \bigcup_{n=0}^{K} \mathbb{B}^n(I)$

and I = the integers.

We write $\mathbb{B}(X)$ to denote the power set of X, i.e. $\mathbb{B}(X) \equiv 2^X$, and $\mathbb{B}^n(X)$ to denote the sets of sets of X to depth n.

i.e.

$$\mathbb{B}^0(I) = I$$

$$\mathbb{B}^1(I) = \mathbb{B}(I)$$

and $$\mathbb{B}^{n+1}(I) = \mathbb{B}(\mathbb{B}^n(I))$$

We shall give an outline of a recursive definition of a function value: expressions in S-1 $\rightarrow \bigcup_{n=0}^{K} \mathbb{B}^n(I)$ which allows the semantics of an expression to be evaluated.

In state $S^0 = \langle V, C^0 \rangle$ value($\mathcal{E}$) is given by

cases/

cases $\mathcal{E}$: $\mathcal{E} \in V$: $c^0(\mathcal{E})$

nilset: $\emptyset$

consset($\mathcal{E}_1, \mathcal{E}_2$): {value($\mathcal{E}_1$)} $\cup$ value($\mathcal{E}_2$)

union($\mathcal{E}_1, \mathcal{E}_2$): value($\mathcal{E}_1$) $\cup$ value($\mathcal{E}_2$)

subtract($\mathcal{E}_1, \mathcal{E}_2$): value($\mathcal{E}_1$)/value($\mathcal{E}_2$)

minus($\mathcal{E}_1, \mathcal{E}_2$): value($\mathcal{E}_2$)/{value($\mathcal{E}_1$)}

choose($\mathcal{E}_1$): x for some $x \in$ value($\mathcal{E}_1$)

and $\mathcal{E}_1 = \mathcal{E}_2 \Rightarrow$ choose($\mathcal{E}_1$) = choose($\mathcal{E}_2$)

$\mathcal{E}_1 \rightarrow \mathcal{E}_2$: then causes state transition $S^0 \Rightarrow S^1$ such that

$S^1 = \langle V, C^1 \rangle$

where $C^1 = C^0 \cup \{\langle \mathcal{E}_2, \text{value}(\mathcal{E}_1)\rangle\}/\{\langle \mathcal{E}_2, \text{value}(\mathcal{E}_2)\rangle\}$

3.2.1.3 L-0 Syntax

As for S-1 except that

$B = B_0 \cup B_1 \cup B_2$

where B_0 = {nil} $\cup$ names for integers

B_1 = {hd,tl}

B_2 = {cons}

Applist(x,f) stands for while not(null(x)) do [f(hd(x)); tl(x)->x]

$P = P_1 \cup P_2$

where P_1 = {null,not}

P_2 = {equal,and,or}

3.2.1.4/

3.2.1.4 L-1 Syntax

As for L-0 except that

An assignment statement has a right hand side and a left hand side. A left hand side is as in L-0.

A right hand side is a tree whose tips are variables and whose nodes are hd or tl, e.g. X->hd(Y) is allowed.

3.2.1.5 L-0 and L-1 Semantics

The state of the computation of a program P written in L-0 or L-1 is given by a 4-tuple.

$S = \langle V, A, L, C \rangle$

where V = {variable symbols in P}

A = {unique names for store locations}

$L = V \times (A \cup I)$ I = integers

$C = A \times ((A \cup \text{'nil'} \cup I) \times (A \cup \text{'nil'} \cup I))$

The explicit mention of store locations is not strictly necessary for L-0 but it is for L-1, where destructive assignment is allowed. We use the same semantic apparatus for both languages as this facilitates the translation of programs written in L-0 into programs in L-1. Store locations are thought of as being taken off the free list when required. The semantics of the freelist are mirrored by a function nextname: $\mathcal{B}$ (A)->A which when given all the store locations used so far produce a name for the next store location.

We/

We again give an outline of a function value

value: expressions->V $\cup$ A $\cup$ I $\cup$ 'nil' which will allow the semantics of any list expression to be calculated.

value($\mathcal{E}$) in state $S^0 = \langle V, A^0, L^0, C^0 \rangle$

Cases $\mathcal{E}$:

$\mathcal{E} \in V$: $L^0(\mathcal{E})$

hd($\mathcal{E}_1$): $(C^0(\text{value}(\mathcal{E}_1)))_1$

tl($\mathcal{E}_1$): $(C^0(\text{value}(\mathcal{E}_1)))_2$

cons($\mathcal{E}_1$, $\mathcal{E}_2$): nextname(A^i), where $S^i = \langle V, A^i, L^i, C^i \rangle$ is the state obtaining after the evaluation of value($\mathcal{E}_1$), value($\mathcal{E}_2$).

Causes state transition $S^i \to S^{i+1}$, $S^{i+1} = \langle V, A^{i+1}, L^i, C^{i+1} \rangle$

$A^{i+1} = A^i \cup \{\text{nextname}(A^i)\}$

$C^{i+1} = C^i \cup \{\langle \text{nextname}(A^i), \langle \text{value}(\mathcal{E}_1), \text{value}(\mathcal{E}_2) \rangle \rangle\}$

$\mathcal{E}_1 \to \mathcal{E}_2$: Causes state transitions $S^i \to S^{i+1}$ where S^i is the state obtaining after the evaluation of $\mathcal{E}_1$ and $\mathcal{E}_2$.

cases

(i) $\mathcal{E}_2 \in V$

$A^{i+1} = A^i$

$L^{i+1} = L^i \cup \{\langle \mathcal{E}_2, \text{value}(\mathcal{E}_1) \rangle\} / \{\langle \mathcal{E}_2, (\text{value}(\mathcal{E}_2))_2 \rangle\}$

$C^{i+1} = C^i$

The/

The next two cases can only occur if the program is written in L-1

(ii) $\mathcal{E}_2$ is hd($\mathcal{E}_3$):

$$A^{i+1} = A^i$$

$$L^{i+1} = L^i$$

$$C^{i+1} = C^i \cup \{\langle value(\mathcal{E}_3), \langle value(\mathcal{E}_1), (C^i(value(\mathcal{E}_3)))_2 \rangle\rangle\} / \{\langle value(\mathcal{E}_3), C^i(value(\mathcal{E}_3))\rangle\}$$

(iii) $\mathcal{E}_2$ is tl($\mathcal{E}_3$):

$$A^{i+1} = A^i$$

$$L^{i+1} = L^i$$

$$C^{i+1} = C^i \cup \{\langle value(\mathcal{E}_3), \langle (C^i(value(\mathcal{E}_3)))_1, value(\mathcal{E}_1)\rangle\rangle\} / \{\langle value(\mathcal{E}_3), C^i(value(\mathcal{E}_3))\rangle\}$$

3.2.1.6 B-0 Syntax

As in L-1 except that

An assignment expression has a right hand side and a left hand side.

A left hand side as in L-0.

A right hand side is either a variable or a tree of depth 1 whose tips are variables and whose root is intsub.

$B = B_0 \cup B_1 \cup B_2$

where B_0 = names for integers

B_1 = {init,length}

B_2 = {intsub,plus,minus}

P/

$P = P_1 \cup P_2$

where P_1 = {not}

P_2 = {and,or,equal}

3.2.1.7 B-0 Semantics

$S = \langle V,I,C \rangle$

V = {variables}

$I = N^+$

$C = V \to (N^+ \to \{0,1\}) \cup I$

i.e. a bit string is viewed as a function (total on some subset of N^+) from N^+ to {0,1}.

Value($\mathcal{E}$) in $S^0 = \langle V,I,C^0 \rangle$

= cases $\mathcal{E}$:

$\mathcal{E} \in I$: $\mathcal{E}$

$\mathcal{E} \in V$: $C^0(\mathcal{E})$

init($\mathcal{E}_1$): (value($\mathcal{E}_1$)$\in N^+$)

λi; if $0 \leq i \leq$ value($\mathcal{E}_1$) then 0 else undef

intsub($\mathcal{E}_1, \mathcal{E}_2$): value($\mathcal{E}_2$)(value($\mathcal{E}_1$))

length($\mathcal{E}_1$): max{i | value($\mathcal{E}_1$)(i) $\neq$ undef}

$\mathcal{E}_1 \to \mathcal{E}_2$:

cases (i) $\mathcal{E}_2 \in V$

then causes transition $S^0 \to S^1$

such that

$S^1 = S^0$ except that $C^1(\mathcal{E}_2)$ = value($\mathcal{E}_1$)

(ii)/

(ii) $\mathcal{E}_2$ is intsub($\mathcal{E}_3$, $\mathcal{E}_4$)

causes transition $s^0 \Rightarrow s^1$

such that $s^1 = s^0$

except that value($\mathcal{E}_4$)(value($\mathcal{E}_3$)) = value($\mathcal{E}_1$)

3.2.2 Compilation S-1=>B-0 or L-0-Example

We will now describe the steps of our algorithm involved in the translation of an S-1 program into an L-0 or B-0 program. The process involved in the step L-0 to L-1 is similar but we have chosen to describe it separately in section 3.2.3 as it involves a more complicated semantic apparatus but does not involve the expansion of iterative operators.

The process takes as input whole programs, splits them into sections as defined in 2.4.2.3 and subjects each section to the process of

(i) Extracting the transformation

(ii) Projecting and optimising the transformation

(iii) Re-writing in the lower domain.

In order again to give our discussion some concreteness we will illustrate the steps involved by means of an example. In order to illustrate several points succinctly in one example we have constructed an artificial example.

We/

We will consider the **section** of program, with various textual points numbered.

```
    while or(nullset(union(s2,s3)),member(x,s2))
                do [1union(s3,s2)->s2;2subtract(s2,subtract(s3,s2))->s33]
    4
  consset(choose(s2),nilset)->s2;
    5
  union(s2,s3)->s3;
    6
```

3.2.2.1 Extracting the Transformation

(i) Initialising the state vector.

The program requires knowledge of the types of each variable present in the program. From this information it constructs an initial state vector, inventing a name for the symbolic value of each variable.

In the example s1, s2, s3 are of type set and x is of type individual, thus we get an initial vector.

s1	s2	s3	x
↓	↓	↓	↓
set1	set2	set3	ob1

Associated with each set name is a membership list which is initially empty.

(ii) Interpreting the program on the state vector.

The/

The program proceeds as a normal interpreter from instruction to instruction. For each assignment instruction it evaluates the left hand side and replaces this as the value of the variable on the right hand side producing an updated state vector. However in this case evaluating consists of forming the free expression in the functions over the set or object names. One special transformation is applied here in the case of implementations into L-O.

Any occurrence of choose($\mathcal{E}$) causes defon($\mathcal{E}$), see 3.2.2.2, to be calculated. If defon($\mathcal{E}$) is nonempty the first element of defon($\mathcal{E}$) is chosen and replaces choose($\mathcal{E}$). If defon($\mathcal{E}$) is empty then if $\mathcal{E}$ is a setname a new object is invented, added to this setnames object list and replaces choose($\mathcal{E}$), otherwise the expression is left as it is.

In the rest of the example we will assume that the implementation is to be in L-O and note where the implementation would differ if it were to be in B-O.

In our example, the state vector at the numbered points is thus

1 s1 s2 s3 x membership list set1 $\emptyset$

↓ ↓ ↓ ↓ set2 $\emptyset$

set1 set2 set3 ob1 set3 $\emptyset$

2/

2 s1 ↓ set1 s2 ↓ union(set3,set2) s3 ↓ set3 x ↓ ob1

3 s1 ↓ set1 s2 ↓ union(set3,set2) s3 ↓ subtract(union(set3,set2),subtract(set3,union (set3,set2))) x ↓ ob1

and at 4

s1 ↓ set1 s2 ↓ set2 s3 ↓ set3 x ↓ ob1

5 s1 ↓ set1 s2 ↓ consset(ob2,nilset) s3 ↓ set3 x ↓ ob1 and the membership list of set2 becomes [ob2]

6 s1 ↓ set1 s2 ↓ consset(ob2,nilset) s3 ↓ union(consset(ob2,nilset),set3) x ↓ ob1

Thus the two transformations extracted are

s1 ↓ set1 s2 ↓ set2 s3 ↓ set3 x ↓ ob1 membership list set1 ∅
set2 [ob2]
set3 ∅

to

s1 ↓ set1 s2 ↓ union(set3,set2) s3 ↓ subtract(union(set3,set2),subtract(set3,union(set3,set2)))

and

s1 ↓ set1 s2 ↓ set2 s3 ↓ set3 x ↓ ob1

to/

to

s1	s2	s3	x
↓	↓	↓	↓
set1	consset(ob2,nilset)	union(consset(ob2,nilset),set3)	ob1

The predicate expression of the while is also included in the transformations. In our example this is

or(nullset(union(set2,set3)), member(ob1,set2))

3.2.2.2 Simplification

For this and subsequent expositions the state description will be thought of as consisting of four elements

(i) the original state which we will call state0

(ii) the predicate expression of the while statement and

(iii)&(iv) the end states resulting from the execution of the while body and the rest of the section;. We will call these statet and statef respectively.

In this process the program goes through the state description and replaces each expression $\mathcal{E}$ by simplify($\mathcal{E}$) where simplify($\mathcal{E}$) is as below.

Simplify: expression->expression.

(i) For implementation into L-0

Consset/

Consset(x,S) can be reduced to S if we know that x is in S or to cons(x,S) if we know x is definitely not in S. Minus(x,S) can be reduced to S if we know x is definitely not in S. If x is definitely in S we attempt to order S to make x the first element so that minus(x,S) can be reduced to tl(S).

```
simplify(expr) = cases expr:
                  setname, objectname
                  or constant: expr
                 f(arg1,arg2):
                        cases f:
                        consset: if simplify(arg1)∈defon(simplify(arg2))
                                  then arg2
                                  elseif simplify(arg1)∈defnoton(simplify(arg2))
                                   then cons(simplify(arg1),simplify(arg2))
                                   else consset(simplify(arg1),simplify(arg2))
                          minus: if simplify(arg1)∈defon(simplify(arg2))
                                  then access(simplify(arg1),simplify(arg2))
                                  elseif simplify(arg1)∈defnoton(simplify(arg2))
                                   then simplify(arg2)
                                   else minus(simplify(arg1),simplify(arg2))
                           rest:  f(simplify(arg1),simplify(arg2))
                      f(arg1): f(simplify(arg1))
```

(ii) For implementation into B-O

Consset/

Consset(x,S) can be reduced to S if we know that x is in S.

Minus (x,S) can be reduced to S if we know that x is not in S.

```
simplify(expr) =
  cases expr
 f(ℰ1, ℰ2):
      cases f:
          consset: if simplify(ℰ1)∈defon(simplify(ℰ2))
                       then simplify(ℰ2)
                       else consset(simplify(ℰ1),simplify(ℰ2))
          minus  : if simplify(ℰ1)∈defnoton(simplify(ℰ2))
                       then simplify(ℰ2)
                       else minus(simplify(ℰ1),simplify(ℰ2))
          rest   : f(simplify(ℰ1), simplify(ℰ2))
      f(ℰ1): f(simplify(ℰ1))
      rest : expr
```

where access(arg1,arg2) produces the code to enable arg1 to be accessed in the list arg2

where defon(ℰ) =

```
          cases ℰ :
                  setname: membership list of ℰ
                  nilset : ∅
          union(ℰ1, ℰ2)    : defon(ℰ1) ∪ defon(ℰ2)
          subtract(ℰ1, ℰ2): defon(ℰ1) ∧ defnoton(ℰ2)
          consset(ℰ1, ℰ2) : {ℰ1} ∪ defon(ℰ2)
          minus(ℰ1, ℰ2)    : defon(ℰ2)/{ℰ1}
```

where/

where defnoton($\mathcal{E}$) = cases $\mathcal{E}$:

setname : $\emptyset$

nilset : $\emptyset$

union($\mathcal{E}_1, \mathcal{E}_2$) : defnoton($\mathcal{E}_1$) $\wedge$ defnoton($\mathcal{E}_2$)

subtract($\mathcal{E}_1, \mathcal{E}_2$): (defnoton($\mathcal{E}_1$) $\vee$ defon($\mathcal{E}_2$))

$\vee$ (defon($\mathcal{E}_1$) $\wedge$ defnoton($\mathcal{E}_2$))

consset($\mathcal{E}_1, \mathcal{E}_2$) : defnoton($\mathcal{E}_2$)/{ $\mathcal{E}_1$}

minus($\mathcal{E}_1, \mathcal{E}_2$) : { $\mathcal{E}_1$} $\vee$ defnoton($\mathcal{E}_2$)

3.2.2.3 Planning

The planning and projection (3.2.2.4) stages of the program are commutative. In the program projection actually occurs before planning, but it is conceptually easier to describe it the other way around.

In this the planning or optimising stage the transformations in the higher domain, in this case S-1, are rearranged using algebraic laws appropriate to this domain but with an aim that when these transformations are projected into the lower domain (B-O or L-O) they can be implemented as efficiently as possible. In this case the planning is done so that when the rearranged transformations are implemented expansion of the higher primitives in terms of lower primitives will present the maximum opportunities for optimising interactions. The main optimising interactions important here are the combination of common iterations and common subexpressions.

3.2.2.3.1 Production of Alternative Forms

The program goes through the state description and produces three lists of expressions which occur in the predicate expression, statet and statef. We call these the plist, the tlist and the flist respectively. It then uses the algebraic manipulation routine (3.3) and the rules giving the algebra of the primitives to form these into a list of lists of all alternative forms for these expressions. While doing this it forms two dictionaries.

1) The expression dictionary that associates a unique number with every expression occurring

and 2) The subexpression dictionary that associates with each expression number a list of numbers corresponding to subexpressions of the expression.

Thus in our example which has a state description of

state0 = s1 ↓ set1, s2 ↓ set2, s3 ↓ set3, x ↓ ob1

predicate expression = or(nullset(union(set2,set3)),member(ob1,set2))

statet = s1 ↓ set1, s2 ↓ union(set3,set2), s3 ↓ subtract(union(set3,set2),subtract (set3,union(set3,set2))), x ↓ ob1

statef = s1 ↓ set1, s2 ↓ consset(ob2,nilset), s3 ↓ union(consset(ob2,nilset),set3), x ↓ ob1

the/

the plist is

[or(nullset(union(set2,set3)),member(ob1,set2))]

the tlist is

[union(set3,set2),subtract(union(set2,set3),subtract(set3,union(set3,set2)))]

and the flist is

[consset(ob2,nilset),union(consset(ob2,nilset),set3)].

Given just the commutative property of union

i.e. union(x,y) = union(y,x), the program goes through the state description and produces alternative form lists

plist =

```
[[or(nullset(union(set2,set3)),member(ob1,set2)),
  or(nullset(union(set3,set2)),member(ob1,set2))]]
```

tlist =

```
[[union(set3,set2),union(set2,set3)]
 [subtract(union(set2,set3),subtract(set3,union(set3,set2))),
  subtract(union(set3,set2),subtract(set3,union(set3,set2))),
  subtract(union(set2,set3),subtract(set3,union(set2,set3))),
  subtract(union(set3,set2),subtract(set3,union(set2,set3)))]]
```

flist/

flist =

[[consset(ob2,nilset)],
[union(consset(ob2,nilset),set3),union(set3,consset(ob2,nilset))]]

The expression dictionary that is formed during this second pass is

No	Expression
1	or(nullset(union(set2,set3)),member(ob1,set2))
2	nullset(union(set2,set3))
3	union(set2,set3)
4	member(ob1,set2)
5	or(nullset(union(set3,set2)),member(ob1,set2))
6	nullset(union(set3,set2))
7	union(set3,set2)
8	subtract(union(set2,set3),subtract(set3,union(set3,set2)))
9	subtract(union(set3,set2),subtract(set3,union(set3,set2)))
10	subtract(union(set2,set3),subtract(set3,union(set2,set3)))
11	subtract(union(set3,set2),subtract(set3,union(set2,set3)))
12	subtract(set3,union(set3,set2))
13	subtract(set3,union(set2,set3))
14	consset(ob2,nilset)
15	union(consset(ob2,nilset),set3)
16	union(set3,consset(ob2,nilset))

And the subexpression dictionary is

	Expression number															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	2	3	4	4	4	7	3	7	3	3	7	3	14	14	14
subexpressions	2	3			5	6		7	9	10	7	12	13		15	16
	3				6			8	12	13	11					
	4				7			12			13					

Thus after the second pass through the state description the plist, tlist and flist are coded as

[[1 5]]

[[7 3][8 9 10 11]]

and [[14][15 16]] respectively.

Each element of these lists is a list giving all alternative forms for an expression occurring in that component of the state description.

3.2.2.3.2 Evaluation of Alternatives

From each list the program now forms a new plist, tlist and flist. Each is a list of n-tuples. Each n-tuple is formed by picking one element from each n-tuple of the old list. Thus our new plist, tlist and flist become

plist [[1][5]]

tlist [[7 8][7 9][7 10][7 11][3 8][3 9][3 10][3 11]]

flist [[14 15][14 16]]

Each/

Each n-tuple of a list represents one form for all the expressions involved in the corresponding state. A choice of one n-tuple from each list represents one form for all expressions involved in the state descriptions. The program now evaluates all such choices in order to try and determine the optimal form to implement.

The program has two measures that it uses in forming this evaluation.

1) The total number of distinct subexpressions that it will need to calculate

and 2) The number of expressions that can have their iterations performed together. For example in L-0 the definitions of union and subtract are

```
union(s1,s2) = s2->res; applist(s1,lambda x; if not(member(x,s2)) then
                                                cons(x,res)->res)
subtract(s1,s2) = nil->res; applist(s1,lambda x; if not(member(x,s2) then
                                                cons(x,res)->res)
```

and if therefore two expressions with outermost functions union and subtract have an equal iterative argument (fixed to be the first argument) their evaluation can be performed on a common loop. The evaluation function regards this as equivalent to saving the evaluation of one subexpression. The program requires only a list showing which functions can be linked together in this way. In a later/

later stage (3.2.2.5) some of the linkings that the program thought possible may be disqualified as one linked expression is a subexpression of another. Thus our planning is done on a simplified space but the computation required is much less. Our evaluation function will always be a lower bound on the actual effort required.

Assume we have (for L-0) a linking list thus

[[union subtract]]

showing that union and subtract can be linked. Thus we can see from the expression dictionary that expressions 7, 12, 13 and 16 are candidates for linking and no others.

The results of the programs evaluation are summarised in the following table.

Combinations	Expressions required	Total no. of expressions	No of linkings	Estimated Effort
[1] [7 8] [14 15]	1 2 3 4 7 8 12 14 15	9	1	8
[1] [7 8] [14 16]	1 2 3 4 7 8 12 14 16	9	1	8
[1] [7 9] [14 15]	1 2 3 4 7 9 12 14 15	9	1	8
[1] [7 9] [14 16]	1 2 3 4 7 9 12 14 16	9	2	7
[1] [7 10] [14 15]	1 2 3 4 7 10 13 14 15	9	1	8
[1] [7 10] [14 16]	1 2 3 4 7 10 13 14 16	9	2	7
[1] [7 11] [14 15]	1 2 3 4 7 11 13 14 15	9	1	8
[1] [7 11] [14 16]	1 2 3 4 7 11 13 14 16	9	2	7
[1] [3 8] [14 15]	1 2 3 4 7 8 12 14 15	9	1	8
[1] [3 8] [14 16]	1 2 3 4 7 8 12 14 16	9	2	7
[1] [3 9] [14 15]	1 2 3 4 7 9 12 14 15	9	1	8
[1] [3 9] [14 16]	1 2 3 4 7 9 12 14 16	9	2	7
[1] [3 10] [14 15]	1 2 3 4 10 13 14 15	8	1	7
[1] [3 10] [14 16]	1 2 3 4 10 13 14 16	8	1	7
[1] [3 11] [14 15]	1 2 3 4 7 11 13 14 15	9	1	8
[1] [3 11] [14 16]	1 2 3 4 7 11 13 14 16	9	2	7
[5] [7 8] [14 15]	4 5 6 7 8 3 12 14 15	9	1	8
[5] [7 8] [14 16]	4 5 6 7 8 3 12 14 16	9	2	7
[5] [7 9] [14 15]	4 5 6 7 9 12 14 15	8	1	7
[5] [7 9] [14 16]	4 5 6 7 9 12 14 16	8	2	6
[5] [7 10] [14 15]	4 5 6 7 3 10 13 14 15	9	1	8
[5] [7 10] [14 16]	4 5 6 7 3 10 13 14 16	9	2	7
[5] [7 11] [14 15]	4 5 6 7 11 13 3 14 15	9	1	8
[5] [7 11] [14 16]	4 5 6 7 11 13 3 14 16	9	2	7
[5] [3 8] [14 15]	4 5 6 7 8 3 12 14 15	9	1	8
[5] [3 8] [14 16]	4 5 6 7 8 3 12 14 16	9	2	7

ccntd./

[5] [3 9] [14 15]	4	5	6	7	3	9	12	14	15	9	2	7
[5] [3 9] [14 16]	4	5	6	7	3	9	12	14	16	9	2	7
[5] [3 10] [14 15]	4	5	6	7	3	10	13	14	15	9	1	8
[5] [3 10] [14 16]	4	5	6	7	3	10	13	14	16	9	2	7
[5] [3 11] [14 15]	4	5	6	7	3	11	13	14	15	9	1	8
[5] [3 11] [14 16]	4	5	6	7	3	11	13	14	16	9	2	7

It can be seen from the table that the combination the program estimates will require least effort to compute is 5 7 9 14 16. This is the one that it chooses to continue with and thus our state description becomes

state0 = s1 ↓ set1, s2 ↓ set2, s3 ↓ set3, x ↓ ob1

predicate expression = or(nullset(union(set3,set2),member(ob1,set2))

statet = s1 ↓ set1, s2 ↓ union(set3,set2), s3 ↓ subtract(union(set3,set2),subtract(set3,union(set3,set2))), x ↓ ob1

statef = s1 ↓ set1, s2 ↓ consset(ob2,nilset), s3 ↓ union(set3,consset(ob2,nilset)), x ↓ ob1

with potential linkings between union(set3,set2),subtract(set3,union(set3,set2)) and union(set3,consset(ob2,nilset)).

3.2.2.4 Projection

3.2.2.4.1 Projection into B-0

For implementation in B-0 the projection is the identity.

3.2.2.4.2 Projection into L-0

For/

For implementation in L-0 the program goes through the state description transforming it to a state description in L-0 that represents the original transformation in S-1. It does this by basically projecting the original state (state0) and the final states (statet and statef), and the predicate expression into the lower domain.

The program goes through each component of the state description, replacing setnames by names for nodes and for every setname that has a non-empty membership list it adds a triple $<t_1,t_2,t_3>$ to that component of the state description where t_1 is the name of the list node this triple represents. t_2 is the name of the head of this node, a member of the set this list represents, and t_3 is the name of the node that forms the tail. The order of the elements (head values) of these lists corresponds to the order decided earlier for the membership lists of the sets (3.2.2.2).

Thus our example becomes

state0 = s1 ↓ n1, s2 ↓ n2, s3 ↓ n3, x ↓ ob1 [[n2 ob2 n3]]

predicate expression = or(nullset(union(n3,n2)),member(ob1,n2))

statet = s1 ↓ n1, s2 ↓ union(n3,n2), s3 ↓ subtract(union(n3,n2),subtract(n3,union(n3,n2)) [[n2 ob2 n3]], x ↓ ob1

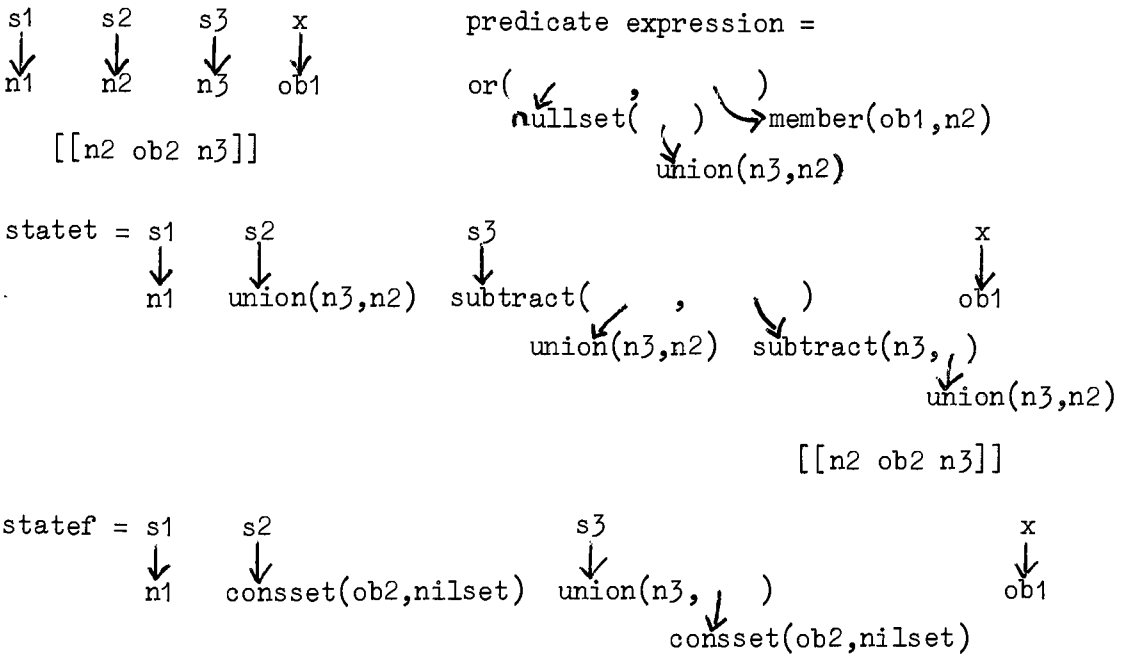
statef = s1 ↓ n1, s2 ↓ consset(ob2,nilset), s3 ↓ union(n3,consset(ob2,nilset)) [[n2 ob2 n3]], x ↓ ob1

3.2.2.5 Shrinking

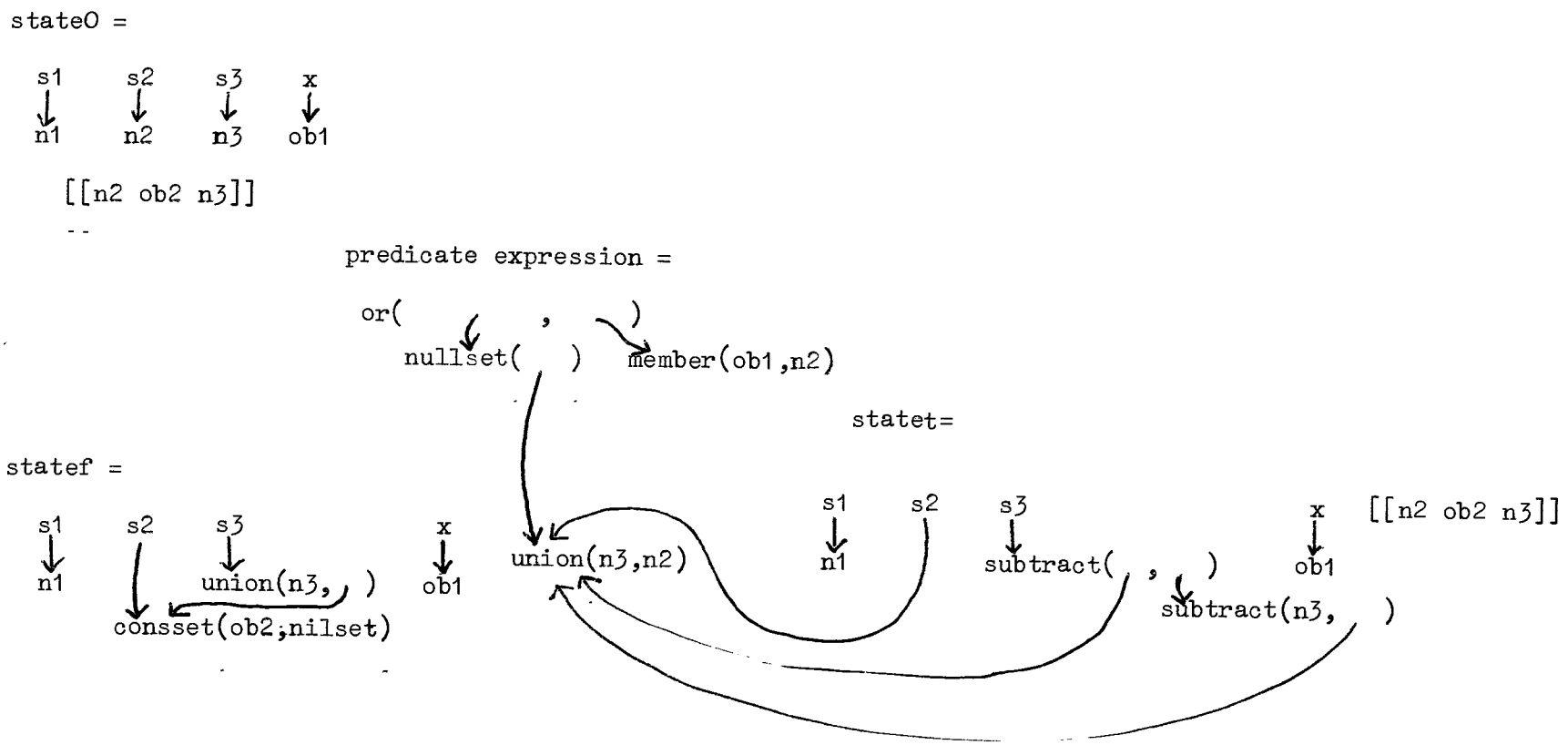
This can be thought of as pre-processing the state description to simplify the task of the program writing algorithm. The program processes the state description so that expressions that are equal become the same expression. This is only done between the predicate expression and statet, between the predicate expression and statef and within statet and statef. Equal expressions between statet and statef are left distinct.

Thus our state description in expanded tree form is

state0 =

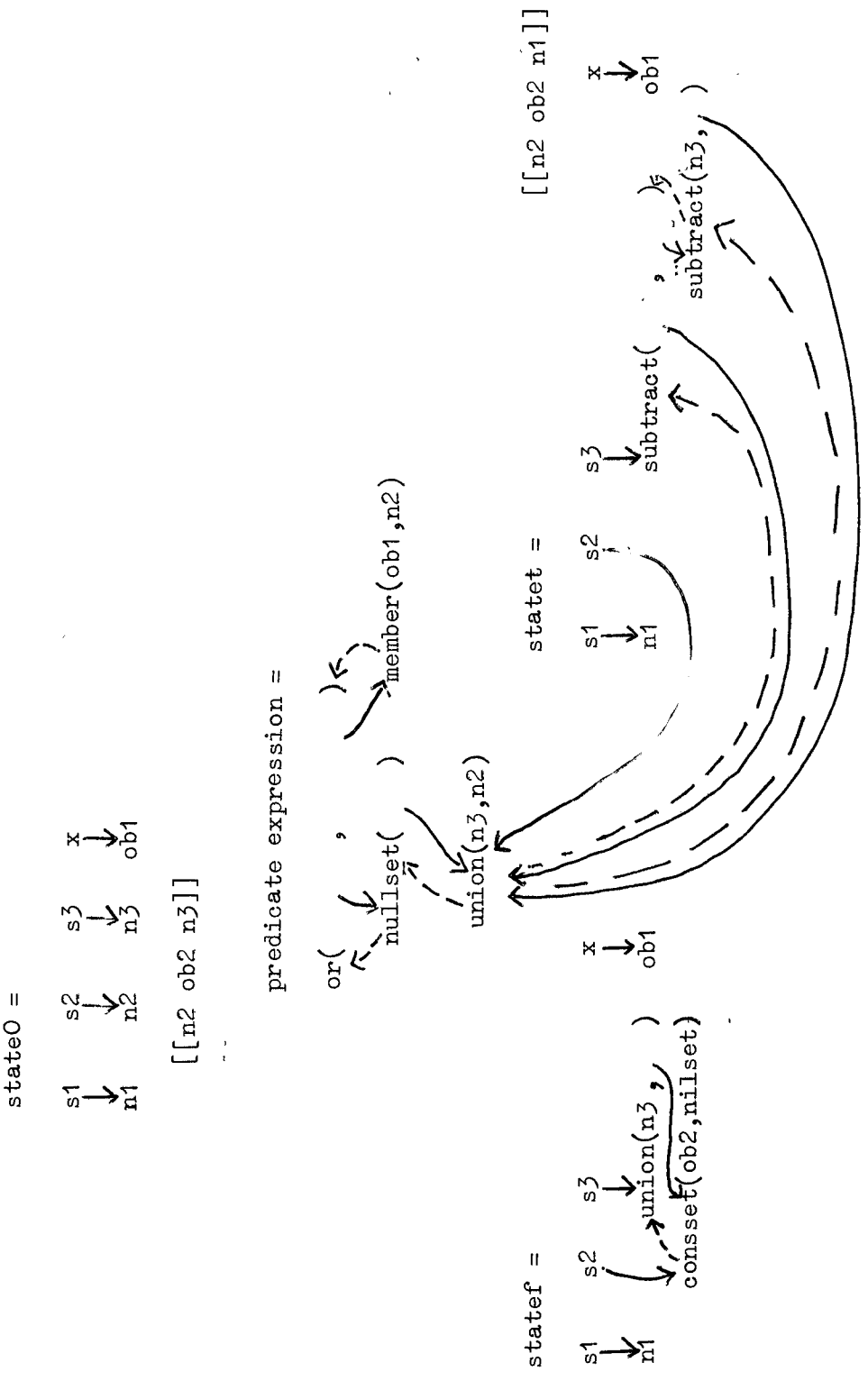


after/


state0 =
s1
s2
s3
x
n1
n2
n3
ob1
[[n2 ob2 n3]]
predicate expression =
or(,)
nullset( )
member(ob1,n2)
statet=
statef =
s1
s2
s3
x
n1
union(n3,)
ob1
consset(ob2;nilset)
union(n3,n2)
s1
s2
s3
x
[[n2 ob2 n3]]
n1
subtract(,)
ob1
subtract(n3,)

The program goes through the shrunken state description inserting backpointers. A backpointer is a pointer from an expression to any other expression which has the first expression as an immediate sub-expression.

Thus our state description becomes, with backpointers shown as - - ->



3.2.2.6 Linking

The program now goes through the state description making lists of any expressions that can be linked together. Two expressions can be linked if

(i) Their outermost functions are in the same linking class according to the appropriate linking list, that is the list showing which functions can be linked together.

For B-O the linking list is

[[nullset choose][union subtract]]

and for L-O

[[union subtract]]

As can be seen from the definitions in 2.4.3.4.

(ii) The arguments on which the iterations depend are equal.

(iii) One expression is not a subexpression of the other in the state description. This is because when two functions are linked it is required that all arguments are known. Thus in our example only two expressions (of the three thought possible in the planning stage) can be linked. Our program produces a tobelinked list of

[[union(n3,n2),union(n3,consset(ob2,nilset))]]

3.2.2.7/

3.2.2.7 Program Writing

The next task is to produce a program that will perform the transitions represented by the state description.

Although the state description is still stated in terms of primitives from S-1 these should be thought of as standing for their expansion in L-0 or B-0. The program is written mainly in terms of these S-1 primitives but with knowledge of their intended expansion, and then the expansion into B-0 or L-0 primitives is performed.

3.2.2.7.1 Code Production

The program writing task is done in 3 stages.

(i) First code is produced to perform the evaluation of the predicate expression. Any expression that will be required elsewhere, indicated by the backpointers, is saved using a temporary variable. Any expression that can share iterations with an expression in the predicate expression, as indicated by being on the to be linked list, is evaluated during the computation of the predicate expression

(ii) and (iii) Code is then produced to perform the transition represented by state0 and statet and state0 and statef. Again common computations are saved and iterations combined where possible but only within each transition not between transitions.

(i) Code for the Predicate Expression

The/

The basic function is codefor which takes an expression and produces the code that performs the evaluation of this expression. This is given in full in appendix II. As extra parameters this function takes, the present state description, a list of sub-expressions that have already been calculated and will be needed again, and a list of expressions that the program has decided will be linked. If codefor finds that it needs to evaluate an expression previously evaluated it looks up where this expression has been stored. If it finds itself evaluating an expression that will be needed again, indicated by the expression having more than one backpointer, it stores the result of its evaluation in a temporary variable and updates the list of evaluated expressions. If codefor comes across an expression that is on its to be linked list it finds all other expressions whose computation is to be linked with the computation of this expression and forms a linkedgroup which is a compound object which is used in the next phase, code expansion (3.2.2.7.2). The components of this compound item are

(i) The type of linking group this is.

(ii) A list of the top functions of the expressions involved in this common iteration.

(iii) The iterative argument.

(iv) A list of the non-iterative arguments, if any.

(v) A list of destination variables where the results of these common iterations will be stored.

As/

As well as forming the linkedgroup codefor updates its list of already evaluated expressions.

(ii) Producing Code for the State Transformations

(i) For each state, statet or statef a list of differences between that state and state0 is produced. A difference is said to occur when a variable has a different value in one state from the other.

(ii) The program then attempts to order these differences in such a way that after the removal of each difference the information required for the removal of the remaining differences is still accessible, either from the present state description or the programs list of temporarily stored values.

For each ordering of differences the program does this by simulating the action of the proposed program on the symbolic states. If there is such an ordering the program continues to the next stage which is producing the code for this ordering. If there is no such ordering the program allows itself one extra instruction to remember information that would be lost and attempts again to re-order the differences in such a way that with this extra instruction it does not lose any necessary information. The program continues in this way until it succeeds in finding such an ordering. Thus the final program produced is guaranteed optimal in the number of instructions needed.

(iii)/-

(iii) When the program has found an ordering of differences it goes through them. It uses codefor to produce the code for the new value for the variable, produces an assignment statement to that variable, updates the present state description and continues until all differences have been eradicated.

Thus in our example, the evaluation of code for the predicate expression produces

```
consset(hd(s2),nilset)->temp1; *
linkedgroup<union,[union union],s3,[s2 temp1],[temp2 temp3]>;
or(nullset(temp2),member(x,s2))
```

There are differences between state0 and statet at s2 and s3. The program produces an ordering <s2, s3> and the code corresponding to this is

```
temp2->s2;
subtract(s2,subtract(s3,s2))->s3;
```

Between state0 and statef there are again differences at s2 and s3. Any ordering will do. The code produced is

```
temp1->s2;
temp3->s3;
```

3.2.2.7.2/

---

* In implementation in L-0 any access of a node in the list description component of the state produces the appropriate accessing hd tl functions.

3.2.2.7.2 Code Expansion

This is straightforward. The program has associated with each primitive in S-1 the code that represents that primitive in B-O or L-O and the program expands the code using these.

The only interesting feature is the expansion of a linked group. Corresponding to each linkedgroup type there is a schema and for every function of that type an expansion. For each linked group the program expands each function and then fits these expansions into the schema. Thus in our example the linkedgroup is

<union,[union union],s3,[s2 temp1],[temp2 temp3]>

Associated with union is the schema

<bit1>
applist(<iterativeargument>,lambda bv; <bit2>)

And associated with union the code schema

<secondargument>-><result> for bit1

and if not(member(bv,<secondargument>)) then
cons(bv,<result>)-><result>
for bit2

Thus bit1 is expanded to

s2/

```
      s2->temp2;
      temp1->temp3;
```

and bit2 to

```
      if not(member(bv,s2)) then cons(bv,temp2)->temp2
      if not(member(bv,temp1)) then cons(bv,temp3)->temp3
```

Thus the whole linked group expands to

```
      s2->temp2;
      temp1->temp3;
       applist(s3,lambda bv; if not(member(bv,s2)) then cons(bv,temp2)->temp2
                             if not(member(bv,temp1)) then cons(bv,temp3)->temp3
                                  );
```

The whole example expands to

```
     while [if member(hd(s2),nilset) then nilset->temp1
                 else cons(hd(s2),nilset)->temp1;
            s2->temp2;
            temp1->temp3;
            applist(s3,lambda bv; if not(member(bv,s2)) then cons(bv,temp2)->temp2;

                                  if not(member(bv,temp1)) then cons(bv,temp3)->temp3;

            or(null(temp2),member(x,s2))]
```

do/

```
do [temp2->s2;
    nil->tvar1;
    applist(s3,lambda bv; if not(member(bv,s2)) then cons(bv,tvar1)->tvar1)

    nil->s3;
    applist(s2,lambda bv; if not(member(bv,tvar1)) then cons(bv,s3)->s3)];

    temp1->s2;
    temp3->s3;
```

3.2.2.8 Examples

(i) For another example we will take the program described earlier (3.1.4). This function takes a set s and a pair p and if there is an element in s equal to the front of p it returns {back(p)} otherwise it returns the empty set.

In S-1, with front and back considered basic, this is

```
while and(not(nullset(s)),not(equal(choose(s),front(p))))
   do [p->xsave; minus(choose(s),s)->s; xsave->p]
      if nullset(s) then nilset->ans else consset(back(p),nilset)->ans;
```

Implemented in B-O this becomes

while/

```
while [true->res2; length(s)->l; 1->n;
       while n≤l
          do [if intsub(n,s) = 1
                 then[n->res1; true->res2]
              n+1->n]
       and(not(res2),not(equal(res1,front(p))))]
   do [p->xsave; 0->intsub(res1,s)]
if res1 then nilset->ans else [nilset->ans; 1->intsub(back(p),ans)]
```

and in L-0

```
     while [null(s)->temp1;
            and(not(temp1),not(equal(hd(s),front(p))))]
        do [tl(s)->s; p->xsave]
     if temp1 then nil->ans else if member(back(p),nil)
                                    then nil->ans
                                    else cons(back(p),nil)->ans;
```

(ii) A function highest that picks the highest integer from a set.

In S-0

$$\text{highest(set)} = \begin{cases} \text{nullset(set)} \longrightarrow 0 \\ \text{not(nullset(set))} \longrightarrow \text{if choose(set)>highest(minus(choose(set),set))} \\ \qquad\qquad \text{then choose(set)} \\ \qquad\qquad \text{else highest(minus(choose(set),set))} \end{cases}$$

In/

In S-1

```
0->ans;
while not(nullset(set))
   do [if choose(set) > ans then choose(set)->ans;
           minus(choose(set),set)->set]
```

Again our compiler finds the natural implementation for this in B-0.

```
0->ans; length(set)->l; 1->n;
  while n≤l
     do [if intsub(n,set) = 1
           then[if n > ans then n->ans;
                0->intsub(n,set)]
         n+1->n]
```

and in L-0

```
0->ans;
  while not(null(set))
     do [if hd(set) > ans then hd(set)->ans;
         tl(set)->set]
```

(iii) An interesting class of programs that one can write using the 'true' set primitives, that is, those that operate on sets as entities, are the one line programs. These are just expressions in the given primitives. We feel the ability to write programs without using the control primitives is an indication of the power of these primitives/

primitives and we will have more to say on this in connection with APL, Iverson (1962) in chapter 5.

Suppose we are asked to write a program that would select from given sets all people who were over 6' together with all those people with dark hair who were under 6' and all those people who have dark hair but don't speak English.

Letting U = all people in the sample

A = all people over 6'

B = all people who have dark hair

C = all people who don't speak English

This transcribes directly into the 1 line program

$$(A \cup ((U/A) \wedge B)) \cup (B/C)$$

Continuing $\cup$ and / as shorthand for union and subtract. Our compiler is able to implement this involving 2 iterations. The expression reduces to

$$(B \cup A) \cup (B/C)$$

and $B \cup A$, and B/C can be linked and implemented as a single iteration.

The deduction the compiler performs to achieve this is

(A/

$(A \cup ((U/A) \wedge B))$

$= (A \cup (U/A)) \wedge (A \cup B)$ by $A \cup (B \wedge C) = (A \cup B) \wedge (A \cup C)$

$= (U \wedge (A \cup B)$ by $A \cup (U/A) = U$

$= A \cup B$ by $U \wedge A = A$

$= B \cup A$ by $A \cup B = B \cup A$

The point that our compiler performs these manipulations not just to minimise the set expression but to optimise the future implementation is illustrated by the fact that it does not reduce

$(A \cup B) \cup (A \cup C)$

Instead it links the iterations involved in the evaluation of $A \cup B$ and $A \cup C$.

3.2.3 Destructive Implementation of L-0 Programs (L-0=>L-1)

We here describe the final stage of one branch of our implementation, namely the implementation of L-0 programs as destructive (L-1) programs. Again this process is divided into the same subprocesses, namely analysis of transformations, optimisations, projection and program writing. The sections of program dealt with by this process are the same as previously.

3.2.3.1 Extraction of Transformation

This is straightforward and we will not describe it in detail, the only novelty is that the state descriptions contain no free expressions, only variable values and list fragments as defined in 3.2.1.5. The program invents an initial symbolic vector and then interprets the program text on this according to the semantic definitions given in 3.2.1.5.

3.2.3.2 Optimisation of Transformation

The optimisation done at present is rather arbitrary. We concentrate on avoiding the introduction of any new list cells. This is dectectable from the state descriptions. If the program finds that any such cells have been introduced, it searches the state description for any cell that has been discarded, i.e. one that is not accessible in the final state and sees if it is possible to use this in place of the old cell. If it can do this it rearranges the state description to do this.

3.2.3.3 Projection

As the semantic apparatus used for L-0 and L-1 is the same the projection is the identity.

3.2.3.4 Program Writing

The algorithms used to transform the state descriptions into an L-1 program are very similar to those used for the earlier program writing activities (S-1 to B-0 or L-0). However we will describe them separately and take this opportunity to use a different mode of explanation. The process is simpler as there are no iterative operators nor any that expand further. We will thus describe how we convert a pair of before-after state vectors into a piece of straight line L-1 code.

Given two states

$$S = \langle V,A,L,C\rangle \text{ and } S' = \langle V',A',L',C'\rangle$$

We need to define a function

makeprogram: state×state->instructions

which will produce a program to perform the transformation represented by the state pair.

We say that a difference occurs between the two states whenever either

(i) $\exists\, v(v \in V' \wedge v \notin V)$

or/

or (ii) $\exists a(a \in A' \wedge a \notin A)$

or (iii) $\exists v(v \in V' \wedge (L(v) \neq L'(v)))$

or (iv) $\exists a((a \in A') \wedge (a \in A) \wedge (C(a)_1 \neq C'(a)_1))$†

or (v) $\exists a((a \in A') \wedge (a \in A) \wedge (C(a)_2 \neq C'(a)_2))$†

Two states are said to be equal when there are no differences between them. Thus a program has to remove all the differences. However removing one difference may lead to the loss of access to a node needed later.

Define: Nodesaccessible: State->Nodes*

such that nodesaccessible(S) where S = <V,A,L,C>

= N where range(V) $\subseteq$ N

and if $n \in N$ then $C(n)_1 \in N$

and $C(n)_2 \in N$

if they are defined.

Define: differences: statexstate->differences*. A function that calculates all the differences between two states.

Define a function Perm: ob*->(ob*)* which takes a list onto a list of permutations of the list.

Define a function process: statexdiff->state which maps a state and a difference onto the state resulting from the removal of that difference.

† $\langle x,y \rangle_1 = x$
$\langle x,y \rangle_2 = y$

Thus given two states: beginstate and finalstate we try to find an ordering of the differences between the two states that can be translated into a program. That is an ordering such that when we come to remove a difference the information needed is available. If there is no such ordering we have to introduce extra instructions to remember such information.

More exactly let Permlist = Perm(Differences(beginstate,finalstate)). We need to find a member of Permlist, perm, say, such that if

perm = $\langle diff_{i_1}, diff_{i_2}, \ldots, diff_{i_n} \rangle$ and we define

$state_0$ = beginstate

$state_{j+1}$ = process($state_j$,$diff_{i_j}$)

(thus $state_n$ = finalstate)

then we require that

$$\forall i(0 \leq i < n)\ \forall (node)(node \in ncdesaccessible(state_i) \wedge node \in nodesaccessible(finalstate) \supset node \in nodesaccessible(state_{i+1}))$$

If there is no member of Permlist that satisfies the above criterion then we require a member of Permlist that **satisfies the above condition for the most i.** For each i that the condition is false we need to introduce an extra instruction to remember the information that would otherwise be lost.

Given/

Given this ordering of the differences it is then straightforward to convert this into a program. Each difference, or requirement to remember information, gives rise to an assignment instruction.

Define a function: routeto: state×node->{v,"hd","tl"}* which maps a state and a node onto an expression which describes how that node can be accessed in the state from a variable via the hd, tl pointers.

Each type of difference gives rise to a different type of assignment instruction which removes that difference. After an assignment instruction has been produced the state is updated and the next difference processed. For example consider the state pair

$$S_b = \langle V_b, A_b, L_b, C_b \rangle \text{ and } S_f = \langle V_f, A_f, L_f, C_f \rangle$$

Say that the first difference in the order chosen is a type (iv) difference (see above) i.e.

$$a \in A' \wedge a \in A \wedge (C_f(a)_1 \neq C_b(a)_1)$$

then the instruction produced is of the form

$$\text{routeto}(S_b, C_f(a)_1) \text{->} \text{hd}(\text{routeto}(S_b, a))$$

S_b is then updated to be process(S_b, diff_1), the next difference chosen and the process repeated until no differences are left.

As/

As it is apparent that each difference requires one assignment instruction (the existence of type (ii) differences complicates this slightly, the introduction of a new node and the placing of this node in the structure removes two differences) it can be seen that the above method produces an optimal program in the sense that for any state pair there is no program with fewer instructions than the one we produce that will remove all the differences between the two states.

For a fuller example let us consider the program produced earlier in section 3.1.3.4 on page 62. Here the state pair would be

$$S_1 = \langle V_1, A_1, L_1, C_1 \rangle \qquad S_2 = \langle V_2, A_2, L_2, C_2 \rangle$$

where $V_1 = V_2 = \{x, ans\}$

$A_1 = A_2 = \{a_1, a_2, a_3, a_4\}$

$L_1 = \{\langle x, a_1 \rangle, \langle ans, a_2 \rangle\}$

$L_2 = \{\langle x, a_4 \rangle, \langle ans, a_1 \rangle\}$

$C_1 = \{\langle a_1, \langle a_3, a_4 \rangle\rangle\}$

$C_2 = \{\langle a_1, \langle a_3, a_2 \rangle\rangle\}$

There are therefore 3 differences between the states, two of type (ii) and one of type (v)

viz.:- 1 $L_1(x) = a_1 \wedge L_2(x) = a_4$

2 $L_1(ans) = a_2 \wedge L_2(ans) = a_1$

3 $C_1(a_1)_2 = a_4 \wedge C_2(a_1)_2 = a_2$

It/

It will be seen that any permutation of these 3 differences is untranslatable into a program, for example attempting to remove the differences in the order <1,2,3> results in node a_1 being lost. However allowing ourselves one extra instruction perms <1,3,2>, <2,1,3> and <3,2,1> are suitable. The program came across <3,2,1> first and accordingly produces

```
tl(x)->newvar1;
ans->tl(x);
x->ans;
newvar1->x;
```

3.2.3.4.1 Further Examples and Remarks

(i) Swapping two variables. S->S' where S = <V,A,L,C> and S' = <V',A',L',C'>

where $V = V' = \{x,y\}$

$A = A' = \emptyset$

$L = \{\langle x,a_1\rangle,\langle y,a_2\rangle\}$

$L' = \{\langle x,a_2\rangle,\langle y,a_1\rangle\}$

$C = C' = \emptyset$

The program produced was

```
x->newvar1;
y->x;
newvar1->y;
```

(ii)/

(ii) We desire to produce a program that will cyclically permute the values of six variables. The state transition is represented thus.

S->S' where $V = V' = \{X,Y,Z,P,Q,R\}$

$A = A' = \emptyset$

$L = \{\langle X,a1\rangle,\langle Y,a2\rangle,\langle Z,a3\rangle,\langle P,a4\rangle,\langle Q,a5\rangle,\langle R,a6\rangle\}$

$L' = \{\langle X,a2\rangle,\langle Y,a3\rangle,\langle Z,a4\rangle,\langle P,a5\rangle,\langle Q,a6\rangle,\langle R,a1\rangle\}$

$C = C' = \emptyset$

when given this state pair the program produced

```
X->newvar1;
Y->X;
R->newvar2;
newvar1->R;
Z->Y;
Q->newvar3;
newvar2->Q;
P->Z;
newvar3->P;
```

The production and printing of this program took about 6 seconds. Note that it is not optimal as regards the number of new variables introduced. In fact whenever an 'extra' instruction is needed we invent a new variable. By doing more work we could find if any variables or accessible location was redundant.

(iii)/

(iii) An example involving cons.

S->S' where $V = V' = \{X1, X, Y\}$

$A = \{a1, a2, a3, a4, a5, a6, a7\}$

$A' = A \cup \{a8\}/\{a1\}$

$L = \{\langle X1, a1\rangle, \langle X, a2\rangle, \langle Y, a5\rangle\}$

$L' = \{\langle X1, a3\rangle, \langle X, a4\rangle, \langle Y, a8\rangle\}$

$C = \{\langle a_2, \langle a_3, a_4\rangle\rangle, \langle a_5, \langle a_6, a_7\rangle\rangle\}$

$C' = \{\langle a_2, \langle a_3, a_4\rangle\rangle, \langle a_5, \langle a_6, a_7\rangle\rangle, \langle a_8, \langle a_3, a_5\rangle\rangle\}$

produces:

cons(hd(X),Y)->Y;

hd(X)->X1;

tl(X)->X;

(iv) S->S' where $V = V' = \{X\}$

$A = \{a_1, a_2, a_3\}$

$A' = \{a_1, a_2, a_4\}$

$L = L' = \{\langle X, a1\rangle\}$

$C = \{\langle a_1, \langle a_2, a_3\rangle\rangle\}$

$C' = \{\langle a_1, \langle a_2, a_4\rangle, \langle a_4, \langle a_2, a_2\rangle\rangle\}$

gives

cons(hd(X),hd(X))->tl(X);

(v)/

(v) S->S' where $V = V' = \{X,Y,Z\}$

$A = A' = \{a_1,a_2,a_3,a_4,a_5\}$

$L = \{<X,a_1>,<Y,a_3>,<Z,a_5>\}$

$L' = \{<X,a_2>,<Y,a_1>,<Z,a_1>\}$

$C = \{<a_1,<a_2,a_3>>,<a_3,<a_4,a_5>>\}$

$C' = \{<a_1,<a_2,a_3>>,<a_3,<a_5,a_4>>\}$

gives

hd(Y)->tl(Y);

X->Y;

Z->hd(tl(X));

X->Z;

hd(X)->X;

At present the program uses no heuristics in its attempt to produce a program. However certain obvious heuristics could be tried. When removing differences it is usually better to affect nodes that are accessible in more than one way. Also for certain configurations of nodes, e.g. those configurations where each node can only be accessed one way, one extra instruction at least is going to be necessary if there is at least one difference between the states and all the information is to be retained.

3.2.3.5/

3.2.3.5 Further Example

A program in L-0 to sort a list of numbers using a swapping technique. A confession, this program can be re-implemented destructively by our compiler. There do exist equally admissible versions that our compiler does not convert.

```
true->doneshuffle;
  while doneshuffle
     do [false->doneshuffle; nil->result; hd(xl)->last;
         while not(null(tl(xl)))
            do [if last≤hd(tl(xl)) then
                   [cons(last,result)->result;
                    tl(xl)->xl; hd(xl)->last]
                else [tl(xl)->xl;
                      cons(hd(xl),result)->result;
                          true->doneshuffle]
                ]
         cons(last,result)->result; rev(result)->xl]
```

(Producing the result in xl.)

The two straight line pieces of text that our program alters are the two branches of the conditional viz.

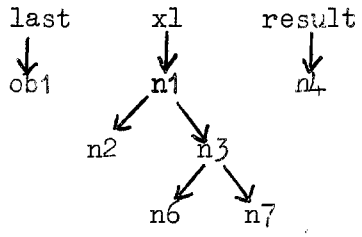
cons/

```
cons(last,result)->result;
tl(xl)->xl;
hd(xl)->last;
```

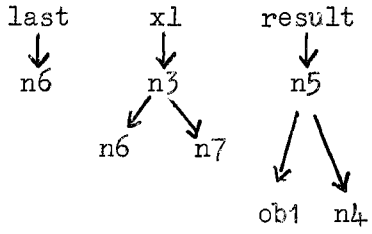
and

```
tl(xl)->xl;
cons(hd(xl),result)->result;
true->doneshuffle;
```

For the first it invents a starting state description



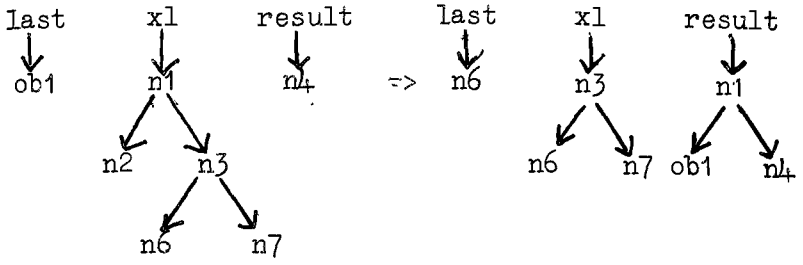
interpreting this on the above text it produces a final state description



where n5 is the new node introduced by the cons.

The optimisation routine, on being given the above transition notes that the cell n1 is no longer pointed to and uses this to rewrite the final state producing a transition

last/



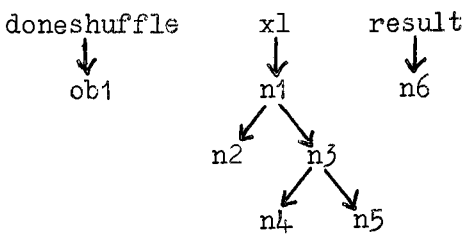
This is then given to the program writing routines which produce a program

```
last->hd(xl);
hd(tl(xl))->last;
tl(xl)->newvar1;
result->tl(xl);
xl->result;
newvar1->xl;
```

The second piece of code viz.

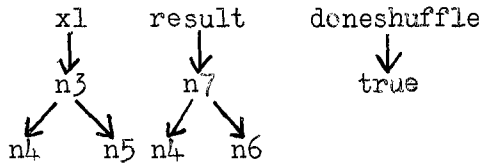
```
tl(xl)->xl;
cons(hd(xl),result)->result;
true->doneshuffle;
```

needs a starting state

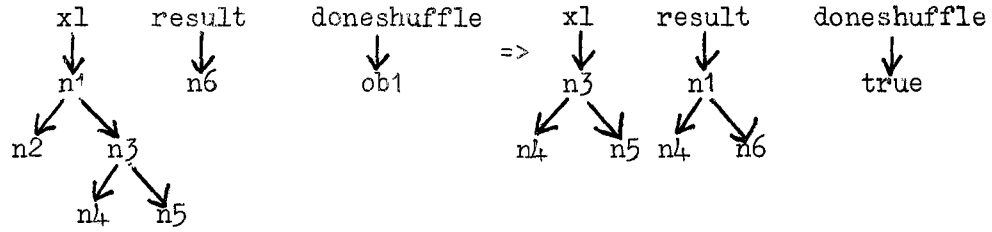


which/

which on being interpreted produces a final state



Again this transition can be optimised, the routines making use of the discarded cell to produce a new transformation



On being given this the program writing routines produce

```
tl(xl)->newvar1;
hd(tl(xl))->hd(xl);
result->tl(xl);
xl->result;
newvar1->xl;
true->doneshuffle;
```

Thus on being re-assembled the program our compiler produces is

```
true/
```

```
true->doneshuffle;
  while doneshuffle
     do [false->doneshuffle; nil->result; hd(xl)->last;
  while not(null(tl(xl)))
     do [if last≤hd(tl(xl)) then
           [last->hd(xl); hd(tl(xl))->last;
            tl(xl)->newvar1; result->tl(xl);
            xl->result; newvar1->xl]
         else [tl(xl)->newvar1; hd(tl(xl))->hd(xl);
         result->tl(xl); xl->result; newvar1->xl;
         true->doneshuffle]
                    ]
  cons(last,result)->result; rev(result)->xl]
```

3.3 Algebraic Manipulation Routines

The algebraic manipulation routines are used by the recursion translation (3.1.3.3) and compilation (3.2.2.3.1) routines. The algebraic manipulation has been kept simply to what is required for these two activities and there has been no attempt to write a powerful and general algebraic routine.

3.3.1 Specification

The basic properties of the primitives are specified as a set of pairs of formulae, indicating that each pair of formulae are equivalent. For example, if we wished to specify that union was commutative, associative and that nilset was the identity we would give the set

{<union(x,y),union(y,x)> , <union(x,union(y,z)) , union(union(x,y),z)>
,<union(x,nilset),x>}

where x y and z are variables.

The basic function is produceallforms

produceallforms: expressionxset of rules->set of expressions which given an expression and a set of rules produces a set of expressions that are equivalent to the given expression by one application of any rule.

Produceallforms/

Produceallforms uses a function

produceforms: expressionxset of rules->set of expressions

that produces all expressions that can be produced from the given expression by applying any rule to the whole of the expression.

```
produceallforms(exp,ruleset) =
    let resultset=∅
      appset(produceforms(exp,ruleset),
             lambda ex;
              cases ex:
               isconstant
               or isvariable: let resultset =
                                 produceforms(exp,ruleset)
                  f(e1)     : appset(produceallforms(e1,ruleset),
                                     lambda arg;
                                      let resultset =
                                         {f(arg)} ∪ resultset)
                  f(e1,e2)  : appset(produceallforms(e1,ruleset),
                                     lambda arg1;
                                      appset(produceallforms(e2,ruleset),
                                             lambda arg2;
                                              let resultset =
                                                 {f(arg1,arg2)} ∪ resultset
                                             ));
                           );
     resultset;
```

produceforms/

```
produceforms(exp,ruleset) =
   let expset={exp}
    appset(ruleset, lambda rule;
                    if trymatch(front(rule),exp) then
                      let expset =
                       {substitute(back(rule),makematch(front(rule),exp))} ∪ expset
                    else if trymatch(back(rule),exp) then
                      let expset =
                        {substitute(front(rule),makematch(back(rule),exp))} ∪ expset
                        )
   expset;
```

appset: α-set×(α->())->()

This function applies a function to every element of a set

trymatch: expression×expression->T/F

This function is true if the second expression can be produced from the first by instantiating some variables. False otherwise.

makematch: expression×expression->m-list

This function produces a list of matches of variables from the first expression to subexpressions of the second expression that perform the instantiation mentioned in trymatch.

substitute:/

substitute: expressionxm-list->expression

This function instantiates variables in the expression according to the matchings given in the matching list.

CHAPTER 4.

Theory of Computation and Computer Science

In this chapter we aim to give a brief survey of some of the aims and developments in the main line of Computer Science and in the Theory of Computation and develop a synthesis between some of these developments that sets the scene for our work.

4.1 Programming and Machine Design

One of the more remarkable features about the development of the computer is that although this has been an area of rapid technological advance the basic design principle or model on which almost all computers are based is fundamentally unaltered since the first true computer Edsac was completed in 1949. Despite the fact that the design of system components used to implement this model has been improved enormously. Most recent computers with the significant exception of the B5000 follow the von Neuman model of which the most important concept is that of a linear addressable store that is shared between program and data. We follow Iliffe (1968) in believing that this 'engineering model' of computation persists in the design of present day high level language and compilers in such a way as to prevent them fully realising their aims. "One should define a component of a computer by what it does rather than by what it is." If we apply this criterion to present day high level languages we see that/

that this criterion is not fully satisfied. These languages [Algol, Fortran, Cobol] were developed to allow programmers to specify algorithms conveniently and independently of any particular machine. However it is clear that several features of the engineering model or the actual implementation of the language are still present. The goto statement is an example and there has been discussion, Dijkstra (1968) as to whether its use inhibits the development, understanding and modification of programs. The concensus and the view to which we subscribe is that it does. We shall attempt to amplify this later. There are other features of high level problem oriented languages which seem to owe little to the problem areas they are supposed to tackle and much to actual methods used to implement these features. Here we could cite the various calling mechanisms and variable binding conventions of Algol. Iliffe again:- "The common problem oriented languages conveniently express the structure of conventional machines and some of the problems to which they are suited and the investment in, for example, Fortran or Cobol programming can be assured by carrying such standards into future machines."

More recently, languages have been developed [LISP, POP-2, Planner] designed to reflect structures more appropriate to problem solving. However, as these languages have still to be run on von Neuman-type machines we are faced with the problems of reducing these structures to fit the logic of the machine. 'Existing compilers only 'recognise' structure by virtue of restrictions on input languages which enables it/

it to be absorbed during translation or by the use of interpretive (and therefore slow) execution modes.' It is our contention that the two aims (i) ease or succinctness of programming and (ii) efficient implementation, are contradictory given the persistance of the von Neuman model. Recursion is a case in point. Its use makes writing of some programs easier, but often efficient implementation of these programs is difficult. We do not want the programmer to be concerned with machine features but often it is concern with just these features of programming that makes for efficient programming. Not all this dichotomy is due to present machine structure. It is often implicit in the nature of the algorithm for example the Fibonacci function. It is these areas with which we concern ourselves. We have approached language design from the users viewpoint and specified a simple language in which we think it is natural for a programmer to specify some problems and then investigated the problems involved in implementing these programs efficiently for the present model of machines. We hope to have demonstrated that it is possible to reconcile these aims but that it requires the design of compilers that understand programs to a far greater extent and possess some limited problem solving or inferential capacity. The problem is one of optimisation but we believe that as more and more natural languages are developed, local low level optimisation will be insignificant. Thus we view compilation of a process as involving a) understanding what the process does and b) implementing the process as efficiently as possible in the target language.

4.2/

4.2 Theory of Computation

In this section we wish to discuss interconnected aspects of Theory of Computation, the semantic definition of programming languages, proving properties of programs and automatic program writing.

4.2.1 Languages and Meta-Languages

Semantics may be defined as the study of the relation between objects and their representation, programming language semantics is concerned with the relationship between programs of a programming language and the objects (functions, data) which they denote. The approaches taken to program language semantics may be broadly split into two camps, the operational approach and the denotational or mathematical approach.

The operational approach requires that meaning be defined in terms of transformational properties rather than in terms of non-operational correspondence between a program and the abstract object it represents. This approach to semantics is representation-dependent since the semantics of a given function will differ depending on the programming language in which it is represented. By contrast the denotational approach attempts to provide rules that allow, for any program written, the mathematical function associated with the program to be produced.

Common/

Common to both these approaches is the need for a semantic meta-language for defining the meaning of a class of object languages. The reasons behind this effort are the desire to clarify and 'formalise' aspects of programs and programming languages and to facilitate investigation of the properties of languages and programs written in the languages. Whether one follows the operational or denotational approach to semantics seems to depend on what questions you are asking. If you are interested in the transformational aspects of certain representations you would prefer the latter. For our purpose we assume that all a programmer is concerned with is what his program viewed as a function should do. He is, therefore, more interested in denotational semantics. Wegner (1969) puts forward four aspects desirable in a semantic meta-language. It should be natural, it should be elegant when considered as a programming language in its own right, it should be simple and it should facilitate proofs of properties. We suggest that from the users viewpoint these are all necessary aspects of a good program specification language.

4.2.2 Proofs about Programs

One of the uses of semantic meta-languages has been to facilitate the proving of properties about programs. We give reasons why this activity has not been too successful and suggest why it may be to some extent misguided.

Some/

Some of the reasons for slow progress are clear. People have attempted the axiomatisation of existing languages and those features that we pointed to as making them difficult to understand also make them difficult to axiomatise concisely. One of the gravest defects that most of these languages possess is the existence of referential opacity manifest in side effects which have their genesis in the fact that objects are located in physical store. Referential transparency has long been considered essential for any decently manipulable mathematical system.

As Dijkstra has suggested the development of program proving as a separate activity from program creating also seems mistaken and involves duplication of many of the tasks of the programmer in that the same understanding about a given program needs to be brought to bear on both occasions.

The approach to proving programs is to provide ways of associating with programs expressions in a meta-language. Questions about these programs are then answered within the meta-language. To prove a program one has to write down what one expects the program to do. We propose that this distinction between programming language and meta-language should be avoided and compilers be designed to accept programs or specifications of programs written in a language clear and concise enough to be thought of as a meta-language and compile them efficiently. Thus proofs of programs could either be represented as transformations within this language or should become unnecessary as no more adequate definition exists of what a program should do than the program itself.

4.2.3 Program Writing

A parallel activity to program proving within Theory of Computation has been the search for automatic program writing systems. The typical approach of the theorem prover school of automatic program writing (Green (1969), Waldinger and Lee (1969), Waldinger and Manna (1971)) is for the user to specify the desired computation by means of input/output predicates stated in first order predicate calculus. A theorem induced by these specifications is then proved and the desired program which is guaranteed correct (relative to the I/O predicates) is then extracted from the proof. In all these attempts the inductive step or iterative structure of the program has to be given by the user either implicitly in the I/O predicates or by explicitly telling the system the iterative structure the final program will have.

If one considers what is happening in these cases it is that one is using the language of the I/O predicates as a source language to describe what the desired computation should do. What is important is not that predicate calculus is being used as a language but the equational forms that are allowed to the programmer for him to specify his program and the primitives he has available.

Consider a high level specification of an algorithm in terms of recursion equations, for example, Euclid's algorithm for the H.C.F.

$$hcf(n,m) = \begin{cases} m>n \longrightarrow hcf(m,n) \\ m=0 \longrightarrow n \\ (m\leq n) \wedge (m\neq 0) \longrightarrow hcf(m,rem(n,m,0)) \end{cases}$$

$$rem(n,m,r) = \begin{cases} n\leq m+r \longrightarrow n-(m+r) \\ n>m+r \longrightarrow rem(n,m,m+r) \end{cases}$$

These definitions have the form

$$\begin{aligned} f_1(x_1,\ldots,x_{n_1}) &= \mathcal{E}_1(x_1,\ldots,x_{n_1},f_1,f_2,\ldots,f_m) \\ f_2(x_1,\ldots,x_{n_2}) &= \mathcal{E}_2(x_1,\ldots,x_{n_2},f_1,\ldots,f_m) \\ &\vdots \\ f_m(x_1,\ldots,x_{n_m}) &= \mathcal{E}_m(x_1,\ldots,x_{n_m},f_1,\ldots,f_m) \end{aligned}$$

where $\mathcal{E}_i(x_1,\ldots,x_n,f_1,\ldots,f_m)$ denotes an expression that may concern $x_1,\ldots,x_n,f_1,\ldots,f_m$ and other basic functions or constants. These can be said to have imperative content as there exist computation (or run time) systems (see Cadiou (1972)) that will for any given input assignment to $x_1,\ldots,x_n$ reduce the program if it is correct to an expression not involving $f_1,\ldots,f_m$. These definitions also have assertional content in that given an input assignment and a proposed answer the equations can be used to check whether the answer is correct. Our compiler translates this specification of a program into another specification for which there is a different evaluation mechanism (run time system).

We distinguish two further developments on the way to a fully fledged automatic program writing system.

(i) The restrictions on the equational form may be relaxed to allow constructs of the basic functions and variables to appear in the place of/

of variables on the left hand side. For example, one may find it easier to define the Fibonacci system as

Fibonacci(n+2) = Fibonacci(n+1)+Fibonacci(n)

and Fibonacci(1) and Fibonacci(0) = 1.

Of course, equations of this form do not always define unique functions e.g.

$f(x^2)=x$.

(ii) A fully fledged program writer completely relaxes the restriction on the left hand side, and allows general equational forms.

$$\begin{aligned} \mathcal{E}_1(x_1,\ldots,x_n,f_1,\ldots,f_m) &= \mathcal{E}'_1(x_1,\ldots,x_n,f_1,\ldots,f_m) \\ &\vdots \\ \mathcal{E}_l(x_1,\ldots,x_n,f_1,\ldots,f_m) &= \mathcal{E}'_l(x_1,\ldots,x_n,f_1,\ldots,f_m) \end{aligned}$$

For example it seems most natural to define a function to compute the number of times y divides x by

$f(x,y).y=x$

$f(x,x)=1$

The first task as we see it for any program writer is to translate this specification into one which has some imperative content, that is, into equations of the first sort. This we see as being in two parts (i) Manipulating the equations into the form

$f_i(x_1,\ldots,x_n) = \mathcal{E}'_i(x_1,\ldots,x_n,f_1,\ldots,f_m)$, $1\leq i\leq m$ and (ii) Ensuring that/

that these manipulations produce equations with the 'desirable' imperative content. If the manipulations have been done correctly the equations produced will be correct, what is important is to ensure that the computation proceeds in the right direction and terminates. Thus in our example the manipulation

$$f(x,y).y=x$$
$$\therefore\ f(x,2.y).2.y=x$$
$$\therefore\ f(x,2.y).2.y=f(x,y).y$$
$$\therefore\ f(x,y)=2.f(x,2.y)$$

indicates a recursive definition

$$f(x,y) = \text{if } x=y \text{ then } 1 \text{ else } 2.f(x,2.y)$$

which is correct and in which the computation proceeds 'in the correct direction' but it does not always terminate.

We do not know exactly how to do this but if we could it would be a preliminary stage to our program and extend it towards an automatic program writing system.

CHAPTER 5.

In this chapter we hope to explore the possibilities for our compiler. In 5.1 we examine some suggested improvements and in 5.2 we hope to show how our transformational compiler could be generalised to provide an intelligent compiler/improver for any abstract programming system that a user may wish to define.

5.1 Some Improvements

At present the program writing routines can produce effectively only straight line text. An obvious extension would be to provide our programs with some inductive capability. There are two factors against this

a) It is difficult.

b) It can be avoided to a certain extent.

We feel that in a properly structured language system iterations in one level are often realisations of primitives that operate non-iteratively on compound objects at a higher level. For example in a list system functions such as concat and rev are single primitives that operate on lists as objects. They are realised lower down as iterations in a virtual machine that deals more with list-cells i.e. its characteristic functions are hd and tl which operate on list-cells rather than on lists.

Thus/

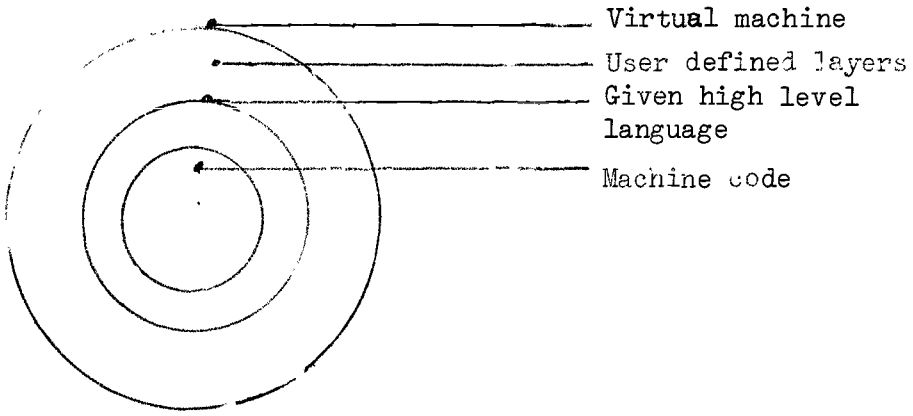
Thus if we wish to manipulate iterations at one level it is perhaps better to do this by seeking, or keeping a record during the compilation, the higher level piece of code from which these iterations originate and performing the manipulations here on hopefully non-iterative expressions and then projecting back to the lower level. Our compiler does this in a hierarchical manner that is just across one layer and in one direction only. A full heterarchical system, Winston (1972), which allows communication between any two layers would be an improvement.

5.2 Language Extension

One of the advantages of a language that allows the definition of functions [LISP, POP-2, Algol] is the ease with which one is able to add to the given primitives. This is done by defining new data objects in terms of given data objects and defining new primitives in terms of given primitives to act over these data objects. This enables one to produce new abstract systems which facilitate writing programs in certain subject areas. For example systems can be produced to deal with trees, relational structures, partial functions, or general algebras, within mathematical domains and files or structured data banks within more general domains.

Typically/

Typically this building of new virtual machines is done in several stages defining a new virtual machine on top of an old. A useful illustrative device is the 'onion diagram' due to Dijkstra where each new language or virtual machine is represented as a layer. The central core is the high level language presented with the machine which itself is built on layers terminating with the hardware implemented machine code.



In our system L-1 or B-0 is the given language, S-0 the final virtual machine and S-1 and L-0 the intermediate layers.

The situation we envisage is that the user has defined a new subject area by just such a structured definitions of primitives. He now wants to write programs in this system and then submit them to a compiler similar to the one defined earlier for sets.

Before going on to describe how our compiler can be generalised and then re-specialised to perform this function we will discuss the various types of semantic apparatus we envisage our compiler using.

5.2.1/

5.2.1 Semantic Apparatus

By definition the top-level languages used (e.g. S-0) will be close to the abstract area under discussion. This means, to our viewpoint, that the programming language and the semantic apparatus can be similar. In particular they can share the same primitives and the semantics can make use of the algebraic laws of these primitives. For example consider the semantic apparatus used in the transition S-1 to L-0 or B-0. Here our transformations are expressed in terms of state vectors whose values are expressions in S-1 primitives and the manipulation is done relative to algebraic laws over these primitives. In contrast lower machine based languages such as L-1 need a much more complicated semantic domain separate from the programming language.

Thus while we (the compiler writers) have to provide the semantic apparatus to adequately describe the lower languages the user can easily specify an adequate semantics for his defined languages as algebraic laws for the primitives.

5.2.2 The Generalised Transformational Compiler

We now describe what a generalised transformational compiler or rather our compiler generalised and then re-specialised to a specific area would look like.

Given that the basic languages are as before the core of our compiler would be the same as previously and the semantic apparatus could be built in.

For/

5.2.3 Example. A transformational compiler for a matrix system

To illustrate these points we will briefly describe how our transformational compiler might be set up to work for another domain. This section will be tentative but it may help to persuade the reader that our methods have wider application.

The domain that we have chosen is that of matrices and vectors. This domain is similar to that dealt with by APL but we wish to emphasise that we are not attempting to implement APL nor would we want to. APL, to our viewpoint, has many appropriate high level primitives but we feel that it is burdened with many other primitives that encourage the production of opaque programs. Also we feel that its control structures are of a low level; instead we would use the control structures of the language S-O, that is simple recursion equations.

We will assume a target language that allows matrix and vector definition and element by element manipulation of these objects.

The primitives of the top level language would include all those used in the normal informal mathematical discourse about matrices. For a very structured system we would perhaps want to implement these primitives first at a level which allows row and column selection and vector operations. However for this brief example we will assume that they are implemented directly in the target language.

Among/

Among the obvious primitives we would define are,

Matrix operations and constants.

For example

+, -, ×, / Element wise addition, subtraction, multiplication and division.

For example

$C = A-B$ where C, A, and B are all $m \times n$

iff $C(i,j) = A(i,j)-B(i,j)$ $i=1,2,\ldots,m$ $j=1,2,\ldots,n$

$\otimes$ Matrix multiplication

$C = A \otimes B$ where C is $m \times r$

A is $m \times n$

and B is $n \times r$

iff $C(i,j) = \sum_{k=1}^{n} A(i,k) \times B(k,j)$ $i=1,2,\ldots,m$ $j=1,2,\ldots,n$

I The identity matrices.

$A \otimes I = A$ and $I \otimes A = A$

Inv Matrix inversion

$Inv(A) \otimes A = I$ and $A \otimes Inv(A) = I$

Vector/

Vector operations

v+,v-,v×,v/ Element wise addition, subtraction, multiplication and division.

s× Scalar multiplication

V' = cs×V iff V'(i) = c×V(i) i=1,...,n

where V' and V are of length n.

In addition to these we could define the more unusual APL-like operations, such as the reduction operations or column and row extractions, as required.

These operations can be defined in the obvious way by iterative definitions in the lower primitives.

For example

A ⊗ B is

```
1->i; row(A)->ra; col(B)->cb; col(A)->ca;
  while i≤ra
     do [1->j; while j≤cb
                  do [1->k; 0->RES(i,j);
                      while k≤ca
                         do [A(i,k)×B(k,j)+RES(i,j)->RES(i,j);
                             1+k->k]
                         j+1->j]
                 i+1->i]
```

There/

There are obviously very many opportunities for linking while statements. Given the obvious definitions of the other primitives we would be able to produce a linking list thus.

[[+ - × / ⊗] [v+ v- v× v/ s×]]

However the greater structure of matrices as opposed to sets give us even greater opportunities for linking. We will return to this briefly in the example given below.

This area is also very rich in algebraic laws. We would be able to give our compiler a rule list thus

A×B = B×A, A+B = B+A, A+(B+C) = (A+B)+C etc.

A ⊗ (B ⊗ C) = (A ⊗ B) ⊗ C

Inv(A ⊗ B) = Inv(B) ⊗ Inv(A)

A ⊗ (B+C) = (A ⊗ B) + (A ⊗ C)

Inv(I) = I

A ⊗ I = A etc.

Inv(A) ⊗ A = I etc.

These things together with the linking schemas are all that the user would have to provide to produce a transformational compiler for a matrix language which would hopefully be an improvement on the usual implementations. Other features could be added, an obvious one being destructiveness, however, this feature is not so clearly separated as it was in the sets example. For example A×b->A has a meaning at the matrix/

matrix level and could be implemented destructively but we must be careful not to allow $\otimes$ to be implemented destructively as $A \otimes B \to A$ would then be calculated wrongly. $A \otimes B \to A$ can be implemented destructively using as temporary storage a vector of length row(A) so there is obviously room for investigation.

Example

We give a simple example, but even this small example is rich in optimising opportunities. Suppose we wish to discover if the powers of a matrix converge, a problem met repeatedly in Markov processes. The simple algorithm

$$\text{convg}(P) = \{T \to \text{convg1}(P,P)$$

$$\text{convg1}(P,Q) = \begin{cases} ||Q-Q\otimes P|| < \epsilon \longrightarrow Q \\ ||Q-Q\otimes P|| \geq \epsilon \longrightarrow \text{convg1}(P, Q\otimes P) \end{cases}$$

which converges to the stable matrix if there is such a matrix has an obvious iterative implementation.

```
P->Q;
while ||Q-Q⊗P|| ≥ ε
   do [Q⊗P->Q]
Q->Ans;
```

Two interesting points emerge on considering the further implementation of this program.

(i)/

(i) The two occurrences of $Q \otimes P$ obviously do not need to be separately computed however if we had inadvertently, but correctly, written one as $P \otimes Q$, our compiler would have been unable to spot this as $\otimes$ does not in general commute. This, we feel, reinforces what was said earlier about the need for an interactive capacity.

(ii) As mentioned earlier, there are richer opportunities for linkings. In the set domain no expressions could be linked if one was a sub-expression of the other. The greater structure of matrices allows us to relax this condition for many combinations of operators, at the expense of a more complicated linking list. In particular the - and $\otimes$ operations in $Q-Q \otimes P$ can be linked to give an implementation of $Q-Q \otimes P$ thus,

```
1->i; row(Q)->rq; col(Q)->cq;
while i≤rq
   do [1->j;while j≤cq
               do [1->k; 0->RES1(i,j);
                while k≤cq
                   do [Q(i,k)xP(k,j)+RES1(i,j)->RES1(i,j);
                       1+k->k]
                   Q(i,j)-RES1(i,j)->RES2(i,j); j+1->j]
               i+1->i]
```

Using this would enable us to make an improvement over the normal closed subroutine implementation for this simple algorithm.

REFERENCES

Ambler, A.P. and Burstall R.M. (1971) LIB POLYSETS. POP-2 Program Library Specification. Department of Machine Intelligence and Perception, University of Edinburgh.

Ashcroft, E. and Manna, Z. (1971) The translation of 'goto' programs to 'while' programs. Stanford Artificial Intelligence Project Memo AIM-138. Computer Science Department, Stanford University.

Berge, C. and Ghouila-Houri (1965) Programming, Games and Transportation Networks. Methuen.

Burstall, R.M. (1969) Proving properties of programs by structural induction. Computer Journal, Vol. 12, No. 1, February, 41-48.

Burstall, R.M., Collins, J.S. and Popplestone, R.J. (1971) Programming in POP-2. Edinburgh: Edinburgh University Press.

Cadiou, J.M. (1972) Recursive definitions of partial functions and their computations. Ph.D. thesis. Computer Science Department, Stanford University (to appear).

Cooper, D.C. (1966) The equivalence of certain computations. Computer Journal, Vol. 9, No. 1, May, 45-52.

Dijkstra, E.W. (1968) Goto statement considered harmful. C.A.C.M. Vol. 11, No. 3, 147-148.

Dijkstra/

Appendix II The Function Codefor

Here we give in full the function codefor described in section 3.2.2.7.1.1. Codefor is used to produce the code that will perform the evaluation of a given expression. Expressions that will be needed again later, indicated by their having more than one backpointer, are stored in temporary variables and remembered in donelist. Expressions that are to be linked together are formed into a compound item, a linked group, and expanded at a later stage.

codefor: expression×state×codelist×donelist×linkedlist->expression×codelist ×donelist

To produce a codelist to perform the evaluation of an expression exp starting in state with no expressions previously evaluated we call

codefor(exp,state,nil,nil,tobelinked)

codefor(exp,state,codelist,donelist,tobelinked) =

cases/

Appendix I The Matching Algorithm

Here we give in full the matching algorithm described in 3.1.3.2.1. We repeat the abstract structure that is assumed.

A circuit is a set of recursion equations.

A recursion equation has a left hand side and a right hand side.

A left hand side has a name and a list of variable names.

A right hand side is a set of rules.

A rule has a predicate tree and a tree.

We attempt to find a match for a circuit **c1** from amongst a set of circuit patterns sp by calling findmatchingcircuit(c1,sp,nil).

findmatchingcircuit: circuit$\times 2^{\text{circuit}} \times$m-list->m-list.

This function searches the set of circuits given as the second argument. It succeeds if it finds one that matches with the circuit in the first argument producing a list associating elements of the selected circuit with fragments of the input circuit.

```
findmatchingcircuit(circuit,posscircuitset,matchlist) =
        if isempty(posscircuitset) then fail close;
           let x = choose(posscircuitset);
              either matchcircuit(circuit,x,matchlist)
              orlast findmatchingcircuit(circuit,
                        posscircuitset/{x},matchlist)
              close
```

The/