FUNCTIONAL MODELS OF PROCEDURAL PROGRAMS

Mark Harman

Submitted in partial fulfilment of the requirements for award of Ph.D.

September 1992

The Polytechnic of North London

Declaration

(i) While registered as a candidate for Ph.D. I have not been registered or enrolled for another award of the CNAA or other academic or professional institution.

(ii) None of this material has been included in any other submission for an academic award.

Abstract

This thesis shows how *any* Procedural Programming notation may be modelled by a purely Functional notation and discusses the applications and implications of this modelling approach.

Existing ad hoc modelling techniques are gathered together within a common framework.

The thesis shows that these techniques break down when the state of computation for a procedural language is not an environment mapping from identifiers to denotable values.

A simple method for overcoming this difficulty is introduced, demonstrating that models may be constructed for all procedural programming notations.

The modelling approach allows the considerable body of functional reasoning techniques to be brought to bear in the analysis of procedural programs.

The thesis introduces a simple technique called "Abstraction Projection", with which the programmer may project a model onto a sub domain suitable for a particular analytic task.

Abstraction Projection removes from the model all details irrelevant to the computation of values within this sub domain.

The thesis also provides semantic definitions for the terms "Functional Language", "Procedural Language" and "Referential Transparency".

Keywords : Functional Programming, Procedural Programming, Denotational Semantics, Program-Proving, Program-Transformation, Referential Transparency, Axiomatic Method, Functional Models, Abstraction Projection.

Acknowledgements

I would like to thank my supervisor, Sebastian Danicic who, often in difficult circumstances, provided the perfect environment in which to carry out the research reported here. He often generalised my original ideas and his considerable knowledge and wisdom has guided me down many fruitful avenues.

I would like to thank Hugh Glaser for agreeing to be my second supervisor.

Professors John Darlington and Dan Simpson, who adjudicated this thesis have made a significant contribution by forcing my attention onto the specifics of the work carried out. This focus has produced a far more cogent account of the modelling strategy, for which I am very grateful.

I would like to thank Albert Langton for introducing me to mathematics.

I would like to thank Roger Bailey and Chris Hankin for introducing me to the practice and theory of functional programming.

I would also like to thank my parents, friends and colleagues for all their support and encouragement.

Contents

CHAPTER ONE

INTRODUCTION

CHAPTER ONE INTRODUCTION

Functional notations are considered superior to their procedural counterparts as a result of the rich algebraic properties that they enjoy.

This thesis describes a complete strategy for modelling procedural programs by equivalent functional programs, thus allowing a procedural programmer to apply functional reasoning techniques to the analysis of their programs.

1.1 Chapter Two: Procedural and Functional Notations

Chapter two sets out the functional notation used in this thesis (a small subset of ML) and briefly highlights the differences between the Procedural and Functional style. This chapter also describes, in outline, the essential semantic framework within which the modelling strategy is constructed.

1.2 Chapter Three: The Modelling Approach

The first author to describe a functional modelling technique was John McCarthy in [44,61]. McCarthy was concerned with describing the semantics of programming languages, his work was subsequently incorporated into the Denotational Description technique [16].

The first author to suggest that functional notation was an ideal notation with which to investigate the properties of procedural programs was James Morris in [79,80].

In [79,80,44,61] Morris and McCarthy describe several techniques for modelling various procedural programming constructs.

Together with well-known techniques, such as modelling loop constructs by tail recursion, there thus exists a set of as hoc approaches to modelling most common procedural programming constructs by functional equivalents.

§3.2 describes these techniques and shows that they break down when the state of computation is not an environment mapping. That is, when the state is not a mapping from the identifiers of the language to denotable values.

In §3.3 a simple technique for overcoming this difficulty is introduced. This technique involves the inclusion of extra identifiers with a consequent increase in the size of the denotable value space.

The inclusion of these extra identifiers allows the implicit part of the state to be modelled (within the procedural notation) by the explicit part of the state (the environment mapping).

Many authors, for example Stoy in [16], claim that the non-referential transparency of procedural languages is attributable to the presence of the assignment statement. §3.4 presents a brief polemic on the subject in which this claim is refuted.

1.3 Chapter Four: Abstraction Projection

The model for a procedural program includes in its result tuple all the semantic values computed by the program.

For large programs this is unacceptable.

Chapter four presents a simple technique called "Abstraction Projection" which allows the programmer to project a model onto a reduced domain.

This means that several, distinct models may be produced for one original procedural program, each projected onto a different domain.

Each of these models is a functional program that computes some part of the overall effect of the procedural program. All computation details which do not contribute to the computation of a model's result will be lost, thus providing the programmer with clarity by abstraction.

1.4 Chapter Five: Converting Models Back Into Procedural Notation

Chapter five describes some techniques for converting procedural models back into functional notation.

1.5 Chapter Six: Applications of Functional Modelling

Chapter six describes some applications of the modelling approach.

These are many and varied, since a model may be used to analyse and prove properties of *any* compile-time feature of a procedural program.

Several programs from textbooks are used to demonstrate the following applications of modelling:

Proof
Specification-Recovery
Error-Detection
Efficiency Improvements
Restructuring
Language Conversion

The examples are only small (the largest is about five pages). However, the modelling techniques described in chapters three, four and five can all be automated allowing the strategy to be applied to programs of arbitrary size.

1.6 Chapter Seven: Semantic Foundations

Chapter seven investigates the semantic foundations of the modelling strategy.

The advantage of a functional notation are to be found in its algebraic properties. Specifically, many authors attribute a property called "Referential Transparency" to functional programming notations. However, authors disagree on what precisely constitutes "Referential Transparency". In §7.2 a definition of "Referential Transparency" is given. The implications of this definition are investigated in §7.3.

§7.4 shows how a modelling strategy can be derived from and proved correct in terms of the semantic description of the programming language.

In many languages it is not possible to substitute two pieces of syntax which have equivalent meanings. This observation highlights a discrepancy between the syntax of a language and its semantics. §7.5 introduces a property called "Algebraic Closure". "Algebraic Closure" is enjoyed by any language in which a piece of syntax can be substituted for any other piece of syntax with equivalent meaning. The definition of Algebraic Closure may be used in a generative mode, allowing a language to be rendered "Algebraically Closed", and thus ironing out the irritating discrepancy which denies a programmer complete algebraic freedom.

The words "Functional Programming" and "Procedural Programming" have been used throughout this thesis (and, indeed, throughout the history of computing) without appeal to any definition of terms. This lack of rigor is addressed and rectified in §7.6 and §7.7.

§7.6 presents a definition of "Functional Language".

§7.7 presents a definition of "Procedural Language".

Finally, §7.8 looks at the issue of representing the interleaving of events in functional language, demonstrating that this information is not captured by a stream-based program in the style of Henderson [30].

1.7 Chapter Eight and Nine: Conclusions and Future Work

Chapter eight summarises the contribution of this thesis and chapter nine briefly lists directions for future work.

CHAPTER TWO

FUNCTIONAL AND PROCEDURAL LANGUAGES

2.1 Introduction

The chapter outlines the semantic issues which differentiate between procedural and functional languages.

The functional language, ML, is introduced in §2.2 and the algebraic rules of manipulation which apply to functional programs are listed in §2.3.

In §2.5 the Axiomatic method is briefly described and attention is drawn to the problems experienced by procedural-language programmers due to the algebraic inflexibility of procedural languages.

2.2 The Functional Notation, ML

There are many functional programming languages currently in use [41,51,52,20,39,37]. All of these languages are semantically very similar. The language ML has been used in this thesis since it seems to be the most popular [53].

Only a very small subset of the whole language ML is required for modelling purposes. This subset is described in appendix A1. The standard reference on the whole ML language is to be found in [41], however, it should not be necessary for the reader to consult this text in order to appreciate the functional modelling strategy described in this thesis.

2.2.1 A Brief ML Tutorial

What follows is a brief tutorial on the subset of ML used in this thesis.

2.2.1.1 Expressions

The basic forms of side-effect-free expressions permitted in all and any procedural notation are assumed to be available in the subset of ML used, together with the operators for forming new elements of these basic types from old ones.

2.2.1.2 Function Definition

In ML, a function definition is introduced with the keyword "**fun**". Thus:

```
fun times x y = x * y ;
```

describes the function which takes two parameters and returns the result of multiplying them together.

2.2.1.3 Function Application

The application of a function to its arguments is written by juxtaposition. Thus the application of the function "times", to the two arguments "2" and "3", would be written:

```
times 2 3
```

2.2.1.4 Tuples

In addition to the basic types of the language, compounds of types may be formed by enclosing several elements (possibly of differing type) in parenthesis. The result is the formation of an object called a tuple:

```
(1,true,'z')
```

The arity of a tuple is often prefixed to the beginning of the word tuple. For example the tuple just described is a "3-tuple".

2.2.1.5 Conditional Expressions

In a functional language, the conditional is an expression. It is written (in ML):

if E_1 then E_2 else E_3

Where E_1 must be a boolean expression, and E_2 and E_3 may be any expressions provided they have the same type.

The value of the conditional expression is E_2 if the result of E_1 is true and E_3 if it is false.

2.2.1.6 Lists

The following notation is used for lists:

"hd" for taking the head of a list and "tl" for the tail. The list construction function "cons" may be written as an infix operator "::". The empty list is written "nil", and the test for an empty list is written "null". A list of specified length may also be constructed by enclosing its elements in square brackets. The "append" function performs list concatenation.

2.2.1.7 Strong Typing

The model functions used in this thesis make use of untyped-lists. This is a generalisation of the ML list (which is typed).

It is *fortunate* that the functional notation *may* be executed, but from the point of view of functional models, it is only the conciseness and "algebraic" nature of the notation that is important.

The untyped lists used in models can always be replaced by completely typed versions with the introduction of some extra construction and selection functions for a "universal type".

2.2.1.8 Partial Application and Higher-Order Semantics

Consider the function "add", defined below:

```
fun add x y = x+y ;
```

This function takes two parameters and returns their sum. Using an ML interpreter, a console session might include the following (the machine's printing is shown in italics):

```
fun add x y  = x+y ;
add : int -> int -> int
add 23 5 ;
28 : int
add 1  (add 2 3) ;
6  : int
```

The machine responds to expressions by evaluating them.

When the programmer enters a function definition, the machine responds by deducing the type of the function, in this case integer to (a function from) integer to integer.

The type-inference aspect of ML is not an issue in this thesis and may be ignored.

A common slogan associated with functional programs is that functions are "first class citizens" [32]. This phrase is intended to convey the idea that a function can be treated in the same manner as integers, characters and other base-types. That is, functions may be the result of expressions and may be combined to form new functions using various operators over functions.

To achieve a paradigm for programming in which functions *are* treated in a similar manner to base types, functional languages allow their functions to be applied to fewer arguments than they actually require.

The result of such a "partial" application is, itself, a function (which requires the rest of its arguments before it can be "fully" evaluated). In [32], Turner demonstrates the dramatic increases in program brevity that may be achieved using partial application.

A simple illustration is the partial application of the addition function defined above. The console session described above could be continued as follows:

```
fun succ  = add 1 ;
succ : int -> int
```

The first line shows the programmer defining the successor function by partially applying the function "add" to only one of its two arguments.

The meaning of "add 1" can be viewed as the substitution of the expression, 1, for the argument, x, in the body of add, thus:

```
succ y ≡ 1+y
```

Applying a function to *none* of its arguments and applying a function to all of its arguments are simply special cases of partial application.

2.2.1.9 Higher-Order Functions

Applying a function to none of its arguments simply yields the function itself. This is how higher-order semantics are achieved in functional notations. Namely, a function "g", may take as its parameter a function. In order to supply a particular (named) function to "g", say "f", all that is required is to apply "f" to *none* of its parameters.

For example, a very useful function, usually called "map", is one which takes a function and a list and applies the function to each element in a list to give a new list. Consider the console session:

```
fun map f L = if null(L) then []
                         else f(head(L)) :: (map f tail(L)) ;
map : (alpha -> beta) -> list(alpha) -> list(beta)
fun listinc = map succ ;
listinc : list(int) -> list(int)
listinc [1,2,3,4,5]  ;
[2,3,4,5,6] : list(int)
```

2.2.1.10 Anonymous Functions

Any function may be supplied as an argument to a higher-order function, not just named functions and their partial applications.

This is achieved by having a notation for describing a function anonymously, using an expression which yields a function as its result. These functions are often called "Lambda Functions", in recognition of the fact that the Lambda notation provides just such a facility, and was the first "functional notation" [12,56].

In ML, a lambda function whose formal parameters are $x_1,\ldots,x_n$, and whose result is given by the expression E, is written:

```
fn x1 ,..., xn => E
```

Thus the definition of the successor function can also be written:

```
val succ = fn x => x+1 ;
```

2.2.1.11 Let Abstraction

The values of expressions which are to be "stored" for later use can be achieved using the let abstraction clause:

```
let val x = E in G ;
```

This is an expression, the value of which is found by evaluating G with all occurrences of "x" replaced by "E".

The let abstraction construct is thus merely syntactic sugar [64] for:

```
(fn x => G) E
```

The let abstraction construct may also be used with a tuple of identifiers, $x_1,\ldots,x_n$ and a tuple-valued expression, E:

let val $(x_1,\ldots,x_n)$ = E in G ;

The let abstraction construct can also be written as a "where" abstraction:

let val x = E in G ≡ G where x = E

2.2.1.12 Sequencing

In a functional notation the only sequencing of expression-evaluation is that which is implicitly demanded by the data-dependency in the function-call hierarchy.

In particular, there is no notion of the procedural statement *sequence*.

2.2.1.13 Henderson's Lazy Streams

Because the programmer has no ability to explicitly demand a certain execution sequence, representing Input and Output sequencing is a particular problem for a functional style.

One solution to this problem, suggested by Henderson in [30], is to write a functional program as a function, "f", from lists to lists. The result of "f" (a list) is the output, and the input (also a list) is provided to "f" as the value of its actual parameter.

In this thesis, functional programs are used as models of procedural programs *only* in order to investigate properties of these programs. Input and output sequencing does not normally form part of such investigations and so the problem of representing it does not normally arise.

In chapter three, a general modelling strategy is described. A modelling strategy can *always* be constructed from the semantics of the programming language to be modelled. This means that it *is* possible to model input and output sequencing in a perfectly natural way *if* it is described in the semantics of the language (see, for example §7.8).

2.3 The Fold/Unfold Transformation Rules

There are well-established rules of algebraic manipulation for functional languages. These rules are collectively known as the "Fold/Unfold Methodology" [22].

The rules allow new functions to be created and existing functions to be altered, whilst *guaranteeing* that the whole functional program will still compute the *same* result. These rules are used extensively throughout this thesis, and form one of the main motivations for the modelling approach.

The six transformation rules are listed below. The notation E[a/b] represents the expression formed by substituting all occurrences of b in the expression E, by a.

(1) Fold

Let f(x1,...,xn) = E and g(x1,...,xm) = E′ be the equations of two functions f and g.

If E′ contains an expression E[t1/x1,...,tn/xn] where t1,...,tn are n terms, then E′ can be written E′[f(t1,...,tn)/E].

(2) Unfold

Let f(x1,...,xn) = E and g(x1,...,xm) = E′ be the equations of two functions f and g, where E′ contains a function application f(a1,...,an′) (n′≤n). The equation for g can be replaced by g(x1,...,xm) = E′[A/f(a1,...,an′)] where A = E[ai/xi].

(3) Instantiation

Let f(x1,...,xn) = E, and let c1,...,cm be constants (m≤n), then some (or all) of the free variables (x1,...,xn) in E can be replaced by c1,...,cm (assuming that the ci's have identical types to the xi's). In other words the equation for f can be replaced by f(x1,...,xn)[ci/xi] = E[ci/xi].

(4) Abstraction

An expression E containing sub expressions A1,...,An can be replaced by the expression: **let val** (φ1,...,φn) = (A1,...,An) **in** E[φi/Ai].

(5) Definition

A new function equation can be introduced providing the left hand side of the equation does not already exist in the program. Thus either a completely new function is introduced, or the domain of an existing function is extended.

(6) Algebraic Laws

The algebraic properties of the equation used in the functional program can be exploited. Although this may seem to be obvious, it is, in fact, a unique advantage of declarative programming over procedural programming. In a procedural program it is not even possible to replace the occurrence of the expression e1 + e2 by the expression e2 + e1, thus exploiting the law of commutativity of addition. This is because the semantics of the language *does not* guarantee that the symbol "+" has the same algebraic properties as the mathematical operator "+". In particular side effects in the expression e1 may cause these properties to be denied.

2.3.1 Partial Correctness

The Fold/Unfold transformation rules are partially correct. That is, although programs produced by transformation are guaranteed not to produce a different result to their untransformed versions, there is no guarantee that the transformed functions will terminate.

It is quite easy to see how the method can introduce non-termination into a terminating program:

```
fun f(x) = x ;
fun f(x) = f(x) ;
                                                    folding
```

2.3.2 Completeness

There are many equivalent sets of recursion equations which cannot be shown to be equal by transformation, however in these cases inductive proof techniques can be employed to demonstrate equivalence. In this sense an extra rule can be added to the six transformation rules above:

(7) Induction

Let f(x1,...,xn) = E. If there is an inductive proof that the function g(x1,...,xn) = E′ is equivalent to f then the function f(x1,...,xn) = E′ can be introduced into the functional program.

The inductive technique used in this thesis is called "structural induction" (It was introduced by Burstall in [23]). Many other proof techniques exist (see, for example [50,57,60,61,85,86]).

There has been considerable debate as to whether or not it is possible and/or desirable to construct proofs for large-scale programs. The main arguments for either side of the debate can be found in [43,46,47,48,49].

2.4 Programming Language Semantics

The unifying framework which binds all programming languages together is the "state of a computation".

The next section describes the concept of the state of computation for procedural and functional languages.

2.4.1 The State Of Computation

The state of computation is an *abstract* representation of the physical configuration of the machine executing the program.

The state is abstract in the sense that it need only describe those features of the machine's state which can effect the *meaning* of a program. The precise construction of the state therefore depends upon what the meaning of a program is. This is a *choice* made by the language designer.

Considerable work has been conducted on the meaning of programming languages, much of which has been seminal to the work reported in this thesis. References to some of the established results are summarised below:

The origins of the state-concept can be found in [9,10,11,26,44,61].

The mathematical aspects of semantics are described in [13,15,55,59,76].

Programming Language Semantics are described in [16,36,57,82].

Applications of and extensions to the basic theory can be found in [17,35,45,81,84].

A semantic description of a program is called a "direct" semantics, if the meaning that it gives a program is a mapping from initial state to final state. This thesis is conerned with procedural programs for which a direct semantics can be defined.

Functional language Semantics are characterised by expression evaluation within a (highly restricted) state called an "environment".

In §7.6 and §7.7 these remarks are made completely precise. In the next section a more informal approach is taken, highlighting the essential difference between the procedural and functional style and laying the conceptual foundations for the modelling strategy described in the next chapter.

2.4.2 The State Of A Procedural Program

Each statement in a procedural program is described as a modification to the state in which it is executed. The meaning of an entire statement-sequence is simply the composition of the individual state-changes which describe each component statement.

2.4.3 The Environment

For any procedural language in which identifiers may be assigned values, the state will include an "environment".

The environment is a function which maps identifiers (in the domain I) to the values (in the domain V). The domain, I, is the set of all identifiers in the syntax of the language. The domain, V, is the domain of all values which may be bound to these identifiers. V is termed the "denotable values" of the language [16].

The environment represents, in an abstract manner, the values stored in the memory of the computer.

For example, an environment which represents the fact that the identifier "x" is bound to 1, "y" to 2, and all the other variables are unassigned could be written (in ML):

```
fun Environment identifier = if identifier = "x" then 1
                             else if identifier = "y" then 2
                             else unassigned ;
Environment : I → V ;
```

2.4.4 The Meaning "Unassigned" and Abstraction

"*unassigned*" is simply a member of V, which describes the value of any and all variables which have not been assigned a value.

The use of "*unassigned*" is an example of abstraction in the semantic description: Unassigned variables on a computer will be bound to "rubbish values" (whatever happens to be in the machine's memory at the time). However these machine-dependant features are deliberately overlooked, all unassigned identifier names being mapped to the same value by the environment. The environment described here is thus sufficiently abstract that it can describe the state of execution upon *any* computing device (including a trained human with pen an paper).

2.4.5 The Meaning of a Statement Sequence

The meaning (or abstract effect of computation) of the statement sequence:

```
x := 1 ; y := 2
```

can be described as a mapping from some initial environment, ρ, to a final environment, which adds to ρ the bindings for "x" and "y" (overwriting any existing binding for these two identifiers in ρ) :

```
fun M ρ identifier = if identifier = "x" then 1
                     else if identifier = "y" then 2
                     else ρ(identifier) ;
```

$M : (I \rightarrow V) \rightarrow (I \rightarrow V)$

2.4.6 The Implicit State

There exist procedural language statements that create changes other than to values in the store. For example, consider the statement: "Mode n", which alters the screen mode of the display device.

Imagine a screen with four modes, numbered zero through to three.

In order to describe the meaning of a program in this language, the state must now be extended from the environment I→V, to a cartesian product, the first component of which is the environment mapping and the second component of which is a number between zero and three, describing the screen-mode.

Using this extended computation-state, the statement sequence:

```
Mode 2 ; Fred := 1
```

will be described by the state mapping M, below:

```
fun M (ρ,μ)     =     (fn z => if z="Fred" then 1 else ρ(z),  2) ;
```

$M : ((I \rightarrow V) \times \text{int}) \rightarrow ((I \rightarrow V) \times \text{int})$

As more procedural features are considered, there is a consequent increase in the size of the state required to describe the language.

For example, describing the effect of input and output statements requires a description of the state of the corresponding devices.

Describing aliasing of identifier names (two names for the same value) requires a more detailed description of the heap store.

2.4.7 The State Of A Functional Language

For a functional language, there are no statements; there are only expression constructs. The meaning of an expression is the value produced by expression-evaluation. This value may *only* depend upon the bindings for the free variables in the expression, and thus the state for a functional language is simply the environment mapping (I→V), which defines the meaning of the free variables.

As more expression constructs are considered the size of the *value space*, V, may increase, but the *state itself* remains simply an environment mapping from (I→V).

2.4.8 Reasoning About Procedural Programs

In §2.3, a powerful reasoning technique for functional programs was described. Using this technique, a programmer can regard their programs as algebraic expressions, performing simple meaning-preserving manipulations upon them.

In procedural languages, it is possible to prove properties of programs using a technique called the "Axiomatic Method", but algebraic manipulation rules, where such rules exist, will be highly complicated as a consequence of the complexity of the computation-state.

2.5 The Axiomatic Method

The proof technique most widely applied to procedural programs is the Axiomatic Method. The technique is also called the "Weakest Precondition" method by Dijkstra in [43].

Assertions, written in the Predicate Calculus, are inserted in between the statements of a program. The free variables in these assertions are simply the identifiers used in the program. This creates a connection between the values computed by the program and the truth-value of assertions.

Each statement of the programming language can be described as an assertion-mapping. The particular mapping for a statement defines the assertion which will hold after the statement has been executed, in terms of the assertion that held prior to its execution.

This approach was first suggested by Alan Turing in 1949 [4], but received little attention until 1967 when Floyd suggested the idea as a means of describing the semantics of a language[6]. (the idea was also put forward in 1966 by Peter Naur[5]).

In 1969, Hoare was the first author to develop a calculus for program proving, using insertion of Predicate Calculus assertions. Hoare extended the approach to include most commonly used language features, creating what is now known as the "Axiomatic Method" [7,8,33,34,42].

Consider a small subset of Pascal, the syntax of which is described below:

Syntax

⟨Statement⟩ ::= **skip** ¦ x := e ¦ S_1 ; S_2 ¦ **if** e **then** S_1 **else** S_2 ¦
while b **do** S **od**

The following rules may be used to modify assertions placed between statements:

(1) **Assignment Axiom**

$$\{p[e/x]\}\ x := e\ \{p\}$$

(2) **Composition Rule**

$$\frac{\{p\}\ S_1\ \{q\} \quad \wedge \quad \{q\}\ S_2\ \{r\}}{\{p\}\ S_1\ ;\ S_2\ \{r\}}$$

(3) **Conditional Rule**

$$\frac{\{p \wedge e\}\ S_1\ \{q\} \quad \wedge \quad \{p \wedge \sim e\}\ S_2\ \{q\}}{\{p\}\ \textbf{if}\ e\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

(4) **While Loop Rule**

$$\frac{\{p \wedge e\}\ S\ \{p\}}{\{p\}\ \textbf{while}\ e\ \textbf{do}\ S\ \textbf{od}\ \{p \wedge \sim e\}}$$

(5) **Skip Axiom**

$$\{p\}\ \textbf{skip}\ \{p\}$$

(6) **Consequence Rule**

$$\frac{\{(p \Rightarrow p') \wedge p'\}\ S\ \{q' \wedge (q' \Rightarrow q)\}}{\{p\}\ S\ \{q\}}$$

2.5.1 An Example Proof Using the Axiomatic Method

The following procedural program calculates the quotient and remainder produced by the division of two numbers. The input to the program is the two values "x" and "y" and the output is the quotient "a" and remainder "b" of "x" divided by "y".

```
a := 0 ;
b := x ;
while b ≥ y do b := b - y ; a := a + 1 od
```

In order to prove that the correct values reside in "a" and "b" after execution, the assertion "ay+b=x ∧ b≥0 ∧ b<y" must be shown to hold. It is necessary to add the conjunction "b≥0 ∧ b<y" to ensure that the solution is not "degenerate", for instance, without this qualifier "a=0, b=x", would satisfy "ay+b=x" without necessarily entailing that "a" is the quotient and "b" the remainder.

Let the entire program be named S. An Axiomatic Proof, taken from [8], is presented below:

$$\{x \geq 0 \wedge y \geq 0\}\ S\ \{ay+b=x \wedge 0 \leq b < y\}$$

by the axiom of assignment :

$\{0*y+x=x \wedge x \geq 0\}\ a := 0\ \{ay+x=x \wedge x \geq 0\}$ (1)

also by the axiom of assignment

$\{ay+x=x \wedge x \geq 0\}\ b := x\ \{ay+b=x \wedge b \geq 0\}$ (2)

combining (1) and (2) using the composition rule gives (3)

$\{0*y+x=x \wedge x \geq 0\}\ a := 0;\ b := x\ \{ay+b=x \wedge b \geq 0\}$ (3)

by the consequence rule :

since $x \geq 0 \wedge y \geq 0 \Rightarrow (0*y+x=x \wedge x \geq 0)$ assertion (3) can be rewritten :

$\{x \geq 0 \wedge y \geq 0\}\ a := 0;\ b := x\ \{ay+b=x \wedge b \geq 0\}$ (4)

Now considering the body of the while loop :

by the axiom of assignment :

$\{(a+1)y+b-y=x \wedge b-y \geq 0\}\ b := b-y\ \{(a+1)y+b=x \wedge b \geq 0\}$ (5)

also by the axiom of assignment :

$\{(a+1)y+b=x \wedge b \geq 0\}\ a := a+1\ \{ay+b=x \wedge b \geq 0\}$ (6)

As before, the composition rule allows (5) and (6) to be combined :

$\{(a+1)y+b-y=x \wedge b-y \geq 0\}\ b:= b-y;\ a:=a+1\ \{ay+b=x \wedge b \geq 0\}$ (7)

by the consequence rule :

since $(ay+b=x \wedge b \geq 0 \wedge b \geq y) \Rightarrow (a+1)y+b-y=x \wedge b-y \geq 0$

assertion (7) can be rewritten :

$\{ay+b=x \wedge b \geq 0 \wedge b \geq y\}\ b:= b-y;\ a:= a+1\ \{ay+b=x \wedge b \geq 0\}$ (8)

(This is the while loop invariant)

The while loop rule allows the invariant (8) to be combined with the while loop predicate "b≥y" giving (9), below:

$$\{ay+b=x \wedge b\geq 0\}\ \textbf{while}\ b\geq y\ \textbf{do}\ b:=b-y;\ a:=a+1\ \textbf{od}\ \{ay+b=x \wedge 0\leq b<y\} \quad (9)$$

Finally combining (9) with (4) gives the required proof:

$$\{x\geq 0 \wedge y\geq 0\}$$
$$a := 0;\ b := x\ ;\ \textbf{while}\ b\geq y\ \textbf{do}\ b:=b-y;\ a:=a+1\ \textbf{od}$$
$$\{ay+b=x \wedge 0\leq b<y\}$$

2.6 Manipulation Of Procedural Programs

Using the Axiomatic Method, the assertions written by the programmer in Predicate Calculus notation "sit on top of" the procedural notation. It is the assertion-notation that is manipulated and not the procedural notation in which the program is written.

In a functional notation, the notation in which the program is written is also the notation in which manipulations are performed and proofs constructed.

On reason why proofs using the Axiomatic Method become drawn-out, appears to be procedural notations' lack of "algebraic flexibility".

A comparison of the Axiomatic Method and the Functional Modelling approach can be found in §6.5.

2.7 Summary

This chapter has introduced the concept of functional and procedural programming styles. The chapter raises the following issues, which shall be taken up in the next chapter:

Proof techniques exist for both procedural and functional notations.

Proof techniques for functional notations use the functional notation itself. Programs written in a functional style can be manipulated algebraically using rules of transformation (the fold/unfold rules).

The procedural proof technique (the Axiomatic Method) requires the introduction of extra notation (in the Predicate Calculus), in which the proof is constructed.

Programs written in a procedural notation are algebraically inflexible. Functional language semantics can be described using a state which only requires a mapping from identifiers to values. Such a state is called an environment.

Procedural Language semantics require a state which includes many other components in addition to the environment.

CHAPTER THREE

THE MODELLING STRATEGY

CHAPTER THREE THE MODELLING STRATEGY

3.1 Introduction

This chapter introduces the functional modelling approach as a means for analysing procedural programs.

A variety of techniques exist in print [79,80,44,61] and in the "folklore" of functional programming, that will allow procedural programming features to be compiled into equivalent functional programming features.

In §3.2 these existing techniques are introduced in the common framework of a procedural language whose computation-state is simply identifiers to denotable values (I→V).

In §3.3 it is shown that the existing techniques break down when the state of computation contains extra components other than (I→V). A simple technique for overcoming this difficulty is introduced.

§3.4 sets out a brief polemic concerning referential transparency and assignment. Specifically it is argued that assignment is *not* a referentially-opaque construct.

3.2 Existing Techniques For Modelling

In §3.2 the state of computation is assumed to be solely an environment mapping. In §3.3 this restriction is relaxed.

3.2.1 What is a Functional Model?

The meaning of a procedural program is a state mapping (this thesis is only concerned with programs for which a direct semantics is possible)

The meaning of a procedural program, P, is thus a function which takes an initial state, S, and produces a final state, S′.

If this state is simply a mapping from the identifiers in the program, I, to some arbitrary value domain, V, then the program can be considered to be a prescription for modifying some number of the identifier bindings in S, in order to create S′.

Since a statement-sequence can only ever affect a *finite* number of these bindings, it can be modelled by an *expression* which defines a set of values. The values of the "affected variables".

The final values of these affected variables will *depend* upon the initial state, S, in which the statement sequence is executed. However, these final values will also only depend upon a *finite* number of the bindings (in S).

Thus, for a statement sequence, defined by a mapping (I→V) → (I→V), there exists a finite set of affected variables, and a finite set of values needed to compute the final value of these affected variables.

The "needed values" are simply the initial bindings (in S) for some of the variables used in the statement sequence.

A model for such a statement-sequence is simply a function, which takes as its argument, a tuple of initial values for needed variables, and returns, as its result, a tuple of final values of affected variables.

This section explores some standard techniques for modelling procedural programming constructs.

These techniques *only* apply where the state of computation is an environment mapping.

3.2.2 Modelling Assignment

The archetype of a procedural language is the assignment statement.

It is possible to model assignment by let abstraction. All that is required, is to ensure that the scope of the let abstracted identifier, exists only up until any re-assignment to the variable. That is, the *scope* of the let abstraction must correspond to the *extent* [16] of the assignment.

A nice illustration of this approach (and the benefits that accrue from modelling) is given by the following example.

Consider the sequence of assignments below:

```
x := x + y ;
y := x - y ;
x := x - y
```

The "needed variables" of this sequence of statements are the identifiers "x" and "y".

The "affected variables" are also "x" and "y".

Thus the model is:

```
fun f(x,y) = let val x  = x + y   in
             let val y  = x - y   in
             let val x  = x - y   in  (x ,y) ;
```

This function can be manipulated (unfolding "x" and "y" in the result tuple) to remove the let abstractions:

```
fun f(x,y) = ( ((x+y)-((x+y)-y)), ((x+y) - y) )
```

Which can be manipulated (using simple arithmetic properties) to:

```
fun f(x,y) = (y,x)
```

Thus, the effect of the three assignment statements is revealed: that of swapping the contents of the variables "x" and "y". (However, the proof relied upon the laws of arithmetic, thus "x" and "y" contain numeric data). This swapping technique was used in the past when there was a high premium on storage use. Nowadays, a programmer would be more likely to use the more familiar technique involving a temporary variable. One benefit of functional modelling, is that these two techniques could be proved equivalent (for numeric data), simply by unfolding model functions.

3.2.3 Modelling Conditionals

In providing functional models, there are often several choices as to the strategy used to model a particular statement.

Of course, one model will be convertible to any other alternative, using functional reasoning.

Here are two methods for modelling conditional statements using conditional expressions:

3.2.3.1 Modelling Conditionals In Context

A conditional may be modelled by appending the statement sequence after the conditional to the end of each of the statement-sequences for the "then" and "else" branches. Clearly, this strategy is inappropriate if a large amount of copying is required.

if e then s_1 else s_2 endif; s_3
⇒ if e then s_1; s_3 else s_2; s_3 endif

⇒ if e then e_1 else e_2

Where e_1 and e_2 are the results of transforming s_1; s_3 and s_2; s_3 respectively.

3.2.3.2 Modelling Conditionals In Isolation

Alternatively, by forming the tuple of all the variables affected by either the **"then"** and/or **"else"** branches, it is possible to form the l.h.s. of a let abstraction, defining the new values for the variables in terms of a conditional expression.

if e then s_1 else s_2
⇒ let val $(x_1, x_2 \ldots x_n)$ = (if e then e_1 else e_2)

where x_1 ... x_n are the affected variables of the statements s_1 and s_2, and e_1 and e_2 are models for the statements s_1 and s_2 respectively.

3.2.3.3 An Example of a Model for a Conditional Statement

```
if x > 10 then a := x else b := x
```

Can be modelled by the let abstraction:

```
let val (a,b) = if x > 10 then (x,b) else (a,x) in ...
```

Where the "..." is the model for the rest of the program after the conditional statement.

3.2.4 Modelling Iteration

Procedural languages typically provide several repetitive constructs. In each, a condition controls the repeated execution of a sequence of statements. It is well known that all these constructs can be converted into a **while** loop which, in turn, can be modelled by a recursive procedure:

```
while b do s ≡ procedure P ;
               begin
               if  b
                   then
                   begin
                   s;
                   P
                   end
               end
```

3.2.5 Modelling Procedures

In order to model a any procedure, it is necessary to collect together all the effects the procedure may have upon the values of variables.

There are two types of variable affected by a procedure call: those passed as call-by-reference parameters and those global variables assigned a new value in the body of the procedure. The two can be both treated in the same manner by re-casting all global variable assignments within a procedure body to call-by-reference parameter assignments.

For example, if "x" is a global variable then the procedure "p"

```
procedure p;
begin
x := e
end
```

can be recast:

```
procedure p'( var x : integer ) ;
begin
x := e
end
```

so long as calls to "p" are recast as calls to "p'", with the global variable "x" passed as actual parameter.

3.2.5.1 Modelling Call-By-Reference Parameter Passing

The approach adopted by Morris in [79] for modelling procedures with call-by-reference parameters is to use a function which returns the final values of the call-by-reference parameters in a tuple.

3.2.6 A Complete Example

All the above modelling techniques can be used together in modelling the procedure "locate", below:

```
type SomeArray = array [1..1000] of integer ;
procedure locate( A              : SomeArray ;
                  Element        : integer   ;
                  var Location   : integer   ;
                  var Found      : integer   ) ;
var i : integer ;
begin
    i := 0 ;
    while (i<=1000)  and  (A[i]<>Element) do i := i + 1 ;
    if i <> 1001 then begin
                        Found := true;
                        Location := i
                        end
                 else Found := false
end
```

The **while** loop, within the procedure "locate", is modelled (in the procedural notation) by the recursive procedure, "loop", below:

```
procedure loop ;
begin
   if i <= 1000 then
                if A[i] <> Element then begin i := i + 1 ;
                                              loop
                                        end
end
```

The conditional has to be nested in order to model the effect of short-circuit evaluation of the boolean expression in the **while** loop.

The global variables affected by the procedure "loop" can be written as a Call-By-Reference parameter, giving:

```
procedure loop( var i : integer ) ;
begin
  if i <= 1000 then
               if A[i] <> Element then begin i := i + 1 ;
                                             loop
                                       end
end ;

procedure locate( A              : SomeArray ;
                  Element        : integer   ;
                  var Location   : integer   ;
                  var Found      : integer   ) ;

var i : integer ;
begin
   i := 0 ;
   loop(i) ;
   if i <> 1001 then begin
                        Found := true;
                        Location := i
                        end
                else Found := false
end
```

Using a strategy also employed by Morris in [79], it is possible to model an array, using a function from array-indexes to values stored in the array.

Thus the procedures "loop" and "locate", can be modelled in ML, by the functions "loop" and "locate":

```
fun loop(A, i, Element) =
  if i <= 1000 then if A(i) <> Element then loop(A, i+1, Element)
                                       else i
               else i ;

fun locate(A, Element, Location) =
  let val i = loop(0) in
    if i <> 1001 then (i,true)
                 else (Location,false) ;
```

3.2.7 Modelling Aliasing

It is possible to create an alias using call-by-reference parameter passing. An alias is a name for a value which already has a name.

When a procedure, P, with two variable parameters is called "P(z,z)", then an alias is created: within the body of P, there will be two names for the global variable "z".

For a procedure-call which creates an alias, the semantics will typically be written with a two-level store [16], and so will not be in the domain (I→V).

In §3.3 a general method is described, which allows modelling of programs which exhibit such semantics.

3.2.8 Modelling Goto Statements

"**goto**" statements can be modelled by breaking the procedural program up into sequences of statements which are not labelled, and modelling these as parameterless procedures. The "**goto**" statements, themselves, are modelled by a call to the corresponding procedure.

This strategy is based on a method first used by McCarthy in [44,61].

Consider the program below:

```
     x := 0 ;
     n := N ;
L1 : if n = 0 then goto L2 ;
     x := x + n ;
     n := n -1
     goto L1 ;
L2 : x := x * 10 ;
     Result := x
```

The program consists of three blocks: the two statements before "L1", the four statements between "L1" and "L2" and the final two statements.

```
        x := 0 ;
        n := N ;
```

```
  L1 : if n = 0 then goto L2 ;
       x := x + n ;
       n := n -1
       goto L1 ;
```

```
  L2 : x := x * 10 ;
       Result := x
```

Each block can be modelled by a procedure:

```
procedure L0 ;
begin
   x := 0 ;
   n := N ; L2
end ;
```

```
procedure L1 ;
begin
   if n = 0 then L2 else begin
                         x := x + n ;
                         n := n - 1;
                         L1
                         end
end ;
```

```
procedure L2 ;
begin
   x := x * 10 ;
   Result := x
end
```

These parameterless procedures can be modelled by functions according to the strategy outlined earlier in §3.2.

In the model below, the function, "L0", takes the initial value of "N" and returns the final value of the affected variables; "x", "n" and "Result":

```
fun L0(N) = L1(0,N) ;

fun L1(x,n) = if n=0 then L2(x,n) else L1(x+n,n-1) ;

fun L2(x) = (x,n,x*10) ;
```

3.2.9 Modelling Calculated Goto Statements

If the label used by a goto statement may *only* be calculated at the time the statement is *executed* then the statement is called a "calculated goto".

Some procedural programs contain goto statements, which cause execution to jump into the body of a procedure.

Programs which make use of either of these features may require a continuation-based semantics [16,45]. Such programs *can* be modelled, but the strategies that may be used lead to models which effectively *interpret* the program that they model, and as such are not particularly well-suited to manipulation.

3.2.10 Program Semantics verses Language Semantics

An important feature of the modelling approach, is that it is concerned with *individual programs* and not *languages*.

The semantics of a *language* has to be constructed in such a way as to prescribe a meaning for *every* program in the language. The denotation for a *particular program* (derived from the semantics of the language) is thus defined over a state which must account for the *most* semantically intricate program.

The semantics of a particular program, *may*, however, only require a highly simple semantic description, even if it *were* to be written in a language which allows for highly intricate semantics.

Thus, a language may require a continuation semantics due to the inclusion of calculated goto statements, but a program which does not use a calculated goto statement can still be described *without* recourse to continuation semantics.

3.3 The Implicit State: A Problem for Existing Techniques

Using the standard techniques described in §3.2, a model may be created for any program which can be described by a mapping from (I→V) to (I→V).

In order to model any language construct one simply has to consider the effect of the statement on the bindings in the environment.

However, procedural language semantic-descriptions typically require a far larger state than simply (I→V) in order to describe other effects of a program's execution, such as changes to the input and output streams, the state of the heap store, the screen-mode and so on.

In this thesis the term *implicit state* is used to describe all those components of the state which are not in (I→V). The *explicit state* (or environment as Stoy calls it in [16] and elsewhere) is simply the domain (I→V). It is "explicit" in the sense that, for all its components, there exists an explicit piece of program *syntax* (the identifier), which is bound (by the environment mapping), to a *semantic* value.

The requirement for an implicit state in a program's semantics, makes the modelling techniques described in §3.2 invalid, as the following example demonstrates:

```
program P ;
var x : integer ;
begin
    x := 1 ;
    write('Hello, world')
end.
```

The model for this program (using the techniques described in §3.2) would be simply the number 1, describing the value residing in the variable "x" after execution of the program and reflecting the change to the explicit state.

Unfortunately, in addition to assigning 1 to "x", changing the *explicit* state, the program also affects the output device, which is part of the *implicit* state. This implicit effect is not modelled.

Were there to be *no* assignment statements at all, then the program would cause *no* change to the explicit state and would thus have *no model*.

3.3.1 The Output List: An Implicit State

The state, S, for the program, P above, is defined by Gordon, in [36], as:

$$S = (I \rightarrow V) \times V^*$$

Where V^*, is a sequence of values drawn from V. V^* is the implicit state, it represents the sequence of values on the output device.

3.3.2 Modelling the Implicit State

Clearly, the solution to modelling a program whose state is not in the domain (I→V), is to alter the description of the state so that it uses the domain (I→V).

This can be achieved in a very straight forward manner:

A new, unused, identifier is introduced into the procedural program. This new identifier is bound to the value of implicit state, thus making it explicit. Where a program affects the implicit state, this can be modelled in the procedural notation as an assignment to the new identifier. Modelling implicit effects as explicit assignment statements is simply modelling of one procedural program (with implicit effects) by another (without implicit effects).

Of course allowing the value of the implicit state to be bound to an identifier, may lead to an extension of the (denotable) value space, V. However, this appears to present no problems.

In practice, it may be more convenient to introduce a separate identifier for *each* component of the implicit state. To illustrate the modelling of the implicit state, two examples of implicit-state semantics are now described: Input/Output and Heap-Store usage.

3.3.3 Modelling Input and Output

Given that input and output are defined as semantic operations upon lists (in a standard manner described by Gordon in [36]), then this implicit state, and the effects upon it can be modelled as follows:

Two extra identifiers are introduced: "Input" and "Output", making explicit the input list and output list state-components.

The affect of **read** and **write** statements upon the input list and output list will be (procedurally) modelled by assignments to "Input" and "Output" as follows:

read(x)
is procedurally modelled by
x := hd(Input) ; Input := tl(Input)

write(E)
is procedurally modelled by
Output := append(Output,[E])

Before the program is executed, no output will have been produced, so the output list will initially be assigned a value "nil".

Thus, a statement sequence consisting of **read** and **write** statements is modelled in the procedural notation by a sequence of assignment statements.

The state will then be the domain (I→V), which can be modelled in the functional notation by the standard techniques described in §3.2

3.3.4 An Example of Input/Output Modelling

Consider the statements:

```
read(x) ; write(x+1)
```

These can be modelled procedurally, by the statement-sequence:

```
Output := nil                    ;
x      := hd(Input)              ;
Input  := tl(Input)              ;
Output := append(Output,[x+1]) ;
```

These statements can be modelled by a function from needed variables to affected variables in the manner described in §3.2:

```
fun f(Input)    =    (tl(Input), hd(Input)+1) ;
```

3.3.5 Modelling Interleaving of Input and Output

Consider the two programs:

```
program P1 ;
var x  : integer ;
begin
     read(x) ; write(1)
end.
```

```
program P2 ;
var x  : integer ;
begin
     write(1) ; read(x)
end.
```

The models for these two programs are identical:

```
fun f(Input)  =  (tl(Input), 1) ;
```

However, they do not *behave* identically when executed: execution of P1 will not cause any output to appear if no input is received, whereas execution of P2 will always cause output, regardless of whether or not any input is ever received.

3.3.6 The Semantics of the Program Defines the Model

The fact that the modelling approach used here does not represent the interleaving of input and output events is a direct consequence of the fact that the semantics of the *language* (as described in [36]) does not describe the interleaving of input and output events.

It is a matter of choice, when describing the semantics of a language, as to what features of a program's execution are significant enough to form part of the semantic description. In §7.8, a different semantic description for Input and Output is used to construct a modelling strategy. For this semantic description the interleaving of input and output events forms part of the state of a computation and, thus, interleaving *is* represented in the model.

3.3.7 Programs Which Use the Heap

The heap store is another example of an implicit state in programming language semantics.

Programs which use a heap may be described by a state which is formed from the cartesian product of three values:

$$(I \rightarrow V) \times A \times (A \rightarrow V)$$

Where "A" is the domain of addresses and "A→V" is an abstract representation of the heap. The second component of the state, "A", records the address of the top of the heap.

The semantic domain, V, will now include an extra value, *Rubbish*, the initial contents of the heap.

3.3.7.2 Modelling the Heap Store

In this exposition, the domain of integers will be used for addresses. To model the heap, two identifiers are introduced:

"H", the heap function, a mapping from A→V
and
"hp", the variable which contains the address of the top of the heap.

Statements which affect the heap are modelled in Pascal by assignment to these two identifiers, thus making explicit the implicit heap, and top of heap.

Of course, procedural languages do not usually allow for assignment of function-valued expressions. However, there is no reason why they should not (see for example [27,37]). For the purpose of modelling, it is possible to suppose that any procedural language contains any denotable value domain required by the introduction of the extra identifiers.

3.3.7.3 Modelling Pointer Dereference

An address may be dereferenced, that is, the information stored at this address in the heap can be referred to, using the address. For the address, *a*, the value stored at *a* in the heap, is found by applying the heap function, H, to the address, *a*.

Of course, the dereference of an address may, *itself*, be an address, thus allowing the heap to contain values which refer to other parts of the heap.

3.3.7.4 Modelling Address Values

The "nil" pointer value will be assigned a "special" value, outside the source domain of H. For example, -1 could be used.

In order to make models easier to read, nil, will be referred to by its symbolic name.

Dereference of the value "nil" is an error.

Demonstrating that the heap function is never applied to "nil", is thus a form of store-access integrity proof.

In languages like "C" and assembly code, where address arithmetic is permitted, it will not always be possible to refer to addresses symbolically. In particular, the store-access integrity proof will be that the heap function, H, is always applied to values x, such that $0 \leq x \leq hp$.

3.3.7.5 Modelling Changes to the Heap

An assignment statement may update the contents of the heap by dereferencing an address.

The simplest example of this is the dereference of a pointer variable. For example:

```
p^ := 42
```

This statement can be modelled procedurally by the (re)assignment to the heap function:

```
H := fn x => if x = p then 42 else H(x)
```

That is, the new heap maps all addresses to the same values as the old heap, except for the address, p, which is mapped to 42.

3.3.7.6 An Example of Modelling: Aliasing

Assignment of one pointer variable to another does not affect the heap itself, but it creates an alias as the following example shows:

```
program SimplePointer ;
var x, y : ^integer ;
begin
    new(x) ; new(y) ;          (* get an address for pointers x and y   *)
    x := y ;                   (* x^ and y^ are now aliases              *)
    x^ := 42 ;                 (* implicitly assigns a value to y^      *)
    write(y^)
end.
```

The call **new**(p) simply assigns to the pointer variable, "p", the value of the top of the heap and increments the top of heap heap pointer.

The single **write** instruction can be modelled by the introduction of an extra identifier, "Result", to store the value that appears on the screen.

3.3.7.7 The Model for the "SimplePointer" Program

The model for the program "SimplePointer" is given below. The intermediate stage of writing assignment statements for implicit state changes is omitted.

The model is simply a sequence of definitions, modelling changes to the environment.

The model is given below:

```
fun SimplePointer =
  let val hp = 0                    ; (* initial value of top of heap *)
      val H = fn z => Rubbish       ; (* initial value of heap *)

      val x = hp                    ;
      val hp = hp + 1               ;  (*  models new(x)  *)

      val y = hp                    ;
      val hp = hp + 1               ;  (*  models new(y)  *)

      val x = y                     ;  (* x and y now identical *)

      val H = fn z => if z = x then 42 else H(z);(* models x^:=42 *)

      val Result = H(y)                    (* models write(y^) *)
  in
  (H,hp,x,y,Result) ;
```

Unfolding "Result" gives: Result = 42.

The aliasing of "x" and "y" is modelled by the fact that these two identifiers are bound to the same integer value. Thus, when the heap function is applied to either, the same value will be returned.

3.3.7.8 Modelling A Self Referential Structure

Used in a disciplined manner, pointers allow the programmer to define and use lists, trees and other Abstract Data Types. Indeed, using the modelling technique, the programmer can *prove* that an Abstract Data Type is correctly implemented (see, for example, §6.6).

Used in an undisciplined manner, however, the structures created can become circular, as this simple example demonstrates:

```
program Money ;
type
      string = ^listrec ;
      listrec = record
                  data : char ;
                  rest : string
                  end ;

var Inf : string ;
begin
   new(Inf) ;
   Inf^.data := '£' ;
   Inf^.rest := Inf
end.
```

3.3.7.8.1 Modelling Record Structures

Record structures can be modelled using the tuple type.

The record type, "listrec", is modelled by a tuple, with selection of tuple-elements written using the symbolic names "data" and "rest", taken from the field names of the record.

For a tuple, T, and an index, f, the notation "$T^{\downarrow}f$" represents an expression which indexes the fth element of the tuple, T.

3.3.7.8.2 The Model for the Program: "Money"

The model for the program "Money" is as follows:

```
fun Money =
  let val hp  = 0                                                          ;
      val H   = fn z => Rubbish                                            ;
      val Inf = hp                                                         ;
      val hp  = hp + 1                                                     ;
      val H   = fn z => if z=Inf then ('£',H(Inf)↓data) else H(z)          ;
      val H   = fn z => if z=Inf then (H(Inf)↓rest,Inf) else H(z) in

          (H,hp,Inf) ;
```

Unfolding, "H" in the returned tuple, gives:

```
H = fn z => if z=Inf then ('£',Inf) else Rubbish ;
```

Unfolding H(Inf) gives:

```
H(Inf) = ('£', Inf) ;
```

In chapter four a technique called "Projective Abstraction" is described. This technique makes it possible to create a model specific to a particular value, or set of values. This technique can be used to produce models specific to particular values, such as:

```
H(Inf) = ('£', Inf) ;
```

It should be pointed out that the pleasing way in which a self-referential structure such as H(Inf) "announces itself" in the model notation is possible, *only* because an address may be referred to symbolically.

3.3.8 Proving a Model Strategy Correct

Proving that a modelling strategy is correct (in terms of the semantics of the language) is a straight forward, but long-winded matter.

A example demonstration of the construction of a simple modelling strategy and a proof of its correctness can be found in §7.4.

3.4 Assignment and Referential Transparency

In literature concerning functional programming, there appears to be some confusion about precisely what feature of procedural languages makes them non-referentially transparent (or referentially-opaque).

Many authors claim that it is the assignment statement that causes a language to be referentially-opaque.

For example on page six of [16] Stoy says:

> "In most programming languages referential transparency appears to be destroyed. For example, the fact that we have deduced x=6 would not imply that we could replace x by 6 everywhere within the scope of the declaration of x; if the program contains a statement like :
>
> **if x>y then x := x - 1**
>
> the value of x is not independent of position - afterwards it even depends upon the value of y."

This view is slightly misguided.

The statement used as an example by Stoy, can be modelled by a function "f", which takes the original values of the variables "x" and "y" and returns the final values of "x" and "y".

```
fun f(x,y) = let val x = if x>y then x-1 else x in (x,y) ;
```

There are six occurrences of the *identifier* "x" in this string of characters. The first, third, fourth and fifth occurrence of "x" refer to one value, the second and sixth refer to a different value. A *variable* is a *binding* of an *identifier* to a *value*. The above string thus contains one identifier, "x", *but two variables* which use "x". Scope rules are used to distinguish between different variables which use the same identifier.

If an assignment statement such as the one described above is referentially-opaque, then the lambda calculus must also be referentially-opaque. However, this conclusion would clearly be absurd: the Beta reduction rule distinguishes between different variables which happen to have the same identifier. It does this using a scope consideration embodied in the notion of bound and free variables[12].

It is not the assignment statement that denies a language the referential transparency property. It is the *implicit state*. Changes to parts of this implicit state cannot be manipulated by substitution simply because there is no identifier for which a value may be substituted.

It *may* be that changes to an implicit state are written using an assignment notation (for example, in the case of heap changes), however, other implicit-state changes are notated differently, for example, input and output statements.

3.5 Summary

This chapter collects together known functional modelling techniques under a common framework.

Some of these techniques exist in print [44,61,79,80] and some are simply part of the "folklore" of programming.

The framework used, is the semantics of the program, specifically the program's computation state.

If this state is of the form identifiers to values, then the "known techniques" can be used to construct a model for a statement sequence.

This model is a function, taking, as its argument, a tuple of the needed variables of the statement-sequence, and returning as its result, the tuple of final values of the affected variables of the statement-sequence.

Unfortunately, the "known techniques" break down when the state of computation is not of the form identifiers to values. However, as shown in §3.3, this problem can be overcome by the introduction of extra identifiers.

The chapter thus presents a strategy for modelling *any* procedural programming language feature.

The chapter also introduces the concept of an implicit and explicit state and demonstrates that it is the latter that prevents a procedural language from being referentially transparent.

CHAPTER FOUR

PROJECTIVE ABSTRACTION

4.1 Introduction

A model is a function which returns a tuple of values. Each value in this tuple is a semantic value computed by the procedural program being modelled. A semantic value is either a value bound to an identifier used by the program or a value of a component of the implicit state (for which an identifier has been introduced according to the strategy described in chapter three).

For any reasonably large program there will clearly be a large number of values in this result tuple, too many to make the model useful as a tool for analysis of the procedural program.

This chapter introduces a very simple technique for analysis of functions which return tuples of values. The technique is called "Projective Abstraction".

In conjunction with the implicit-state modelling approach described earlier in chapter three, Projective Abstraction allows a programmer to create many distinct models of a single procedural program. Each model is specific to the analysis of a particular set of semantic values computed by the procedural program and "abstracts away" from all other details of the execution which do not contribute to the computation of this set of values.

4.2 The Projective Abstraction Technique

The Projective Abstraction Technique consists in simply omitting some of the values of a model function's result tuple.

Thus, a function is projected onto a smaller target domain by restricting the result tuple.

An important consequence of this projection is that the amount of computation required to produce the result tuple is also reduced. Specifically, an expression is only included in the projected function if it contributes to the evaluation of the restricted result tuple.

Using the approach described in chapter three, a model function will contain an identifier for any and every semantic value change created by the execution of the program.

The Projective Abstraction Technique thus allows the programmer to choose *some* of these semantic values, and to produce a functional program which computes *only* the changes in these values arising from execution.

The large body of work on functional reasoning techniques [2,20,23,40,50,60] can then be brought to bear in analysing the affect of the procedural program upon the semantic values.

Consider the example below:

```
program TwoAssignments ;
var x, y : integer ;
begin
   x := E1 ;
   y := E2
end.
```

The model for this program is simply a function:

$$f(x,y) = (E_1, E_2) ;$$

If the programmer is interested in the final value of "x", then the projected model would be:

$$f(x,y) = E_1$$

4.3 Reduction in Needed Variables

A further simplification that arises from projective abstraction is that the size of the needed variable tuple may be reduced.

If the programmer decides to project the model onto "x", then the needed variable tuple tells the programmer what the final value of "x" depends upon. For example, if the needed variable tuple is *empty*, then "x" is a constant.

The Projective Abstraction Technique allows a programmer to use similar kinds of analysis familiar from "run-time debugging", that is, the inspection of the contents of a particular variable or variables. However, there are *two* crucial differences:

(1) The analysis is performed at *compile time* and is thus a *symbolic* analysis, ranging over *all* possible executions of the program.

(2) *All* values of semantic interest are available for inspection due to the modelling of the implicit state (with identifiers).

The Projective Abstraction Technique is thus simple, but powerful.

In the next few sections some applications of the technique are described:

4.4 Henderson's Lazy Streams

If the programmer projects the model onto the output list, then the model will be a stream-based program in the Henderson Lazy Stream style[30].

Using stream-based models of this kind does not allow a programmer to investigate the *interleaving* of input and output *events*.

However, this is often an advantage, as the programmer will want to ignore such details and focus on the functional relationship between input and output.

Should the programmer wish to investigate the interleaving of input and output then they could employ the modelling strategy described in §7.8.

4.5 New Programs From Old

Many programs compute several values within one part of the program. A programmer may wish to reuse only a *part* of a program to compute just one of these values.

Consider for example the program below:

```
program ArrayProcessing ;
var i,total, Biggest, Smallest : integer ;
    A : array [1..1000] of integer ;

begin
  total := 0 ; Biggest := A[1] ; Smallest := A[1] ;
  for i := 1 to 1000 do
      begin
      total := total + A[i] ;
      if A[i] > Biggest then Biggest := A[i] ;
      if A[i] < Smallest then Smallest := A[i] ;
      end
end.
```

Separate programs can be created for each value computer using Projective Abstraction. For example, choosing the final value of the identifier "Biggest" as the result of the model yields the following model:

```
fun Biggest(A) = for(A,A(1),1)
     where fun for(A,Biggest,i) = if i <= 1000
                                  then if A(i) > Biggest
                                       then for(A,A(i),i+1)
                                       else for(A,Biggest,i+1)
                                  else Biggest ;
```

Using the techniques described in chapter five it is possible to convert this model back into a procedural notation.

However, it would be foolish to convert *any* model back into a procedural notation without first investigating the model a little. The whole point of modelling is to permit the use of functional analytic techniques. One such technique is partial evaluation, which can be conducted symbolically at compile-time.

4.5.1 Partial Evaluation and Efficiency Improvement

Partially evaluating a functional program can result in an improvement in the efficiency of the program. This benefit is demonstrated by partial evaluation of the model "Biggest".

The call to "for" in the definition of "Biggest" is partially evaluated, giving:

```
for(A,A(1),1) = if 1 <= 1000
                then if A(1) > A(1)
                        then for(A,A(1),2)
                        else for(A,A(1),2)
                else A(1) ;
```

Clearly

"1 <= 1000" is "true"

and

"A(1) > A(1)" is "false"

Thus

```
for(A,A(1),1) = for(A,A(1),2)
```

Using this identity, the model can be rewritten:

```
fun Biggest(A) = for(A,A(1),2)
     where fun for(A,Biggest,i) = if i <= 1000
                                  then if A(i) > Biggest
                                          then for(A,A(i),i+1)
                                          else for(A,Biggest,i+1)
                                  else Biggest ;
```

When converted back into the Pascal notation this leads to the program below:

```
program Biggest ;
var Biggest, i : integer ;
    A : array [1..1000] of integer ;
begin
  Biggest := A[1] ;
  for i := 2 to 1000 do if A[i] > Biggest then Biggest := A[i] ;
end.
```

4.6 Values of Program Constants

In the program "ArrayProcessing", projecting the model onto the final value of the identifier "i" gives the model:

```
val i = 1001 ;
```

This does not tell the programmer much about the program but then, since the techniques can be automated, it does not require any effort either.

4.7 Heap Store Use

The identifier "hp" is introduced to model the top of the heap. Projecting a model onto the value of "hp" will tell a programmer how much heap store is used.

If the model yields a constant when projected onto this value, then it reveals that fact that there is *no need* for the heap. The heap storage strategy is *only* required when a programmer does not know how much store will be required.

4.9 Nothing New

Of course, there is *nothing* that a programmer can do using Projective Abstraction (or modelling in general) that was not possible using the original procedural program. The procedural program's text is a complete specification of the programs behaviour, and *can* be used to answer these sorts of question about the program's execution.

A good programmer will be able to extract the "Biggest" program from the "ArrayProcessing" program (§4.5) and will notice the efficiency improvement to be gained by omitting the first loop cycle.

The modelling approach offers two significant enhancements to such "ad hoc" reasoning:

(1) All the reasoning performed using the manipulation of the model is *guaranteed* to be correct. That is, *no* manipulation can alter the values computed by the program. The programmer is thus free to "play" with their program as if it were simply a piece of algebra, ignoring the fact that it is to be executed by a computer. This algebraic freedom is the principal advantage of a functional notation over a procedural notation (see, for example [80,90] and [21,18,32]). The modelling technique allows procedural programmers to avail themselves of this advantage.

(2) The production of projected models may be performed *entirely automatically* by a CASE tool (or to put it more prosaically, by a *Compiler*, albeit a compiler parameterised by the choice of Projection).

In a small program, such as the array processing example above, it may well be that a programmer can *see* immediately how to alter a program to calculate only one value. However, the dependencies that have to be considered in order to do this become too intricate for programs of greater size and Projective Abstraction becomes a necessity.

4.10 Summary

The technique of Projective Abstraction is very simple.

A programmer simply omits some values from the result tuple of a function, thus simplifying the function by specialising its result.

Together with the (equally simple) technique of introducing identifiers to model the implicit state described in chapter three, the programmer can analyse, evaluate, manipulate and prove properties of any semantic value of interest.

The functional modelling technique thus involves *two* abstractive filters which "filter out" irrelevant aspects of a program's execution, sharpening the analytic focus.

The first of these filters is applied by the programming language designer, who chooses which aspects of *all possible* programs are to be described in the semantic description of the programming language.

The modelling strategy uses this semantic description when creating a model, and so any aspect of execution ignored in the semantics will be ignored in the model.

The second abstractive filter is applied by the programmer, who chooses (by Projective Abstraction) those aspects of *a particular* program that are to be modelled.

The technique of Projective Abstraction reduces the complexity of the model to that which is sufficient to compute the semantic values onto which it is projected. In small programs this complexity-reduction is of little consequence, but for large programs its benefit will be keenly felt.

Chapter Five

CONVERTING MODELS BACK INTO PROCEDURAL NOTATION

CHAPTER FIVE CONVERTING MODELS BACK INTO PROCEDURAL NOTATION

5.1 Introduction

In this chapter various strategies for converting functional models back into a procedural notation are discussed.

Converting a model back into a procedural program is inherently a matter for the programmer, since the point of functional modelling is to reveal inadequacies in the original procedural program.

What constitutes an inadequacy depends upon the program concerned, so a highly intelligent (i.e. *human*) strategy is required to produce an improved program from a model.

Other techniques are discussed in the seminal work of John Darlington, which was originally aimed at compilation of functional programs into efficient procedural counterparts [91].

5.2 Evolution of program transformation rules

The techniques presented here are intended to be an *aid* to the programmer only. It may well be that using a semi-automated system, various heuristic rules will evolve and then become incorporated into automated parts of the system (see appendix A2).

For example the method of goto removal set out in [88] and [89] are examples of such heuristic rules: if the model produced by a "goto program" is converted into "while" loops by the strategy outlined here, then the resulting program will be very much like that produced by the algorithm described in [89].

5.3 Some Things which Will Not be Converted Back

The strategy outlined here is directed at converting functions in iterative form and does not address the problem of I/O interleaving.

5.3.1 Iterative Form

The model functions which can be converted back into a procedural notation, using the strategy outlined here, are those which are in iterative form [44,61].

A function, f, is in iterative form if all calls to the function do not occur within calls to any other function.

5.3.2 Loss of I/O Sequencing Information

As shown in chapter three, the information concerning the interleaving of input and output events is lost when a procedural program is modelled, using a simple list-based input and output semantics.

If a program, P, is manipulated to alter its structure and subsequently converted back into the procedural program, P′, then P′ will not *necessarily* interleave the input and output events in the same order as P.

5.3.2.1 Keeping the I/O Information Means Restricting the Fold/Unfold Rules

If the programmer wishes to maintain interleaving information then the model should include a "trace" in its projection (see §7.8).

5.4 Converting Functions in Iterative Form to Loop Constructs

The strategy proceeds by performing manipulations to the model functions, rendering all models in a common form, from which, conversion to loop constructs becomes trivial.

5.4.1 Maximal Substitution

All let abstractions, "let val x = E1 in E2", where E1 does not involve a call to a model function, are unfolded so that model functions all take the form:

$$f(\bar{x}) = \text{if } p_1(\bar{x}) \text{ then } E_1 \text{ else if } p_2(\bar{x}) \text{ then } E_2 \text{ else } \ldots \text{ else } E_n$$

Where $\bar{x}$ is the tuple of identifiers which form the parameters to the function f, p_i are predicates on this tuple and E_i are expressions containing no conditional sub expressions.

5.4.2 Unfolding of Functions

The expressions E_i will be one of four possible forms:

i) An expression involving only base functions and members of $\bar{x}$.

ii) An expression of the form $g(\bar{x})$, where g is a model function.

iii) An expression involving a model function which is not in iterative form.

iv) The fourth possibility is "let val $(\bar{y})$ = $g(\bar{z})$ in E", where $\bar{y}$, and $\bar{z}$ are subsets of $\bar{x}$, g is another model function and E is an expression in one of the forms i) - iv). This fourth possible form, when converted into a procedural notation, leads to nested looping, it is discussed separately in §5.7.

As stated earlier, it is not possible, using this strategy, to convert functions which are not in iterative form, so case iii) is ignored.

Expressions in form ii) should be unfolded until either a recursive call is encountered, or a call to a recursive function is encountered. That is, if an Expression, E, occurring in a function, f, is unfolded to produce a call to f (by definition in iterative form) or a call to a function g, where g is a recursive function, then E should not be unfolded any further.

If the model contains mutually recursive functions then the strategy that must be employed is less elegant due to the introduction of extra variables. For this reason models containing mutual recursion are discussed separately in §5.8.

5.4.3 The Fully Unfolded Model

After unfolding, each model function, $f(\bar{x})$, is in the form described in §5.4.3, where each E_i is one in one of the following three forms:

i) An expression involving only base functions and members of $\bar{x}$.

ii) A recursive call, f(e), where e is an expression involving only base functions and members of $\bar{x}$.

iii) A call to a function g, of the form g(e), where e is an expression involving only base functions and members of $\bar{x}$.

5.4.4 The Terminating Condition for a Model Function

"Termination" occurs when an expression of the form i) or iii) above is evaluated.

In this discussion, the condition under which such a call is evaluated is called the "termination condition" of the function.

That is, within a function, "f", the condition under which any of the expressions Ei which contain no recursive calls to "f", are evaluated.

The "termination" condition of a model function is not, therefore, a condition under which the model program *itself*, would terminate, but as will be seen, it *is* the condition under which the loop that models the function will terminate.

5.4.4.1 Examples

Below are two functions together with their "terminating conditions":

```
fun f(x,y) = if x = y then x
                      else if x = (y+1) then y-x
                                        else f(x-1,y+1) ;
```

The function, f, has "terminating condition", p(x,y):

```
p(x,y) = (x=y) or (x=y+1)
```

```
fun h(x,y,z) =
     if x<>y then h(x-1,y+1,x+z)
             else if y=z then y
                         else if x=z then h(x-1,y-1,z)
                                     else g(x,y,z) ;
```

The function h has "terminating condition", p(x,y,z):

```
p(x,y,z) = not(x<>y) and (y=z or (not(x=z))
```

In each case, the condition is produced, merely by examining the predicates in the conditionals, without regard to the value computed by the function.

Of course, the terminating condition may, itself, be manipulated according to the rules of the predicate calculus. For example the second predicate, p, can be rewritten:

```
p(x,y,z) = x=y
```

5.4.5 Converting to an Unbounded Loop

A recursive function, in iterative form, is converted into an unbounded loop. That is, it is converted, either into a "while" loop, or a "repeat" loop.

A recursive function will be converted into a "while" loop if it "terminates" without modifying any of its parameters. That is, if it returns a subset of its parameter tuple or passes some subset of its parameter tuple to another model function.

A recursive function will be converted to a "repeat" loop if it "terminates" with some modified subset of its parameter tuple. That is, if it returns an expression involving a subset of its parameter tuple and some base functions, or passes such an expression to another model function.

There is, of course, some choice that may be exerted, since a "repeat" loop with a body S, is equivalent to a "while" loop with S executed once before entry.

5.4.5.1 The Loop's Boolean Expression

If a "repeat" loop is being produced then the boolean expression is simply the "terminating condition" of the model function.

If a "while" loop is being produced then the boolean expression is the negation of the "terminating condition" for the model function.

5.4.5.2 The Loop's Body

The nested conditional expression is converted to a nested conditional statement:

```
if p1(x̄) then e1
         else if p2(x̄) then e2
                       else ...
                                    else en
```

Is converted to:

```
if p1(x̄) then S1
         else if p2(x̄) then S2
                       else ...
                                    else Sn
```

Except that those predicates that form part of the "terminating condition" need not be included.

Each statement, S_i, is produced by converting the expressions, e_i, as follows:

5.4.6 Local States and Variables

The parameters of a model function are the "local state" in which the function is evaluated.

The parameter names of this local state will be used as variable names in the procedural program.

The body of the loop produced contains assignments to these variables. These assignments are formed by assigning to each variable, the expression passed on recursive call to the function.

5.4.7 Input and Output

The re-assignments to the output list and input list will be converted back into write and read statements.

If input occurred in the part of the procedural program modelled by a model function, f, then f will include a parameter, "inp" (or some other name introduced according to the strategy set out in chapter 3). If output occurred in that part of the procedural program, then f will include a formal parameter "out" in its parameter list.

The parameters "inp" and "out" correspond to the input list and output list. (The situation where there are more than one output device can be dealt with naturally by extending the strategy outlined here).

Obviously, names are significant in the model, so if conversion into a procedural notation is performed *automatically*, a convention on names for devices must be followed (see §5.9).

5.4.7.1 Output

Output is modelled by appending output elements onto the output list. So to convert this back into the procedural notation, where ever the actual parameter "append(Out,L)" is passed for the formal parameter "Out" in a function call this will be converted to **write**(L′), where L′ are the elements of the list L.

5.4.7.2 Input

Converting operations on the input list back into the procedural notation as read statements presents a some problems:

The statement "**read**(x)" is modelled by "**let val** x = hd(Inp) **in let val** Inp = tl(Inp) **in** ...".

Now, these two definitions may become "separated" in the model due to manipulation, but in converting a model into procedural notation expressions, involving "hd(Inp)" and "tl(Inp)" must be converted to a single "**read**" statement.

5.4.7.2.1 Introducing New Variables

Normally (for example, in Pascal, C and Fortran) the read statement takes a variable parameter, in which the input is to be stored. The name for this variable parameter will have to be "invented".

5.4.7.2.2 Where to Locate the "read" Statements

In order to discuss conversion of operations on the input list it is necessary to introduce two concepts: the "depth of input" in a model function and the "amount of input consumed", corresponding to the expressions "hd(Inp)" and "tl(Inp)" respectively.

5.4.7.2.1 The Amount of Input Consumed

The recursive call to a function will pass an actual parameter for the formal parameter "Inp". The expression for this actual parameter represents "how much input is consumed on that call".

5.4.7.2.2 The Depth of Input

The selection of elements from the input list has a "depth", this is the highest index used to select an element from the input list.

The value of the depth and the amount of input consumed in a particular model function are not *necessarily* identical.

5.4.7.2.3 Depth is the Same as Amount Consumed

In most model functions the depth will be identical to the amount of input consumed.

In such cases all the read statements are to be put *inside* the loop at the appropriate point in the conditional statement structure.

For example, the function f, below, the depth is identical to the amount of input consumed on each call:

```
fun f(Inp,t) = if t>100 then t
                        else f(tl(Inp),t+hd(Inp)) ;
```

It will be converted into the "while" loop:

```
while not(t>100) do
                 begin
                 read(x);
                 t := t + x
                 end
```

5.4.7.2.4 Depth is Smaller than Amount of Input Consumed

If the depth is smaller than the amount consumed, then "dummy" read statements must be included to read the extra input.

For example the function f, below, consumes 3 elements from the input list on each call, and has a depth of 1.

```
fun f(Inp, t) =
   if t>100 then t
            else f(tl(tl(tl(Inp))),t+hd(Inp)) ;
```

It will be converted into the "while" loop:

```
while not(t>100) do
                 begin
                 read(x);
                 read(Dummy) ;
                 read(Dummy) ;
                 t := t + x
                 end
```

5.4.7.2.5 The Depth is Greater than the Amount of Input Consumed

If the depth is greater than the amount of input consumed then some input must be read before the loop is executed, and then at the end of the loop body these variables must be updated.

For example consider the model function, f, below:

```
fun f(Inp,t) =
  if t > 1
     then t
     else
     f(hd(Inp)+hd(tl(Inp))+hd(tl(tl(Inp))),tl(Inp));
```

In this function the depth is 3 but the amount of input consumed on each recursive call is 1.

Two values must therefore be read before the loop is executed, and these variables must be updated at the end of the loop body.

The "while" loop produced to model this procedure is:

```
read(X1) ;
read(X2) ;
while not(t>100) do
                 begin
                 read(X3) ;
                 t := X1 + X2 + X3 ;
                 X1 := X2 ;
                 X2 := X3
                 end
```

This is a highly unlikely eventuality however.

5.4.8 Example

Consider the function, "S", below:

```
fun S(Inp,N,M,t) = if N=M then t
                          else let val t' = t+hd(Inp) in
                                  let val N' = N+1 in
                                     S(tl(Inp),N',M,t') ;
```

5.4.8.1 Unfolding

First, the let abstractions are unfolded, to produce:

```
fun S(Inp,N,M,t) = if N=M then t
                          else S(tl(Inp),N+1,M,t+hd(Inp));
```

Next, all function calls are unfolded until recursion is encountered. In this case, this means no unfolding, since the function call "S(tl(Inp),N+1,M,t+hd(Inp))" is already recursive.

5.4.8.2 The Loop-Body

The conditional structure is

if M=N **then** S_1 **else** S_2

Where S_1 and S_2 are produced according to the strategy outlined in §5.4.6. (This simply reduces to S_2, since there is no need to test the terminating condition within the loop).

5.4.8.3 Converting to a "while" Loop

The function is converted into a "while" loop, since it returns "t" unaffected if its "termination condition" is satisfied.

The "Termination condition" is "p(N,M) = N=M", so the "while" loop predicate is "**not**(N=M)".

5.4.8.4 Input and Output

Now, this function consumes one element of "Inp" on each call so the expression "hd(Inp)" is converted to the read statement "**read**(x)" which is to be placed inside the loop, and where ever "hd(Inp)" occurs in an expression it is replaced by "x".

5.4.8.5 Re-Introduction of Assignment

The body of the loop contains the assignments:

t := t+x

and

N := N+1

The "while" loop produced is thus:

```
while not(N=M) do
               begin
               read(x) ;
               N := N+1 ;
               t := t + x
               end
```

5.5 Putting the Loops into a Statement Sequence

All that is required to produce a complete procedural program, is to put the various "while" and "repeat" loops into a sequence, reflecting the call graph of the model.

The model is a function f_1, which may call a functions $f_2,\ldots,f_n$, each of which may, themselves, call more functions.

The first "while" loop in the statement sequence is the one which was produced from f_1. If there is only one other function, f_2, called from f_1, then the next loop in the statement sequence is that produced from f_2.

If there is more than one function, $\{f_2,\ldots,f_n\}$, called from f_1, then a conditional statement is the next statement in the procedural program produced. This conditional selects which loop is to be executed next from those produced from the functions $\{f_2,\ldots,f_n\}$.

5.5.1 Avoiding complexity

A little consideration leads to the realisation that for a large set of model functions, each of which calls several other model functions, the procedural statement-sequence produced by this naive strategy will be long and complicated, involving many nested conditionals. The depth of conditional nesting being roughly proportional to the number of model functions.

This situation is clearly unacceptable; it arises from the fact that the procedural program produced according to this strategy makes no use of procedure-abstraction: the program produced will be one monolithic sequence of statements.

The obvious way to subdivide this monolith is to make use of the procedural abstraction contained in the original procedural program.

The model functions fall into three categories: Those whose names are introduced to model "goto" statements (i.e. those whose names are labels), those whose names are introduced to model loop constructs, and those which model procedures and functions in the original program, and use the original names.

The last of these three are those model functions which correspond to procedures in the original program. These model functions can be converted into separate procedural programs, and each of these programs can then be written as a procedure. The procedure will take as formal parameters, the needed variables of the statement sequence produced, and return (via variable parameters) the affected variables of the statement sequence produced.

5.6 Automation

All of these conversions can be carried out automatically. However, it is doubtful whether a *fully* automated strategy would be desirable: The programmer's intuitions about the model functions will provide valuable insights into what manipulations to perform, and what conversion strategy to use when converting back into a procedural notation.

For example, the programmer may decide that some of the model functions produced from loops or "goto" statements should be converted back into procedures, thus increasing the resulting procedural abstraction, and improving the readability (and *reusability*) of the resulting procedural program.

What *is* highly desirable is a semi-automated "CASE tool", many parts of which would be *fully* automatic (for example production of the initial model functions, converting back according to various pre-defined strategies, fold/unfold rules in the functional notation, maximal substitution and so on).

Such a tool would be used *interactively* by a programmer, who would analyse and manipulate a procedural program, producing proofs of program properties, correcting mistakes, removing redundant computations and altering the program's structure to produce a better documented, more reliable, efficient, reusable and proven program.

Some preliminary work has already been conducted on these lines (see appendix A2), and work continues[74], but more time and resources are required.

The strategy is not complicated, all examples could easily be performed by a programmer in a heuristic manner without reference to an algorithm of conversion. Automation would merely act to remove the "donkey work" involved in conversion of notations.

5.7 Nested Loops

The discussion so far has been concerned with model functions which contain no nested calls to functions.

A nested call, occurring within a function f, is an expression E of the form below:

$$\boxed{\textbf{let val } (\bar{x}) = g(\bar{x}) \textbf{ in } E'}$$

Where E′ is either of the four forms listed in §5.4.2.

Such nested calls to functions produce procedural programs with nested loops.

5.7.1 Example

In this example the function "r" is used to strip leading spaces from a list, it terminates when the head of the list is not a character.

```
fun f(Inp,Out) =
  if hd(Inp) = '.'
     then Out
     else
     if hd(Inp) = ' '
        then let val Inp' = r(tl(Inp))
               in f(Inp',Out)
        else f(tl(Inp),append(Out,[hd(Inp)])) ;

fun r(Inp) = if hd(Inp) = ' ' then r(tl(Inp))
                              else tl(Inp) ;
```

The function "r" is converted to the "repeat" loop:

```
repeat read(x) until x <> ' '
```

The function "f" is converted to the repeat loop:

```
repeat
    read(y) ;
    if y = ' ' then repeat read(x) until x <> ' '
              else write(y)
until y = '.'
```

For greater perspicuity, the programmer may elect to form a procedure out of such nested loops, particularly if the same model function is nested in several places. This will be another area where a "CASE tool" would *need* to be interactive, allowing the programmer to choose how a model is converted into a procedural program.

5.8 Mutually-Recursive Model Functions

In [89] the authors demonstrate that it is not possible to convert all procedural programs, which contain "goto" statements, into procedural programs, where the "goto" statements are replaced by "while" loops, *without* introducing extra variables.

This is also obviously true for functional models. Those models which cannot be converted into "while" loop programs, without the introduction of extra variables, are *precisely* those model functions which are *mutually recursive.*

Consider the two mutually recursive functions below:

```
fun f(x1,...,xn) = if p1(x1,...,xn) then E1
                   else if p2(x1,...,xn) then g(E2)
                   else f(E3)  ;

fun g(x1,...,xn) = if p3(x1,...,xn) then E4
                   else if p4(x1,...,xn) then f(E5)
                   else g(E6)  ;
```

These two mutually recursive functions can be joined together into one function, "Both", by adding an extra parameter to "switch" between the two in the body if this new function:

```
fun Both(x1,...,xn,switch) =
  if switch = "f"
     then
     if p1(x1,...,xn) then E1
        else if p2(x1,...,xn) then Both(E2,"g")
        else Both(E3,"f")
  if switch = "g"
     then
     if p3(x1,...,xn) then E4
                      else if p4(x1,...,xn) then Both(E5,"f")
                      else Both(E6,"g")
  else (* invalid switch *) ;
```

Now the call "f(e_1,...,e_n)" is equivalent to Both(e_1,...,e_n,"f") and a call "g(e_1,...,e_n)" is equivalent to Both(e_1,...,e_n,"g").

These are the *only* two calls to the function "Both" which are valid. Passing a value for "switch" other than "f" or "g" will not reflect *any* possible call in the original model program.

However, the production of a function like "Both" is only intended to indicate a strategy for converting mutually recursive model functions into procedural notation; functions like "Both" will never actually be *executed*.

The strategy can clearly be extended to cope with an arbitrary number of mutually recursive functions, all that is required is for "switch" to have as many *possible* values as there are *possible* mutually recursive calls.

Now the function "Both" can be converted into a "while" loop according to the strategy outlined in §5.4.5. Of course, the "while" loop will also mention the *new* variable "switch".

Clearly, it would be possible to convert the model into mutually recursive procedures, in which case *no* extra variables would need to be introduced.

5.9 Naming Conventions in the Model

In converting operations on input and output lists, an assumption is made that the names used for input and output lists are "inp" and "Out" respectively. It has also been assumed that names of procedures will be maintained in the model.

This presents no problems, and is only mentioned since it runs contrary to the normal experience that the choice of particular identifiers is unimportant and that consistent name changes may be performed in functional programs without changing meaning.

The model functions are used to model procedural programs and so in addition to their "meaning" as recursion equations, model functions also have a "meaning" in terms of the program that they model.

CHAPTER SIX

APPLICATIONS OF FUNCTIONAL MODELLING

CHAPTER SIX APPLICATIONS OF FUNCTIONAL MODELLING

6.1 Introduction

Any aspect of a procedural program which can be analysed, proved or altered will be an aspect to which the functional modelling technique may be applied. There are thus many applications of the modelling approach.

Moreover, by simply projecting a model onto the desired "problem domain" as described in chapter four, the programmer can "home in" on the particular aspect of interest an drop from the model any irrelevant details.

This chapter demonstrates some of the wide variety of application areas for which the modelling strategy is suited. It seems that the larger a procedural program is, the more benefit will be gained from the abstractive features offered by the approach. However, only reasonably small examples can be contained in this exposition (the largest program is about five pages (§6.4)).

Automation of the modelling techniques will allow the programmer to approach programs of unlimited size.

6.1.1 A Brief Outline of the Examples Contained Here

There are seven examples in this chapter.

These are as follows:

In §6.2 a Fortran program is modelled. The model is manipulated and converted back into procedural notation. Several equivalent Pascal programs are produced and an equivalent Fortran IV program is produced with "goto" statements removed.

In §6.3 a Fortran program is modelled. For this program the (English language) specification is known. The program is shown not to obey its specification and is corrected. Once again the model is converted back into several equivalent procedural programs (in Pascal).

In §6.4 a larger program is considered. This program is taken from a standard reference on the programming language Modula-2 [67].

The program implements a simple data base.

The modelling strategy is ideally suited to proving properties of such programs. Many simple proofs are constructed, demonstrating simple properties of the Modula-2 program. These simple proofs are then used to construct a few more general assertions about the program. These general assertions can be treated as a *specification* for the behaviour of the Modula-2 program.

The construction of proofs for the program reveals some shortcomings in the program design whereby certain input values could cause the program to misbehave. The procedural program is thus rewritten to take account of these problems, making it more robust.

In §6.5 the Algol 68 program "FIND", first proved correct by Hoare in [33], is modelled. Since the program was constructed and proved in [33] using the Axiomatic Method, the program provides an opportunity to compare the relative merits of the Axiomatic Method and the Modelling strategy. The conclusion of this comparison is that modelling is better for analysis of *existing* programs, whilst the Axiomatic Method is ideally suited to the *construction* of correct programs.

One of the lemmas used in the proof of the program "FIND" presented here, allows the program to be manipulated to remove its "goto" statement.

In §6.6 an example of disciplined use of pointers is examined. A set of Pascal functions which implement a linked-list are modelled.

A proof is then constructed, demonstrating that the Pascal functions respect the list Abstract Data Type axioms.

§6.7 turns attention to the programming language "C". This language contains many semantic subtleties which become transparent when modelled.

Finally, §6.8 looks at the possibility for locating possible paths for parallel evaluation using the modelling strategy.

The modelling strategy may well be applicable to other areas of analysis and proof. Some of these are described in chapter nine.

6.2 Goto Removal in Fortran IV

In [43] Dijkstra argues against the use of the goto statement.

In this example a Fortran IV program from [87] has been used to illustrate how manipulation of the functional model can lead to a more elegant program, in the sense of goto-removal

After manipulation the model can be converted back into a procedural notation in several ways. In this example the program is converted back into a "repeat loop" program in Pascal, a "for loop" program in Pascal and a "do loop" program in Fortran.

The strategy used to convert back into the procedural notation is the one described in chapter five.

6.2.1 The Program

The program is taken from page 145 of [87].

```
      I = I1 - 1
6     J = J1 - 1
7     IF (I.EQ.I1 .AND. J.EQ.J1) GO TO 9
      IF (I.LT.1 .OR. I.GT.8 .OR. J.LT.1 OR J.GT.8) GO TO 9
      IF (BOARD(I,J).EQ.0) WRITE(6,8) I,J
8     FORMAT (2I5)
9     J = J + 1
      IF (J.LE.J1+1) GO TO 7
      I = I + 1
      IF (I.LE.I1+1) GO TO 6
```

6.5.2 The Model

The model uses a function to model the array "Board" and the McCarthy strategy for modelling goto statements. The identifier "Out" which is a list modelling the output device 6, is the result onto which the model is projected.

The program is modelled by the ML function f:

```
fun f(Out,i1,j1,Board) = f6(Out,i1-1,i1,j1,Board) ;
fun f6(Out,i,i1,j1,Board) = f7(Out,i,j1-1,i1,j1,Board) ;

fun f7(Out,i,j,i1,j1,Board) =
  if (i=i1) and (j=j1)
     then f9(Out,i,j,i1,j1,Board)
     else if (i<1) or (i>8) or (j<1) or (j>8)
             then f9(Out,i,j,i1,j1,Board)
             else if Board(i,j) = 0
                     then f9(append(Out,[i,j]),i,j,i1,j1,Board)
                     else f9(Out,i,j,i1,j1) ;

fun f9(Out,i,j,i1,j1,Board) =
  if (j+1) <= (j1+1)
     then f7(Out,i,j+1,i1,j1,Board)
     else if (i+1) <= (i1+1)
             then f6(Out,i+1,j,i1,j1,Board)
             else End(Out) ;
```

6.5.3 Manipulation of the Model

6.5.3.1 Initial Manipulation

The parameters il,jl and Board are not altered in any of the model functions and so they can be treated as constants. The whole program is then modelled by the call f(Out), and is a function from the original output list to the final output list resulting from executing the program.

```
fun f(Out) = f6(Out,il-1) ;
fun f6(Out,i) = f7(Out,i,jl-1) ;

fun f7(Out,i,j) =
  if (i=il) and (j=jl)
     then f9(Out,i,j)
     else if (i<1) or (i>8) or (j<1) or (j>8)
             then f9(Out,i,j)
             else if Board(i,j) = 0
                     then f9(append(Out,[i,j]),i,j)
                     else f9(Out,i,j) ;

fun f9(Out,i,j) =
  if (j+1) <= (jl+1)
     then f7(Out,i,j+1)
     else if (i+1) <= (il+1)
             then f6(Out,i+1,j)
             else End(Out) ;
```

6.5.3.2 Unfold f6

Note that f6(a,b) = f7(a,b,jl-1) so the calls to f6 can be replaced by the corresponding call to f7.

6.5.3.3 Predicate Properties

Trivially, the following identity for predicates holds:

```
if p then e                          if p or q
     else if q then e        ≡          then e
               else r                   else r
```

Thus the first two calls to f9 in f7 may be "collected together" and guarded by a single predicate.

Finally, the predicates in f9 may be "cleaned up" using the (equally trivial) property of "<=" that a+1 <= b+1 ⇔ a <= b.

These manipulations further increase the "readability" of the model, and begin to unveil the algorithm that the model implements:

```
fun f(Out) = f7(Out,i1-1,j1-1) ;

fun f7(Out,i,j) =
  if ((i=i1) and (j=j1)) or (i<1) or (i>8) or (j<1) or (j>8)
     then f9(Out,i,j)
     else if Board(i,j) = 0
             then f9(append(Out,[i,j]),i,j)
             else f9(Out,i,j) ;

fun f9(Out,i,j) = if j<=j1 then f7(Out,i,j+1)
                           else if i<=i1
                                   then f7(Out,i+1,j1-1)
                                   else End(Out) ;
```

6.5.3.4 Folding to Reduce the Number of Calls to f9

The three calls to the function "f9" in "f7" can be collected together by introducing a let abstraction to define the value of the first actual parameter in the call.

```
fun f(Out) = f7(Out,i1-1,j1-1) ;

fun f7(Out,i,j) =
  let val Out' =
    if (i≠i1 or j≠j1) and i≥1 and i≤8 and j≥1 and j≤8 and Board(i,j)=0
       then append(Out,[i,j])
       else Out                    in

      f9(Out',i,j) ;


    fun f9(Out,i,j) = if j<=j1 then f7(Out,i,j+1)
                               else if i<=i1
                                       then f7(Out,i+1,j1-1)
                                       else End(Out) ;
```

6.5.3.5 Finally

Finally, by unfolding the one call to the function "f9" in the function "f7", the model is transformed into one recursive function.

```
fun f(Out) = f7(Out,i1-1,j1-1) ;

fun f7(Out,i,j) =
  let val Out' =
    if (i≠i1 or j≠j1) and i≥1 and i≤8 and j≥1 and j≤8 and Board(i,j)=0
       then append(Out,[i,j])
       else Out                    in

            if j<=j1 then f7(Out',i,j+1)
                     else if i<=i1
                             then f7(Out',i+1,j1-1)
                             else End(Out') ;

fun End(Out) = Out ;
```

6.5.4 Conversion of the Model Into Pascal

The termination of the function f7 occurs *only* when "End" is called.

Deducing the predicate which has to be true in order for this function to be called, allows a recursive function like f7 to be converted into a "while" or "repeat" loop in the manner described in chapter 5.

In this example the recursive function f7 corresponds to a repeat loop.

6.5.4.1 The Termination Condition

The termination condition for f7 is:

```
fun P(j,j1,i,i1) = (j>j1) and (i>i1) ;
```

6.5.4.2 Re-Introducing Assignment

A function in iterative form has a local state : the bindings of its parameters on call. Modification of these parameters is a direct analog of assignment. This is used in the transformation strategy to model assignment by let abstraction (see §3.2.2). Here, the correspondence is used in reverse (let abstraction being converted into assignment).

6.5.4.3 A Pascal Program with a Repeat Loop

A Pascal program which corresponds to the model is:

```
j := j1-1 ; i  := i1 - 1 ;
repeat
   if ((i=i1) and (j=j1)) or (i<1) or (i>8) or (j<1) or (j>8)
      then
      else if Board(i,j) = 0 then write(i,j) ;
   if j<=j1 then j := j+1
            else begin i := i+1; j := j1-1 end
until (j>j1) and (i>i1)
```

6.5.4.4 An Alternative Pascal Program

Of course there are many procedural programs that correspond to functional models just as there are many modelling strategies for a particular procedural construct.

By manipulating the model further using functional reasoning, it is possible to use the strategy described in chapter 5 to produce two nested repeat loops.

This is achieved by introducing a new function, which is done using the following rule:

```
fun f(x1,...,xn) =
   if p(x1,...,xn)
      then f(g1(x1,...,xn),...,gn(x1,...,xn))
      else E
```

can be re-written:

```
fun f(x1,...,xn) = let val (x1,...,xn) = inner(x1,...,xn) in E

fun inner(x1,...,xn) =
   if p(x1,...,xn)
      then inner(g1(x1,...,xn),...,gn(x1,...,xn))
      else (x1,...,xn) ;
```

Thus

```
fun f(Out) = f7(Out,i1-1,j1-1) ;


fun f7(Out,i,j) =
  let val Out' =
    if (i≠i1 or j≠j1) and i≥1 and i≤8 and j≥1 and j≤8 and Board(i,j)=0
       then append(Out,[i,j])
       else Out                    in

            if j<=j1 then f7(Out',i,j+1)
                     else if i<=i1
                             then f7(Out',i+1,j1-1)
                             else End(Out') ;

fun End(Out) = Out ;
```

can be written:

```
fun f(Out) = f7(Out,i1-1,j1-1) ;

fun f7(Out,i,j) =
    let val Out' = inner(Out,i,j) in
            if i<=i1
               then f7(Out',i+1,j1-1)
               else End(Out') ;

fun inner(Out,i,j) =
  let val Out' =
    if (i≠i1 or j≠j1) and i≥1 and i≤8 and j≥1 and j≤8 and Board(i,j)=0
       then append(Out,[i,j])
       else Out                    in

            if j<=j1 then inner(Out';i;j+1)
                     else Out' ;

fun End(Out) = Out ;
```

6.5.4.4.1 Converting This to Pascal Notation

Converting this model function into the procedural notation gives rise to nested repeat loops (the same conversion strategy is used as that outlined in chapter 5, which was used to produce the repeat loop in §6.5.4.3):

```
i := i1-1 ;
repeat
  j := j1-1;

  repeat
    if ((i<>i1) or (j<>j1)) and (i>=1) and (i<=8) and (j>=1) and (j<=8)
       then if Board(i,j) = 0 then write( i,j) ;
    j := j+1
  until j>j1 ;

  i := i+1
until i>i1
```

6.5.4.4.2 Converting Repeat Loops to For Loops

Given the statement sequence S and expressions E1 and E2, a repeat loop of the form:

```
j := E1 ;
repeat S; j := j+1 until j>E2
```

is equivalent to:

```
for j := E1 to E2+1 do S
```

Provided E2 ≥ E1.

So the model function could also be converted to two nested for loops as follows:

```
for i := i1-1 to i1+1 do
  for j := j1-1 to j1+1 do
    if (Board(i,j) = 0) and
       (i<>i1) and (j<>j1) and
           (i>=1) and (i<=8) and (j>=1) and (j<=8)
       then write(i,j) ;
```

6.5.4.4.3 Converting Into a Fortran-IV Program

It is most likely that conversion of a model back into a procedural notation will mean converting the program back into the same procedural notation from which it originated. In the case of this program the Fortran "do" loop can be used, and in this case leads to a far more elegant solution to the problem, the principal inelegance being the cumbersome notation for boolean expressions in Fortran IV.

The Original Program

```
      I = I1 - 1
    6 J = J1 - 1
    7 IF (I.EQ.I1 .AND. J.EQ.J1) GO TO 9
      IF (I.LT.1 .OR. I.GT.8 .OR. J.LT.1 OR J.GT.8) GO TO 9
      IF (BOARD(I,J).EQ.0) WRITE(6,8) I,J
    8 FORMAT (2I5)
    9 J = J + 1
      IF (J.LE.J1+1) GO TO 7
      I = I + 1
      IF (I.LE.I1+1) GO TO 6
```

The Modified Program

```
      DO 10 I=I1-1,I1+1
      DO 10 J=J1-1,J1+1
   10 IF (BOARD(I,J)=0 .AND. (I.NE.I1 .OR. J.NE.J1) .AND.
          I.GE.1 .AND. I.LE.8 .AND. J.GE.1 .AND. J.LE.8)
      WRITE(6,30) I,J

   30 FORMAT(2I5)
```

6.3 Language Conversion and Error Analysis

This example is also taken from McCracken's "guide to Fortran IV programming" [87].

The example shows that analysis can reveal errors in a program, and like §6.2 shows how the modelling strategy may be used to produce a more elegant program (in the sense of goto removal).

In this case the program is converted into an equivalent Pascal program. The program could be converted back into a later version of the Fortran programming language (Fortran 77, for example), since such later versions of Fortran contain a "while" loop construct.

Of course, revealing the error in the program requires *a priori* knowledge of the programmer's intent, and as with the transformation to a "structured" program, such analysis and manipulation may be carried out without using functional models.

These examples merely serve to demonstrate the applicability of the modelling strategy to analysis, proof and manipulation. The technique clearly cannot reveal anything that is not already in the program text itself.

6.3.1 The Fortran program

The Fortran program, written in Fortran IV, is taken from page 145 of [87].

It is supposed to perform the following task:

> "A rook is on square *I*1, *J*1. If the path from there to *I*2, *J*2 is unobstructed, set MOVE to .TRUE., and .FALSE. otherwise. (The square *I*2, *J*2 *itself* may or may not be occupied.) Do this only if *I*1 = *I*1 or *J*1 = *J*2; if neither of these is true, set LEGAL to .FALSE.."
> Page 81 of [87].

The program is as follows:

```
      IF (I1.EQ.I2) GO TO 16
      IF (J1.EQ.J2) GO TO 17
      LEGAL = .FALSE.
      GO TO 15
16    J = MINO(J1,J2) + 1
      LIMIT = MAXO(J1,J2)
      IF (LIMIT.EQ.J + 1) GO TO 18
20    IF (BOARD(I1,J) .NE. 0) GO TO 19
      IF (J + 1.EQ.LIMIT) GO TO 18
      J = J + 1
      GO TO 20
17    I = MINO(I1,I2) + 1
      LIMIT = MAXO(I1,I2)
      IF (LIMIT.EQ.I + 1) GO TO 18
21    IF (BOARD(I,J1) .NE. 0) GO TO 19
      IF (I + 1.EQ.LIMIT) GO TO 18
      I = I + 1
      GO TO 21
18    MOVE = .TRUE.
      GO TO 15
19    MOVE = .FALSE.
15
```

6.3.2 The Model

The model is projected onto the final value of the variable "Move".

The original value of this variable is chosen to be *undef*. Should this be a possible final value of the model, it will thus be clear that the program may fail to assign a value to the variable "Move".

The program is modelled by calling the function f(*undef*,i1,i2,j1,j2,Board):

```
fun f(move,i1,i2,j1,j2,Board) =
  if i1 = i2 then f16(i1,i2,j1,j2,Board)
             else if j1=j2 then f17(i1,i2,j1,j2,Board)
                           else f15(move) ;

fun f16(i1,i2,j1,j2,Board) =
  let val j = min(j1,j2) + 1 in
    let val limit = max(j1,j2) in
      if limit = j+1 then f18
                     else f20(i1,j,Board) ;

fun f20(i1,j,Board) =
  if Board(i1,j) <> 0
     then f19
     else if j+1 = limit
             then f18
             else let val j = j+1 in f20(i1,j,Board);

fun f17(i1,i2,j1,j2,Board) =
  let val i = min(i1,i2) + 1 in
    let val limit = max(i1,i2) in
      if limit = i+1 then f18
                     else f21(j1,i,Board) ;

fun f21(j1,i,Board) =
  if Board(i,j1) <> 0
     then f19
     else if i+1 = limit
             then f18
             else let val i = i+1 in f21(j1,i,Board) ;
fun f18 = f15(true) ;
fun f19 = f15(false) ;
fun f15(move) = move ;
```

6.3.3 Manipulation

It may be thought, on first inspection, that the model is in fact more complex than the original program! This would certainly be a likely observation if the reader is familiar with Fortran notation and unfamiliar with ML notation. However, the model *is* easy to *manipulate* as will be shown. The modelling strategy *could* be viewed as automatically producing an, initially, longer program, but one with far simpler transformation rules.

The following discussion is typical of an analysis session that might be conducted by a programmer investigating whether the program meets is specification.

6.3.3.1 First Manipulations

The following manipulations are all fairly trivial and can, of course, be automated, so that the programmer may simply "press buttons" marked "remove constant parameters", "maximal substitute let abstractions" and "unfold functions ... ". For details of automated functional reasoning see [40].

The identifiers "i1", "i2", "j1", "j2" and "Board" are not changed in any model functions, and so can be treated as constants.

Identifiers introduced by let abstraction and functions "f18" and "f19" are unfolded.

These manipulations lead to the following model:

```
fun f = if i1=i2 then f16
                 else if j1=j2 then f17
                               else undef ;

fun f16 = if min(j1,j2)+2 = max(j1,j2)
          then true
          else f20(min(j1,j2)+1,max(j1,j2)) ;

fun f17 = if min(i1,i2)+2 = max(i1,i2)
          then true
          else f21(min(i1,i2)+1,max(i1,i2)) ;


fun f20(j,limit) = if Board(i1,j) <> 0
                   then false
                   else if j+1 = limit then true
                                       else f20(j+1,limit) ;

fun f21(i,limit) = if Board(i,j1) <> 0
                   then false
                   else if i+1 = limit then true
                                       else f21(i+1,limit) ;
```

6.3.3.2 Unfolding

Next the function calls "f16" and "f17" are unfolded in the function "f".

```
fun f = if i1=i2 then if min(j1,j2)+2 = max(j1,j2)
                      then true
                      else f20(min(j1,j2)+1,max(j1,j2))
   else if j1=j2 then if min(i1,i2)+2 = max(i1,i2)
                      then true
                      else f21(min(i1,i2)+1,max(i1,i2))

   else undef ;

fun f20(j,limit) = if Board(i1,j) <> 0
                   then false
                   else if j+1 = limit then true
                                       else f20(j+1,limit) ;

fun f21(i,limit) = if Board(i,j1) <> 0
                   then false
                   else if i+1 = limit then true
                                       else f21(i+1,limit) ;
```

6.3.4 Errors in the Program

The specification does not say what the program should do to the variable "move" in the case where the move is "illegal" (that is, not in a straight horizontal or vertical line).

It is clear from the model that in this situation, the model returns "*undef*", that is, the variable move is not assigned a value.

Strictly speaking, this is *not* an error : it does not contradict the specification, however, it is not desirable to leave the value of this variable undefined. A suitable value would be "false", since a move cannot be made by a rook if it is not horizontal or vertical.

There is also an error in the program in the boolean expression in "f". This *does* contradict the specification.

The test:

```
"if min(j1,j2)+2 = max(j1,j2)
    then true"
```

causes the value of "move" to be true even if there *is* an intervening, occupied, square on the board.

The test should be:

```
"if min(j1,j2)+1 = max(j1,j2)
    then true"
```

This also applies to the test for the case where j1=j2.

These observations lead to the model function, "f", being rewritten. It is then possible to consider the conversion of the model back into a procedural programming notation.

The corrected program is:

```
fun f = if i1=i2 then if min(j1,j2)+1 = max(j1,j2)
                        then true
                        else f20(min(j1,j2)+1,max(j1,j2))
   else if j1=j2 then if min(i1,i2)+1 = max(i1,i2)
                        then true
                        else f21(min(i1,i2)+1,max(i1,i2))

   else false ;

fun f20(j,limit) = if Board(i1,j) <> 0
                      then false
                      else if j+1 = limit then true
                                          else f20(j+1,limit) ;

fun f21(i,limit) = if Board(i,j1) <> 0
                      then false
                      else if i+1 = limit then true
                                          else f21(i+1,limit) ;
```

6.3.5 Conversion Back into a Procedural Notation

The functions "f20" and "f21" can be converted into a "while" loop according to the strategy described in chapter 5:

6.3.5.1 Termination condition

The termination condition for the function "f20" is:

```
Board(i1,j) <> 0   or   j+1 = limit
```

Thus to convert this function into a "while" loop it is be necessary to negate this condition to form the loop's boolean expression.

```
   not(Board(i1,j) <> 0  or   j+1  = limit)
 = not(Board(i1,j) <> 0)  and  not(j+1 = limit)
 = Board(i1,j) = 0 and j+1 <> limit
```

The while loop produced is:

```
while (Board(i1,j) = 0) and (j+1 <> limit) do j := j+1 ;
if Board(i1,j) = 0 then move := true else move := false ;
```

6.3.5.2 The model function f21

A very similar strategy leads to the following "while" loop for f21:

```
while (Board(i,j1) = 0) and (j+1 <> limit) do i := i+1 ;
if Board(i,j1) = 0 then move := true else move := false ;
```

6.3.5.3 The Model Function f

The model function "f" is a conditional choice between these two "while" loops, leading to a procedural program:

```
if i1=i2
   then if min(j1,j2)+1 = max(j1,j2)
           then move := true
           else
           begin
           j := min(j1,j2)+1 ;
           limit := max(j1,j2) ;
           while (Board(i1,j) = 0) and (j+1 <> limit)
              do j := j+1 ;
           if Board(i1,j) = 0 then move := true
                              else move := false
           end

else if j1=j2
        then if min(i1,i2)+1 = max(i1,i2)
                then move := true
                else
                begin
                i := min(i1,i2)+1 ;
                limit := max(i1,i2) ;
                while (Board(i,j1) = 0) and (j+1 <> limit)
                   do i := i+1 ;
                if Board(i,j1) = 0 then move := true
                                   else move := false
                end
else move := false ;
```

6.3.6 Different Manipulations Give Rise to a Different Procedural Programs

A Pascal programmer would, of course, instantly notice the similarity between the two "while" loops produced, and seek to form one "while" loop that captured the effect of both.

However, as this is an exposition of the modelling strategy, the same approach will be taken in analysing the functions "f20" and "f21", which will lead to one function, which can then be automatically converted into "while" loop.

The function "g" is introduced:

```
fun g(i,j,limiti,limitj,addi,addj) =
  if Board(i,j) <> 0
     then false
     else if (i+1 = limiti) or (j+1 = limitj)
             then true
             else g(i+addi,j+addj,limiti,limitj,addi,addj) ;
```

Now

$$f_{20}(j,limit) = g(i1,j,i1,limit,0,1)$$

and

$$f_{21}(i,limit) = g(i,j1,limit,j1,1,0)$$

Proof

It is trivial to prove these identities in the functional notation: partial evaluation of g(i1,j,i1,limit,0,1) gives:

```
g(i1,j,i1,limit,0,1) =
  if Board(i1,j) <> 0
     then false
     else if (i1+1 = i1) or (j+1 = limit)
             then true
             else g(i1,j+1,i1,limit,0,1) ;
```

Now since i1+1 = i1 ≡ false, this can be written:

```
g(i1,j,i1,limit,0,1) =
  if Board(i1,j) <> 0
     then false
     else if j+1 = limit
             then true
             else g(i1,j+1,i1,limit,0,1) ;
```

Clearly therefore this call to "g" returns an identical value to the call "f21(j,limit)".

A very similar argument shows that f20(j,limit) is equivalent to g(i1,j,i1,limit,0,1).

Notice that the functional notation is ideally suited to the introduction of functions in this manner. To achieve the same "certainty" in the procedural notation, the programmer might well find themselves embroiled in an proof using the Axiomatic method.

In the functional notation all that is required is to partially evaluate the function for some particular choice of arguments.

Using the new function "g" and the two identities for the calls to the function "f20" and "f21", the model can be equivalently written:

```
fun f = if i1=i2
        then if min(j1, j2)+1 = max(j1,j2)
                then true
                else g(i1,min(j1,j2)+1,i1,max(j1,j2),0,1)
    else if j1=j2
        then if min(i1, i2)+1 = max(i1,i2)
                then true
                else g(min(i1,i2)+1, j1,max(i1,i2),j1,1,0)
    else false ;

fun g(i,j,limiti,limitj,addi,addj) =
   if Board(i,j) <> 0
      then false
      else if (i+1 = limiti) or (j+1 = limitj)
              then true
              else g(i+addi,j+addj,limiti,limitj,addi,addj) ;
```

6.3.6.1 Conversion to the Procedural Notation

Converting this version of the functional model into Pascal gives rise to a Pascal program with only one while loop. This is the program that the Pascal programmer might have produced having seen the program produced in §6.3.5.3, however, it should be stressed that this new Pascal program has been proved to be equivalent to the original program.

6.3.6.1.1 Converting "g" into a "while" Loop

The termination condition for the function "g" is:

```
(Board(i,j) <> 0)  or  (i+1 = limiti)  or  (j+1 = limitj)
```

The loop produced is:

```
while (Board(i, j) = 0) and (i+1 <> limit1) and (j+1 <> limitj)
   do begin i := i+addi ; j := j + addj end ;
if Board(i, j) = 0 then move := true else move := false
```

6.3.6.1.2 Functions Can be Used in Pascal

Since the function "g" does not return a tuple but merely returns a boolean value, it is an ideal candidate to be converted back into a Pascal function:

```
function g(i,j,limiti,limitj,addi,addj : integer) : integer ;
begin
   while (Board(i, j) = 0) and (i+1 <> limit1) and (j+1 <> limitj)
      do begin i := i+addi ; j := j + addj end ;
   if Board(i,j) = 0 then g := true else g := false
end ;
```

Of course, there was, in fact, no need to convert the recursion equation for "g" into a "while" loop; Pascal call-semantics allows for recursion. However, many programmers would not be pleased to find goto statements removed, only to be replaced by recursive functions.

The whole Pascal program can be written as:

```
(* auxiliary function definition *)
function g(i,j,limiti,limitj,addi,addj:integer) : integer;
begin
  while (Board(i,j)=0) and (i+1<>limit1) and (j+1<>limitj)
     do begin i := i+addi ; j := j + addj end ;
  if Board(i,j) = 0 then g := true else g := false
end ;

(* Main program *)
   if i1=i2
      then if min(j1,j2)+1 = max(j1,j2)
                then move := true
                else
             move := g(i1,min(j1,j2)+1,i1,max(j1,j2),0,1)
else if j1=j2
        then if min(i1,i2)+1 = max(i1,i2)
                  then move := true
                  else
               move := g(min(i1,i2)+1,j1,max(i1,i2),j1,1,0)
else move := false ;
```

6.3.7 Concluding Remarks

The two Pascal programs produced here should be equivalent. "Should", that is, if the fold/unfold manipulation rules have been followed correctly, and conversion back into procedural notation is free from error.

However, it should be noted that the Pascal programs will *not* produce the same effect when executed as the Fortran program would.

This is not simply because the program has been "corrected", but because a variable, "legal", has been omitted in the choice of abstractive projection.

Clearly, to obtain a program with *identical* behaviour, but differing structure, using functional models, it is necessary to choose *all* affected variables as the result onto which the model function is projected.

In this case the programs produced guarantee to assign identical values to the variable "move".

The original program and the final Pascal version produced by modelling are as follows:

```
      IF (I1.EQ.I2) GO TO 16
      IF (J1.EQ.J2) GO TO 17
      LEGAL = .FALSE.
      GO TO 15
16    J = MINO(J1,J2) + 1
      LIMIT = MAXO(J1,J2)
      IF (LIMIT.EQ.J + 1) GO TO 18
20    IF (BOARD(I1,J) .NE. 0) GO TO 19
      IF (J + 1.EQ.LIMIT) GO TO 18
      J = J + 1
      GO TO 20
17    I = MINO(I1,I2) + 1
      LIMIT = MAXO(I1,I2)
      IF (LIMIT.EQ.I + 1) GO TO 18
21    IF (BOARD(I,J1) .NE. 0) GO TO 19
      IF (I + 1.EQ.LIMIT) GO TO 18
      I = I + 1
      GO TO 21
18    MOVE = .TRUE.
      GO TO 15
19    MOVE = .FALSE.
15
```

Pascal version:

```
(* auxiliary function definition *)
function g(i,j,limiti,limitj,addi,addj:integer) : integer;
begin
  while (Board(i,j)=0) and (i+1<>limit1) and (j+1<>limitj)
     do begin i := i+addi ; j := j + addj end ;
  if Board(i,j) = 0 then g := true else g := false
end ;

(* Main program *)
    if i1=i2
       then if min(j1,j2)+1 = max(j1,j2)
                 then move := true
                 else
              move := g(i1,min(j1,j2)+1,i1,max(j1,j2),0,1)
else if j1=j2
       then if min(i1,i2)+1 = max(i1,i2)
                 then move := true
                 else
              move := g(min(i1,i2)+1,j1,max(i1,i2),j1,1,0)
else move := false ;
```

6.4 Proof Construction for a Modula-2 Program

In this example a program is modelled and properties of the program are proved. The analysis reveals some possible input values which might cause the program to crash. This possibility is removed leading to a more robust version of the program.

The properties which are proved for the program are of quite a general character and could be used as a *specification* of the features offered and limits enforced by the robust version of the program.

The program, which implements a mailing list, is taken from the book "Modula-2 Made Easy" [67]. Models are given for each procedure in the program, and are used to analyse the properties of the program.

From the proofs of properties for individual procedures it is possible to construct more general proofs about the properties of the program as a whole. These proofs are called "Global Assertions", and refer to the compactness of storage used and the behaviour of the program when the list is full and/or contains duplicates.

6.4.1 The Program

```
MODULE Mlist; (* a simple mailing list program
                 that uses an array of RECORDS  *)


FROM InOut IMPORT Read, Write, WriteString, WriteLn,
                  WriteCard, ReadCard, ReadString, EOL;

FROM Strings IMPORT CompareStr;


CONST
  LSIZE = 100;


TYPE
  ADDR = RECORD
      name: ARRAY [0..30] OF CHAR;
      street: ARRAY [0..30] OF CHAR;
      city: ARRAY [0..30] OF CHAR;
      state: ARRAY [0..3] OF CHAR;
      zip: ARRAY [0..10] OF CHAR;
  END;

VAR
  mlist: ARRAY [0..LSIZE] OF ADDR; (* array of addresses *)
  choice: CARDINAL;


PROCEDURE Gets(VAR a:ARRAY OF CHAR);
  CONST
    BS = 8;    (* backspace *)

  VAR
    ch: CHAR;
    i:CARDINAL;
  BEGIN
    i:=0;
    REPEAT
      Read(ch);
      Write(ch);
      IF ORD(ch)=BS THEN i:=i-1;    (* is backspace *)
      ELSIF (ch<>EOL) AND (i<HIGH(a)) THEN
        a[i]:=ch;
        i:=i+1;
      END;
    UNTIL (ch=EOL) OR (i=HIGH(a));
    a[i]:=CHR(0);    (* all strings and in 0 *)
END Gets;
```

```
PROCEDURE Puts(s:ARRAY OF CHAR);
  BEGIN
    WriteString(s);
    WriteLn;
  END Puts;


PROCEDURE Menu():CARDINAL;
  VAR
    ch:CARDINAL;
  BEGIN
    Puts('      1. Enter an address');
    Puts('      2. Delete an address');
    Puts('      3. Find an address');
    Puts('      4. List all addresses');
    Puts('      7. Quit');
  REPEAT
    WriteString('Enter  Choice:  ');
    ReadCard(ch);
    WriteLn;
  UNTIL (ch>=1) AND (ch<=7);
  WriteLn;
  RETURN ch;
END Menu;


PROCEDURE GetEmpty(): INTEGER ; (* returns next empty
                          Location in list, -1 if full *)
  VAR
    i:CARDINAL;
BEGIN
  FOR i:=0 TO LSIZE DO
    IF  CompareStr(mlist[i].name,"")=0  THEN
      RETURN i;    (* is an empty location *)
    END;
  END;
  RETURN -1; (* list full *)
END GetEmpty;
```

```
PROCEDURE Enter(i: INTEGER);  (* enter a name into the list *)
BEGIN
  (* if i = -1 then find new slot; otherwise modify
     existing entry *)

  IF i=(-1) THEN i:=GetEmpty(); END;
  IF i<>(-1) THEN
    WriteString('Enter  name:  ');
    Gets(mlist[i].name);
    WriteString('Enter  street:  ');
    Gets(mlist[i].street);
    WriteString('Enter  city:  ');
    Gets(mlist[i].city);
    WriteString('Enter  state:  ');
    Gets(mlist[i].state);
    WriteString('Enter  zip:  ');
    Gets(mlist[i].zip);  WriteLn;  WriteLn;
  END
END Enter;


PROCEDURE Display(ml: ADDR);
BEGIN
  Puts(ml.name);
  Puts(ml.street);
  Puts(ml.city);
  Puts(ml.state);
  Puts(ml.zip);
  WriteLn;  WriteLn;
END  Display;


PROCEDURE List;    (* display the entire list *)
  VAR
    i:CARDINAL;
BEGIN
  FOR i:=0 TO LSIZE DO
    IF  CompareStr(mlist[i].name,"")<>0  THEN
      Display(mlist[i]);
    END
  END
END List;


PROCEDURE Find():INTEGER;  (* return the index of a name *)
  VAR
    f : CARDINAL;
    s : ARRAY [0..30] OF CHAR;
BEGIN
  WriteString('Enter  name  to  find:  ');
  Gets(s);
  FOR f:=0 TO LSIZE DO
    IF CompareStr(s,mlist[f].name)=0 THEN RETURN f END;
  END;
  RETURN -1;  (* not found *)
END Find;
```

```
PROCEDURE Locate;  (* display an address based
                      on the name field *)
  VAR
    i: INTEGER;
BEGIN
  i:=Find(); (* find the name *)
  IF i<>(-1) THEN Display(mlist[i]); END;
END Locate;


PROCEDURE Delete; (* remove an address based
                     on the name field *)
  VAR
    i: INTEGER;

BEGIN
  i:=Find(); (* find the name *)
  IF i<>(-1) THEN
    mlist[i].name:=""; (* mark as empty *)
  END;
END Delete;


PROCEDURE Init; (* initialize list *)
  VAR
    t:CARDINAL;
BEGIN
  (* use a null string in name field to indicate an
     empty RECORD *)
  FOR t:=0 TO LSIZE DO
    mlist[t].name:="";
  END;

END Init;


BEGIN
  Init;   (* prepare the list *)
  REPEAT
    choice:=  Menu();
    CASE choice OF
      1: Enter(-1) ; (* new entry *)
      2:  Delete ;
      3:  Locate ;
      4:  List ;
      5:  ;
      6:  ;
      7:  Puts('program  completed');
    END;
  UNTIL choice=7;

END Mlist.
```

6.4.2 Auxiliary Functions

Some auxiliary functions are introduced to model access to data structures.

The particular choices made here have little impact upon the model produced and the kind of proofs that are constructed.

A tuple is used to model the record structure, and a list to model the array structure.

Two list access functions are used: *s*, the list selection function, and *Update*, the list update function.

```
fun s( i ,L) = if i=1 then hd(L) else s(i-1,tl(L)) ;
fun Update(L, i ,e) =
      if i=1 then [e] else hd(L)::Update(tl(L), i-1,e) ;

Axiom A1 For Update :
      ∀L,e.∀i>0. i≠x ⇒ ( s(Update(L,x,e),i) = s(L,i) ∧
                          s(Update(L,x,e),x) = e )


fun PeelOff(Inp) = (s(1,Inp),s(2,Inp),s(3,Inp),
                    s(4,Inp),s(5,Inp));

    fun Name(ATuple) = ATuple↓1 ;
    fun Street(ATuple) = ATuple↓2 ;
    fun City(ATuple) = ATuple↓3 ;
    fun State(ATuple) = ATuple↓4 ;
    fun Zip(ATuple) = ATuple↓5 ;
```

Where ↓ is the infix tuple-selection function.

6.4.3 The Model

In the model some substitution has been performed, and input of strings is assumed to be correct. It is only the storage and retrieval of entries into the mailing list that are analysed, thus the model is projected onto the value of the mailing list's global array structure, "mlist".

```
fun GetEmpty(m) =
    let fun f(result,m,p) =
              if p > LSIZE
                 then result
                 else if not(Defined(m(p)))
                         then p
                         else f(result,m,p+1)
      in
    f(-1,m,0) ;


fun Enter(i,m,Inp) =
    let val i' = if i=-1 then GetEmpty(m)
                         else i                    in
      if i' ≠ -1 then Update(m,i',PeelOff(Inp))
                 else m ;


fun List(m) =
    let fun f(p,m,L) =
              if p > LSIZE
                 then L
                 else if Defined(m(p))
                         then f(p+1,m,L<>[m(p)])
                         else f(p+1,m,L)
      in
    f(0,m,[]) ;


fun Find(m,Inp) =
    let fun g(m,f,s,result) =
              if f > LSIZE
                 then result
                 else if s = Name(m(f))
                         then g(m,LSIZE+1,s,f)
                         else g(m,f+1,s,result)
     in
    g(m,0,hd(Inp),-1) ;


fun Locate(m,Inp) =
    let val i = Find(m,Inp) in
      if i ≠ -1 then m(i)
                else void ;


fun Delete(m,Inp) =
    let val i = Find(m,Inp) in
      if i ≠ -1
         then Update(m,i,("",Street(m),City(m),State(m),Zip(m)))
         else m ;
```

The Model for the Mailing List Program

6.4.4 Assertions About Model Functions

Defined(x) ≡ Name(x) ≠ ""

FULL(m) ≡ ∀x.0≤x≤LSIZE.Defined(m(x))

GetEmpty

A1 ∀m.FULL(m) ⇒ GetEmpty(m) = -1

A2 ∀m. -1 ≤ GetEmpty(m) ≤ LSIZE

A3 ∀m.~FULL(m) ⇒ Name(m(GetEmpty(m))) = ""

Enter

A1 ~FULL(m) ⇒ ∀x.x≠GetEmpty(m).Enter(-1,m,Inp)(x) = m(x) ∧
Enter(-1,m,Inp)(GetEmpty(m)) = *PeelOff*(Inp)

A2 FULL(m) ⇒ Enter(-1,m,Inp) = m

List

A1 ∀x.0 ≤ x ≤ LSIZE. Defined(m(x)) ⇒ m(x) ∈ List(m)

Find

A1 ∃x.(0 ≤ x ≤ LSIZE ∧ Name(m(x)) = hd(i)) ⇒
0 ≤ Find(m,i) ≤ LSIZE ∧ Name(m(Find(m,i))) = hd(i)

A2 ~∃x.Name(m(x)) = hd(Inp) ⇒ Find(m,Inp) = -1

A3 ~∃y.0≤y<Find(m,i) ∧ Name(m(x)) = hd(i)

Locate

A1 Find(m,Inp) ≠ -1 ⇒ Locate(m,Inp) = m(Find(m,Inp))

A2 Find(m,Inp) = -1 ⇒ Locate(m,Inp) = void

Delete

A1 0 ≤ Find(m,Inp) ≤ LSIZE ⇒
∀x. x≠Find(m,Inp). Delete(m,Inp)(x) = m(x) ∧
Name(Delete(m,Inp)(Find(m,Inp))) = ""

A2 Find(m,Inp) = -1 ⇒ Delete(m,Inp) = m

6.4.5 Proofs for the assertions on Model Functions

In this section proofs are given for the assertions about model functions. Most of the proofs follow by very simple inductive arguments, or directly from other assertions, which have already been proved. In all cases the inductive proof strategy used is structural induction.

6.4.5.1 GetEmpty

```
fun GetEmpty(m) =
    let fun f(result,m,p) =
              if p > LSIZE
                 then result
                 else if not(Defined(m(p)))
                         then p
                         else f(result,m,p+1)
      in
    f(-1,m,0) ;
```

A1 : FULL(m) ⇒ GetEmpty(m) = -1

```
show FULL(m) ⇒ f(r,m,p) = r   so f(-1,m,0) = -1
induction on i = LSIZE - p
base case  i < 0
   i < 0 ⇒ p > LSIZE ⇒ f(r,m,p) = r
inductive step  i ≥ -1
  assume f(r,m,p) = r
  show   f(r,m,p-1) = r
   f(r,m,p-1) = f(r,m,p)     (FULL(m) ⇒ Name(m(p-1)) ≠ "")
              = r            (Inductive Hypothesis)
```

A2 : -1 ≤ GetEmpty(m) ≤ LSIZE

```
show  r < p ≤ LSIZE ⇒ r ≤ f(r,m,p) ≤ LSIZE
  so  -1 ≤ f(-1,m,0) ≤ LSIZE
induction on i = LSIZE - p
base case i < 0
   i < 0 ⇒ p > LSIZE ⇒ f(r,m,p) = r
inductive step i ≥ -1
  assume r ≤ f(r,m,p) ≤ LSIZE
  show   r ≤ f(r,m,p-1) ≤ LSIZE
f(r,m,p-1) has two cases :
          either ~Defined(m(p-1)) ⇒ f(r,m,p-1) = p-1
             in which case i ≥ 0 ⇒ p ≤ LSIZE ⇒ p-1 < LSIZE
              or Defined(m(p-1)) ⇒ f(r,m,p-1) = f(r,m,p)
             in which case the assertion is proved by Ind. Hyp.
```

A3 : ~FULL(m) ⇒ Name(m(GetEmpty(m))) = ""

```
show ∃x.(m(x) = "" ∧ p ≤ x ≤ LSIZE) ⇒ Name(m(f(r,m,p))) = ""
so ~FULL(m) ⇒ Name(m(f(-1,m,0))) = ""
induction on i = x - p  where m(x) = "" ∧ p ≤ x ≤ LSIZE
base case i = 0
   i = 0 ⇒ x = p ⇒ f(r,m,p) = p = x
inductive step i ≥ 0
   assume Name(m(f(r,m,p))) = ""
   show   Name(m(f(r,m,p-1))) = ""
f(r,m,p-1) has two cases
          either ~Defined(m(p-1)) ⇒ f(r,m,p-1) = p-1
          or Defined(m(p-1)) ⇒ f(r,m,p)  (Ind. Hyp.)
```

6.4.5.2 Enter

```
fun Enter(i,m,Inp) =
    let val i' = if i=-1 then GetEmpty(m)
                         else i                in
      if i' ≠ -1 then Update(m,i',PeelOff(Inp))
                 else m ;
```

Substituting -1 for i in the definition of Enter gives :

```
fun Enter(-1,m,Inp) = let val i' = GetEmpty(m) in
                        if i' ≠ -1 then Update(m,i',PeelOff(Inp))
                                   else m
```

```
A1   ~FULL(m) ⇒ ∀x.x≠GetEmpty(m).Enter(-1,m,Inp)(x) = m(x)   ∧
                Enter(-1,m,Inp)(GetEmpty(m)) = PeelOff(Inp)
```

By maximal substitution in Enter and the axiom A1 of update

A2 FULL(m) ⇒ Enter(-1,m,Inp) = m

```
FULL(m) ⇒ GetEmpty(m) = -1      by A1 for GetEmpty
        ⇒ Enter(-1,m,Inp) = m   by maximal substitution
```

6.4.5.3 List

```
fun List(m) =
    let fun f(p,m,L) =
              if p > LSIZE
                 then L
                 else if Defined(m(p))
                         then f(p+1,m,L<>[m(p)])
                         else f(p+1,m,L)
      in
    f(0,m,[]) ;
```

A1 : $\forall x.0 \le x \le LSIZE \wedge Defined(m(x)) \Rightarrow m(x) \in List(m)$

LEMMA : $\forall p.p \le LSIZE \Rightarrow L_1 <> f(p,m,L_2) = f(p,m,L_1 <> L_2)$

proof :

induction on i = LSIZE - p

base case i = -1

i = -1 ⇒ p = LSIZE + 1

$f(p,m,L_1 <> L_2) = L_1 <> L_2$

$L_1 <> f(p,m,L_2) = L_1 <> L_2$

inductive step

assume $f(p,m,L_1 <> L_2) = L_1 <> f(p,m,L_2)$

show $f(p-1,m,L_1 <> L_2) = L_1 <> f(p-1,m,L_2)$

$f(p-1,m,L_1 <> L_2)$ has two cases :

either Defined(m(p-1))

$f(p-1,m,L_1 <> L_2) = f(p,m,(L_1 <> L_2) <> [m(p)])$
$= f(p,m,L_1 <> (L_2 <> [m(p)]))$
$= L_1 <> f(p,m,L_2 <> [m(p)])$

$L_1 <> f(p-1,m,L_2) = L_1 <> f(p,m,L_2 <> [m(p)])$

or ~Defined(m(p-1))

$f(p-1,m,L_1 <> L_2) = f(p,m,L_1 <> L_2)$
$= L_1 <> f(p,m,L_2)$

$L_1 <> f(p-1,m,L_2) = L_1 <> f(p,m,L_2)$

Using the lemma the proof of A1 for List can proceed.

```
A1 : ∀x.0≤x≤LSIZE ∧ Defined(m(x)) ⇒ m(x) ∈ List(m)
to do this show  ∀x.p≤x≤LSIZE ∧ Defined(m(x)) ⇒ m(x)∈f(p,m,L)
which means f(0,m,[]) satisfies A1
induction on i = LSIZE - p
base case i = -1
   ∀x.p≤x≤LSIZE ∧ Defined(m(x)) ⇒ m(x) ∈ f(p,m,L) vacuously
   satisfied.
inductive step i ≥ -1
   assume ∀x.p≤x≤LSIZE ∧ Defined(m(x)) ⇒ m(x) ∈ f(p,m,L)
   show   ∀x.p-1≤x≤LSIZE ∧ Defined(m(x)) ⇒ m(x) ∈ f(p-1,m,L)
f(p-1,m,L) has two cases
   either Defined(m(p-1))
   f(p-1,m,L) = f(p,m,L <> [m(p-1)])
              = L <> [m(p-1)] <> f(p,m,[])     by Lemma
so ∀x.p≤x≤LSIZE ∧ Defined(m(x)) ⇒ m(x)∈f(p-1,m,L)  by Ind. Hyp.
and Defined(m(p-1)) and m(p-1) ∈ f(p-1,m,L)
so ∀x.p-1≤x≤LSIZE ∧ Defined(m(x)) ⇒ m(x) ∈ f(p-1,m,L)

   or ~Defined(m(p-1))
   f(p-1,m,L) = f(p,m,L)
so ∀x.p≤x≤LSIZE ∧ Defined(m(x)) ⇒ m(x)∈f(p-1,m,L)   by Ind. Hyp.
and ~Defined(m(p-1)) and m(p-1) ∉ f(p-1,m,L)
so ∀x.p-1≤x≤LSIZE ∧ Defined(m(x)) ⇒ m(x) ∈ f(p-1,m,L)
```

6.4.5.4 Find

```
fun Find(m,Inp) =
    let fun g(m,f,s,result) =
              if f > LSIZE
                 then result
                 else if s = Name(m(f))
                         then g(m,LSIZE+1,s,f)
                         else g(m,f+1,s,result)
     in
    g(m,0,hd(Inp),-1) ;
```

```
A1 : ∃x.(0 ≤ x ≤ LSIZE ∧ Name(m(x)) = hd(i)) ⇒
     0 ≤ Find(m,i) ≤ LSIZE ∧
     Name(m(Find(m,i))) = hd(i)
```

to do this show $\exists x.(0 \le x \le LSIZE \wedge Name(m(x)) = s \wedge 0 \le f \le x) \Rightarrow$
$0 \le g(m,f,s,r) \le LSIZE \wedge Name(m(g(m,f,s,r))) = s$
induction on i = x-f where $0 \le x \le LSIZE \wedge Name(m(x)) = s$
base case i = 0
g(m,f,s,r) = g(m,LSIZE+1,s,f) = f = x
inductive step $i \ge 0$
$i > 0 \Rightarrow 0 \le f < x \le LSIZE$
assume g(m,f,s,r) satisfies A1
show g(m,f-1,s,r) satisfies A1
g(m,f-1,s,r) has two cases
either $s = Name(m(f-1)) \Rightarrow$ g(m,f-1,s,r) =
g(m,LSIZE+1,s,f-1) =
f-1 (f-1 satisfies A1)
or $s \neq Name(m(f-1)) \Rightarrow$ g(m,f-1,s,r) = g(m,f,s,r)
(which satisfies A1 by the Induction Hypothesis)

A2 : $\sim\exists x.Name(m(x)) = hd(Inp) \Rightarrow Find(m,Inp) = -1$

to do this show $\sim\exists x.Name(m(x)) = s \Rightarrow g(m,f,s,r) = r$
induction on i = LSIZE - f
base case $i < 0$
$i < 0 \Rightarrow f > LSIZE \Rightarrow$ g(m,f,s,r) = r
inductive step $i \ge -1$
$i \ge -1 \Rightarrow f \le LSIZE$
assume g(m,f,s,r) = r
show g(m,f-1,s,r) = r
$\sim\exists x.Name(m(x)) = s \Rightarrow g(m,f-1,s,r) = g(m,f,s,r) = r$
(by Induction Hypothesis)

A3 $\sim\exists y.0 \le y < Find(m,i) \wedge Name(m(x)) = hd(i)$

```
Show ∃y.0≤y≤Find(m,i) ∧ Name(m(x)) = hd(i) ⇒
      Name(m(x)) = Find(m,i)

let k be the index of the first element that equals the element to
be found :
0≤k≤LSIZE. m(k) = hd(Inp) ∧
           Find(m,Inp) = k ∧
           (∀y.0≤y<k.m(y)≠hd(Inp))

show f ≤ k ⇒ g(m,f,hd(Inp),r) = k
induction on i = k-f, i ≥ 0
base case i = 0
i=0 ⇒ k=f
      k=f ⇒ m(f) = hd(Inp)
      k≤LSIZE ⇒ g(m,f,hd(Inp),r)
                = g(m,LSIZE+1,hd(inp),k)
                = k

inductive step i≥0
assume g(m,f,hd(Inp),r) = k
show   g(m,f-1,hd(Inp),r) = k
i≥0 ⇒ 0≤f≤k≤LSIZE
    ⇒ g(m,f-1,hd(Inp),r) =
      if hd(Inp) = Name(m(f)) then g(m,LSIZE+1,hd(Inp),r)
                              else g(m,f,hd(Inp),r)
since ∀y.0≤y<k.m(y)≠hd(Inp),
    g(m,f-1,hd(Inp),r) = g(m,f,hd(Inp),r) = k
```

6.4.5.5 Locate

```
fun Locate(m,Inp) =
    let val i = Find(m,Inp) in
      if i ≠ -1 then m(i)
                else void ;
```

A1	Find(m,Inp) ≠ -1 ⇒ Locate(m,Inp) = m(Find(m,Inp))

By maximal substitution in Locate and substitution of true for Find(m,Inp) ≠ -1.

A2	Find(m,Inp) = -1 ⇒ Locate(m,Inp) = void

By maximal substitution in Locate and substitution of false for Find(m,Inp) ≠ -1.

6.4.5.6 Delete

```
fun Delete(m,Inp) =
    let val  i = Find(m,Inp) in
      if i ≠ -1
         then Update(m,i,("",Street(m),City(m),State(m)Zip(m)))
         else m ;
```

```
A1   0 ≤ Find(m, Inp) ≤ LSIZE ⇒
     ∀x. x≠Find(m, Inp). Delete(m,Inp)(x) = m(x)   ∧
     Name(Delete(m,Inp)(Find(m,Inp))) = ""
```

0≤Find(m,Inp)≤LSIZE ⇒
Delete(m,i,("",*Street*(m),*City*(m),*State*(m),*Zip*(m)))

by substitution

⇒ ∀x.x≠Find(m, Imp).Delete(m,Inp)(x) = m(x)
∧ Name(Delete(m,Inp)(Find(m,Inp))) = ""

by axiom A1 for Update

```
A2   Find(m,Inp) = -1 ⇒ Delete(m,Inp) = m
```

By maximal substitution in Delete.

6.4.6 Higher-Level Proofs

The proofs constructed so far concern the properties of individual routines that are used to implement the mailing list program. These proven assertions can be used to construct proofs of a general character about the mailing list as a whole.

Some examples of such "Universal Assertions" are described below:

6.4.6.1 UA1: Compaction

UA1 asserts that the entries in the mailing list are stored contiguously. This is proved by showing that the free-space allocation function "GetEmpty" always returns the *first* free space in the mailing list. Of course, Delete will leave a "gap" in the mailing list, however, the assertion UA1 guarantees that this space will be used up before any of the block of free space at the end of the mailing list array is used.

```
UA1 : (∀m∀x.0≤x<GetEmpty(m) ⇒ Defined(Name(m(x)))) ∧
      ~Defined(Name(m(GetEmpty(m))))
```

The proof for the second half of this conjunction comes from assertion A3 for GetEmpty for all *non-full* mailing lists. The case where the mailing list is full is vacuously satisfied by assertion A1 of GetEmpty. (∀x.0≤x<-1.P(x) is a tautology).

In order to show the first half of the conjunction of UA1 it is sufficient to show that *if* there is another "least, undefined element" *then* it is one and the same as that returned by GetEmpty.

```
Assume ∃x.x≤k ∧ x≥0 ∧ ~Defined(Name(m(x))) ∧
       (∀y.y<x ∧ y≥0 ⇒ Defined(Name(m(y))))                    (1)
    where k = GetEmpty(m)

Show (1) ⇒ x = k.
     To do this show 0≤p≤x ⇒ f(r,m,p) = x
Induction on i = x - p

base case i = 0
i = 0 ⇒ x = p
Now (x ≤ k ≤ LSIZE) ⇒ p ≤ LSIZE (A1 of GetEmpty and (1))
and from (1) ~Defined(Name(m(x))) ⇒ ~Defined(Name(m(p)))
    ~Defined(Name(m(p))) ⇒ f(r,m,p) = p = x.

inductive step
assume : 0≤p+1≤x ⇒ f(r,m,p+1) = x
show   : f(r,m,p) = x
p+1 ≤ x ⇒ p < x
so from (1) f(r,m,p) = f(r,m,p+1) = x by the induction hypothesis.
```

6.4.6.2 UA2: Delete is the Inverse of Enter

This assertion shows that deleting an element from a mailing list to which it has been added leaves the mailing list unaltered.

The assertion is *only* true given certain provisos which will be discussed after the proof has been described.

$$UA2 : \sim\exists x.(Name(m(x)) = hd(Inp1) \wedge hd(Inp1) = hd(Inp2) \wedge \sim Full(m) \Rightarrow Delete(Enter(-1,m,Inp1),Inp2) = m(x)) \quad (2)$$

```
let s = hd(Inp1) = hd(Inp2).
let Find(Enter(-1,m, Inp1),Inp1) = k
(by A1 for Find and (2))
   ~∃x.Name(m(x)) = s ∧ Name(Enter(-1,m,Inp1)k) = s
(and by A1 for Enter)
   Name(Enter(-1,m, Inp1)(GetEmpty(m))) = Name(PeelOff(Inp1)) = s
so Find(Enter(-1,m, Inp1), Inp1) = k = GetEmpty(m)

Now (by A1 for Delete)
   Name(Delete(Enter(-1,m, Inp1), Inp2)(Find(m,Inp1))) = "" =
   Name(m(GetEmpty(m))).                                          (2)
And (From A1 Enter and A1 Delete)
   ∀x. x≠GetEmpty(m) ⇒ Name(Delete(Enter(-1,m,Inp1), Inp2)x) =
      Name(Enter(-1,m,Inp1)x) = m(x)                              (3)
Conjoining (2) and (3) gives UA2.
```

6.4.6.3 A Problem Case for "Find"

As shown by assertion UA1, the function "GetEmpty" finds the *first* available space to enter, and by assertion A3 for GetEmpty, which is:

$\forall m.\sim FULL(m) \Rightarrow Name(m(GetEmpty(m))) = ""$

the "name part" of this space is "".

Now, by assertion A3 for the function "Find", Find(m,Inp), finds the first element of the mailing list, x, that satisfies

$Name(m(x)) = hd(Inp)$.

So

$\sim FULL(m) \Rightarrow Find(m,"") = GetEmpty(m)$.

This is simply a reflection of the fact that "" is used in the name field to indicate an unused space. So "" should be disallowed as a valid string, or "rubbish" may be printed on the screen.

6.4.7 Discussion

The restrictions on UA2 are that the mailing list must not be full and that the item added to the mailing list must not already be a member of the list. This reflects the implementation restrictions of the mailing list program.

Specifically, the program cannot be expected to behave when the entire space allocated to the array, which stores the elements of the list, is used up. It also does not behave well when duplicate names are included in the mailing list.

Now, *no checks* are included in the original program to detect these two situations.

Also, the Null string, (""), is used to denote an empty entry in the list, so it should not be considered to be a valid name for searching or addition.

Adding checks to the program to detect and report on these problem cases improves the robustness of the mailing list program.

6.4.8 The Modified Mailing List Program

The mailing list program has been modified to trap the errors uncovered by the analysis conducted above.

```
MODULE Mlist; (* a simple mailing list program
                 that uses an array of RECORDS  *)
               (* Update 1991 Mark Harman *)

FROM InOut IMPORT Read, Write, WriteString, WriteLn,
                  WriteCard, ReadCard, ReadString, EOL;

FROM Strings IMPORT CompareStr;


CONST
  LSIZE = 100;


TYPE
NameType = ARRAY [0..30] OF CHAR ;
ADDR = RECORD
    name: NameType;
    street: ARRAY [0..30] OF CHAR;
    city: ARRAY [0..30] OF CHAR;
    state: ARRAY [0..3] OF CHAR;
    zip: ARRAY [0..10] OF CHAR;
END;
```

```
VAR
  mlist: ARRAY [0..LSIZE] OF ADDR; (* array of addresses *)
  choice: CARDINAL;



PROCEDURE Duplicate(s:NameType) : BOOLEAN ;
(* returns true iff the name s already
   exists in the mailing  list *)

VAR f : INTEGER ;

BEGIN
  FOR f:=0 TO LSIZE DO
    IF CompareStr(s,mlist[f].name)=0 THEN RETURN TRUE END;
  END;
  RETURN FALSE
END Duplicate ;



PROCEDURE Gets(VAR a:ARRAY OF CHAR);
  CONST
    BS = 8;     (* backspace *)

  VAR
    ch: CHAR;
    i:CARDINAL;
  BEGIN
    i:=0;
    REPEAT
      Read(ch);
      Write(ch);
      IF ORD(ch)=BS THEN i:=i-1;     (* is backspace *)
      ELSIF (ch<>EOL) AND (i<HIGH(a)) THEN
        a[i]:=ch;
        i:=i+1;
      END;
      UNTIL (ch=EOL) OR (i=HIGH(a));
      a[i]:=CHR(0);     (* all strings and in 0 *)
  END Gets;



PROCEDURE Puts(s:ARRAY OF CHAR);
  BEGIN
    WriteString(s);
    WriteLn;
END Puts;
```

```
PROCEDURE Menu():CARDINAL;
  VAR
    ch:CARDINAL;
  BEGIN
    Puts('      1. Enter an address');
    Puts('      2. Delete an address');
    Puts('      3. Find an address');
    Puts('      4. List all addresses');
    Puts('      7. Quit');
  REPEAT
    WriteString('Enter  Choice:  ');
    ReadCard(ch);
    WriteLn;
  UNTIL (ch>=1) AND (ch<=7);
  WriteLn;
  RETURN ch;
END Menu;


PROCEDURE GetEmpty(): INTEGER ; (* returns next empty
                        Location in list, -1 if full *)
  VAR
    i:CARDINAL;
BEGIN
  FOR i:=0 TO LSIZE DO
    IF CompareStr(mlist[i].name,"")=0 THEN
      RETURN i;    (* is an empty location *)
    END;
  END;
  RETURN -1; (* list full *)
END GetEmpty;
```

```
PROCEDURE Enter(i: INTEGER);  (* enter a name into the list *)
VAR s : NameType ;
BEGIN
  (* if i = -1 then find new slot; otherwise modify
     existing entry *)
    WriteString('Enter  name:  ');
    REPEAT Gets(s) UNTIL (s[0] <> CHR(0)) ;
    IF  Duplicate(s)
       THEN
       WriteString('Duplicate name, can not add it the the list') ;
       WriteLn
    ELSE
    IF i=(-1) THEN i:=GetEmpty(); END;
    IF   i<>(-1)
       THEN
       mlist[i].name  :=  s  ;
       WriteString('Enter   street:   ');
       Gets(mlist[i].street);
       WriteString('Enter   city:   ');
       Gets(mlist[i].city);
       WriteString('Enter   state:   ');
       Gets(mlist[i].state);
       WriteString('Enter   zip:   ');
       Gets(mlist[i].zip);   WriteLn;   WriteLn

       ELSE
       WriteString('Can not add this name, the mailing  list is full')
    END
END Enter;



PROCEDURE Display(ml: ADDR);
BEGIN
   Puts(ml.name);
   Puts(ml.street);
   Puts(ml.city);
   Puts(ml.state);
   Puts(ml.zip);
   WriteLn;  WriteLn;
END  Display;


PROCEDURE  List;     (* display the entire list *)
   VAR
     i:CARDINAL;
BEGIN
   FOR i:=0 TO LSIZE DO
     IF   CompareStr(mlist[i].name,"")<>0   THEN
       Display(mlist[i]);
     END
   END
END  List;
```

```
PROCEDURE Find():INTEGER;  (* return the index of a name *)
  VAR
    f : CARDINAL;
    s : ARRAY [0..30] OF CHAR;
BEGIN
  WriteString('Enter name to find: ');
  REPEAT Gets(s) UNTIL (s[0] = CHR(0));

  FOR f:=0 TO LSIZE DO
    IF CompareStr(s,mlist[f].name)=0 THEN RETURN f END;
  END;

  RETURN -1;  (* not found *)
END Find;
```

```
PROCEDURE Locate;  (* display an address based
                      on the name field *)
  VAR
    i: INTEGER;
BEGIN
  i:=Find(); (* find the name *)
  IF  i<>(-1)  THEN  Display(mlist[i])
              ELSE WriteString(' Can not find this name ')
  END
END Locate;


PROCEDURE Delete; (* remove an address based
                     on the name field *)
  VAR
    i: INTEGER;

BEGIN
  i:=Find(); (* find the name *)
  IF  i<>(-1) THEN
    mlist[i].name:=""; (* mark as empty *)
  END;
END Delete;


PROCEDURE  Init; (* initialize list *)
  VAR
    t:CARDINAL;
BEGIN
  (* use a null string in name field to indicate an
     empty RECORD *)
  FOR t:=0 TO LSIZE DO
    mlist[t].name:="";
  END;

END  Init;
```

```
BEGIN
  Init;   (* prepare the list *)
  REPEAT
    choice:=  Menu();
    CASE choice OF
      1: Enter(-1) ; (* new entry *)
      2: Delete ;
      3: Locate ;
      4: List ;
      5: ;
      6: ;
      7: Puts('program  completed');
    END;
  UNTIL choice=7;
END Mlist.
```

6.5 The Program "Find"

"Find" is an algorithm invented by Hoare. In his paper "A proof of the program FIND" [33], Hoare shows how a procedural program that implements this algorithm could be constructed and proved simultaneously, using the Axiomatic Method (described in chapter two). As such, the procedural program "FIND", provides a good example for comparing the proof and manipulation possibilities offered by functional modelling with those of the Axiomatic Method.

By converting the manipulated functional model back into a procedural program, it is possible to see how to remove the **"goto"** statement, at the expense of one extra iteration of the main **"while"** loop.

The procedural program "FIND" is written in Algol 68, with comments written in italics. It is a remarkable, compact and efficient algorithm.

```
begin
integer m,n;
m := 1; n := N;
while m < n do
  begin integer r,i,j,w;

    r := A[f]; i := m; j := n;
    while i <= j do
    begin while A[i] < r do i := i + 1;
      while r < A[j] do j := j - 1;
      if i <= j then
      begin w := A[i]; A[i] := A[j]; A[j] := w;
        i := i + 1; j := j - 1;
      end
    end increase i and decrease j;
    if f <= j then n := j
  else if i <= f then m := i
    else go to L
  end reduce middle part;
L:
end Find
```

```
(* f is a constant, as is N *)
fun find(A) = f1(A,1,N) ;

fun f1(A,m,n) = if m < n
                then
                let (A',i,j) = f2(A,m,n,s(A,f)) in
                if f <= j then f1(A',m,j)
                          else if i <= f then f1(A',i,n)
                                         else A'
                else A ;

fun f2(A,i,j,r) = if i <= j
                  then
                  let i' = f3(A,i,r) in
                  let j' = f4(A,j,r) in
                  if i' <= j'
                  then
                  let A' = u(u(A,i',s(A,j')),j',s(A,i')) in
                  f2(A',i'+1,j'-1,r)

                  else
                  (A,i',j')

                  else
                  (A,i,j) ;

fun f3(A,i,r) = if s(A,i) < r then f3(A,i+1,r) else i ;
fun f4(A,j,r) = if r < s(A,j) then f4(A,j-1,r) else j ;
```

```
types

find : list int → list int
f1   : (list int × int × int) → list int
f2   : (list int × int × int × int) → (list int × int × int)
f3   : (list int × int × int) → int
f4   : (list int × int × int) → int
```

6.5.1 Auxiliary Functions Used in the Model

The function "*s*" and "*u*" are the list selection an list update functions.

6.5.2 The Proof

6.5.2.1 Lemma 0

For the functions f3 and f4, two very simply properties are stated, which can been seen to hold from their predicates.

6.5.2.1.1 Lemma 0 (f3)

$$\exists x.\, x \geq i \wedge s(A,x) \geq r \Rightarrow$$
$$f3(A,i,r) = i' \quad \text{s.t.} \quad \forall x.\, i \leq x < i'.\ s(A,x) < r \ \wedge\ s(A,i') \geq r$$

6.5.2.1.2 Lemma 0 (f4)

$$\exists x.\, x \leq j \wedge s(A,x) \leq r \Rightarrow$$
$$f4(A,j,r) = j' \quad \text{s.t.} \quad \forall x.\, j' < x \leq j.\ s(A,x) > r \ \wedge\ s(A,j') \leq r$$

6.5.2.2 Lemma 1

$$f2(A,x,x,s(A,x)) = (A,x+1,x-1)$$

Proof

$x \leq x$
$f3(A,x,s(A,x)) = x$ (since not(s(A,x) < s(A,x))
$f4(A,x,s(A,x)) = x$ (since not(s(A,x) < s(A,x))

So $f2(A,x,x,s(A,x)) =$
$f2(A',x+1,x-1)$
where $A' = u(u(A,x,s(A,x)),x,s(A,x))$
but $not(x+1 \leq x-1) \wedge u(u(A,x,s(A,x)),x,s(A,x)) = A$

So $f2(A,x,x,s(A,x)) = (A,x+1,x-1)$

6.5.2.3 Lemma 2

$$m \leq f \leq n \Rightarrow f1(A,m,n) = f1'(A,m,n)$$

where fun f1' = if m <= n
then
let (A',i,j) = f2(A,m,n,s(A,f)) in
if f <= j then f1'(A',m,j)
else if i <= f then f1'(A',i,n)
else A'
else A

This lemma allows the function f1 to be slightly altered (the predicate "m < n" has been replaced by the relaxed predicate "m ≤ n").

Proof

```
The proof uses partial evaluation of the function f1.
First, note that m ≤ f ≤ n ⇒ ( (m=n) ⇒ (m=n=f) )

Recursive calls to f1 preserve the relationship m≤f≤n :
     f1(A',m,j) where m ≤ f ≤ j
and  f1(A',i,n) where i ≤ f ≤ n

   So, if m=n, then since m≤f≤n, the call to f1'(A,m,n) will
be f1'(A,f,f).
   Substituting these values in the body of f1 gives the
result as :

   let (A',i,j) = f2(A,f,f,s(A,f)) in
   if f <= j then f1(A',f,j)
             else if i <= f then f1(A',i,f)
                            else A'

   by lemma 1  f2(A,f,f,s(A,f)) = (A,f+1,f-1)
   So f1'(A,f,f) = A
   So m ≤ f ≤ n ⇒ f1(A,m,n) = f1'(A,m,n)
```

Using Lemma 2 it is now possible to rewrite the model function f1 as follows :

```
fun f1(A,m,n) = if m <= n
                then
                let (A',i,j) = f2(A,m,n,s(A,f)) in
                if f <= j then f1(A',m,j)
                          else if i <= f then f1(A',i,n)
                                         else f1(A',i,j)
                else A';
```

6.5.2.4 Lemma 3

$$f_1(A,m,n) = f_1''(A,m,n)$$

```
where fun f1"(A,m,n) =
          if m <= n
             then
             let (A',i,j) = f2(A,m,n,s(A,f)) in
             if f <= j then f1"(A',m,j)
                       else if i <= f then f1"(A',i,n)
                                      else f1"(A',i,j)
             else A
```

This lemma allows f_1 to be replaced by f_1'', where f_1'' has only one condition leading to termination (instead of two).

Proof

The proof is by partial evaluation of f'.

The call $f_1''(A',i,j)$ only occurs if both f <= j and i <= f. With i substituted for m and j for n, the recursive call $f_1''(A',i,j) = A'$, since i>j.

Using Lemma 3 it is now possible to rewrite the model function f_1 as follows :

```
fun f1(A,m,n) = if m <= n
                then
                let (A',i,j) = f2(A,m,n,s(A,f)) in
                if f <= j then f1(A',m,j)
                          else if i <= f then f1(A',i,n)
                                         else f1(A',i,j)
                else A';
```

6.5.2.5 Lemma 4

$$\forall ijrA.\, i \le j \wedge r \in \{x \mid x = s(A,y) \wedge i \le y \le j\} \Rightarrow$$
$$f_2(A,i,j,r) = (A',i',j')$$
$$\text{s.t. } \forall xy.\, i \le x < i' \wedge j' < y \le j \Rightarrow s(A',x) \le r \le s(A',y)$$
$$\wedge \quad \forall x.\, x < i \Rightarrow s(A',x) = s(A,x)$$
$$\wedge \quad \forall x.\, x > j \Rightarrow s(A',x) = s(A,x)$$

6.5.2.5.1 Explanation

Let L(n,m] denote all the elements of a list L from, but not including n, up to, and including m.

Let L[n,m) denote all the elements of a list L from, and including n, up to, but excluding m.

Let L[n,m] denote all the elements of a list L from, and including n, up to, and including m.

Lemma 4 says : if $r \in A[i,j]$ then f_2 yields the tuple (A',i',j') where $L[i,i') \le r \le L(j',j]$. The last two conjuncts of the implication dictate that all other elements of the list outside [i,i') and (j',j] are unaffected.

Proof

(by Structural Induction on k = j-i)

base case

```
k=0 ⇒ ( j=i ∧ r=s(A,j))
      substituting A,j,j and s(A,j), as actual parameters, for the
      formal parameters A,i,j and r in f2 respectively.
      gives :

      if  j<=j
          then
          let i' = f3(A,j,s(A,j)) in
          let j' = f3(A,j,s(A,j)) in
          if i' <= j'
              then
              let A' = u(u(A,i',s(A,j')),j',s(A,i')) in
              f2(A',i'+1,j'-1)

    Now
    f3(A,j,s(A,j)) = f4(A,j,s(A,j)) = j          (by substitution)
    and u(u(,A,j,s(A,j)),j,s(A,j)) = A
    so f2(A,j,j,s(A,j)) = f2(A,j+1,j-1) = (A,j+1,j-1)
    ∀xy. j≤x<j+1 ∧ j-1<y≤j ⇒ s(A,x) ≤ s(A,j) ≤ s(A,y) is true
       since it degenerates to s(A,j) ≤ s(A,j) ≤ s(A,j)
    And since A' = A then ∀x.x<i ⇒ s(A',x) = s(A,x) and
    ∀x. x>j ⇒ s(A',x) = s(A,x) are clearly true.
```

inductive hypothesis

```
∀ijrA. i≤j ∧ r ∈ {x|x = s(A,y) ∧ i≤y≤j} ⇒
     f2(A,i,j,r) = (A',i',j')
          s.t. ∀xy. i≤x<i' ∧ j'<y≤j ⇒ s(A',x) ≤ r ≤ s(A',y)
          ∧ ∀x.x<i ⇒ s(A',x) = s(A,x)
          ∧ ∀x.x>j ⇒ s(A',x) = s(A,x)
for j-i = k, k≥0
```

call the assertion D.

show D for j-i = k+1 ⇒ k>0

Now, in the body of f2 :

i≤j is true since k>0

i' = f3(A,i,r) & j' = f4(A,j,r)

There are two possibilities, call them "case 1" and "case 2" :

case 1 : $j' - i' = k$

$j'-i' = k \Rightarrow (j'=j \wedge i'=i) \Rightarrow s(A,i) \geq r \wedge s(A,j) \leq r$

(from the predicate in f3 & f4)

so in this case $f2(A,i,j,r) = f2(A',i+1,j-1,r)$
where $A' = u(u(A,i,s(A,j)),j,s(A,i))$
Now, by the induction hypothesis
$f2(A',i+1,j-1,r) = (A'',p,q)$
s.t. $\forall xy.\ i+1 \leq x < p \wedge q < y \leq j-1 \Rightarrow s(A'',x) \leq r \leq s(A'',y)$
$\wedge\ \forall x.\ x < i+1 \Rightarrow s(A'',x) = s(A',x)$
$\wedge\ \forall x.\ x > j-1 \Rightarrow s(A'',x) = s(A',x)$
also $s(A'',j) = s(A,j)$ & $s(A'',j) = s(A,i)$
so since $s(A,j) \leq r$ & $s(A,i) \geq r$
then $f2(A,i,j,r) = f2(A',i+1,j-1,r) = (A'',p,q)$ s.t.
$\forall xy.\ i \leq x < p \wedge q < y \leq j \Rightarrow s(A'',x) \leq r \leq s(A'',y)$
$\wedge\ \forall x.\ x < i \Rightarrow s(A'',x) = s(A,x)$
$\wedge\ \forall x.\ x > j \Rightarrow s(A'',x) = s(A,x)$

case 2 : $j' - i' < k$

$j'-i' < k$

in which case either $i' > j'$ or $i' \leq j'$

case 2.1 : $i' > j'$
in which case $f2(A,i,j,r) = (A,i',j')$
Now $r \in \{ x \mid x = s(A,y) \wedge i \leq y \leq j \}$
So $f3(A,i,r) = i'$ s.t. $\forall x.\ i \leq x < i'.\ s(A,x) < r \wedge s(A,i') \geq r$
and $f4(A,j,r) = j'$ s.t. $\forall x.\ j' < x \leq j.\ s(A,x) > r \wedge s(A,j') \geq r$

(by lemma 0)

which, together with $i' > j'$ satisfies D.

case 2.2 : $i' \leq j'$
in which case $f2(A,i,j,r) = f2(A',i'+1,j'-1,r)$
Now $f2(A',i'+1,j'-1,r) = (A'',p,q)$ s.t.
$\forall xy.\ i'+1 \leq x < p \wedge q < y \leq j'-1.\ s(A'',x) \leq r \leq s(A,y)$ (by i.h.)
$\wedge\ \forall x.\ x < i'+1 \Rightarrow s(A',x) = s(A'',x)$
$\wedge\ \forall x.\ x > j'-1 \Rightarrow s(A',x) = s(A'',x)$
and from lemma 0 and the properties of u
$s(A',i') = s(A,j) \leq r$ & $s(A',j') = s(A,i) \geq r$
and from lemma 0
$\forall x.\ i \leq x < i'.\ s(A,x) \leq r$ & $\forall x.\ j' < x \leq j.\ s(A,x) \geq r$
Putting these together gives $f2(A,i,j,r) = (A',p,q)$ s.t.
$\forall xy.\ i \leq x < p \wedge q < x \leq j.\ s(A',x) \leq r \leq s(A',y)$
$\wedge\ \forall x.\ x < i \Rightarrow s(A',x) = s(A,x)$
$\wedge\ \forall x.\ x > j \Rightarrow s(A',x) = s(A,x)$

6.5.2.6 The Main Proof

For the purpose of the main proof the function f1 shall be projected onto A, m and n. This allows the values bound to m and n at termination, to be used in an assertion about f1.

```
fun f1(A,m,n) = if m <= n
                then
                let (A',i,j) = f2(A,m,n,s(A,f)) in
                if f <= j then f1(A',m,j)
                          else if i <= f then f1(A',i,n)
                                         else f1(A',i,j)
                else (A,m,n);
```

First prove :

$$m \le f \le n \Rightarrow f_1(A,m,n) = (A',m',n') \text{ s.t.}$$
$$\forall xy.\, m \le x < m' \wedge n' < y \le n.\ s(A',x) \le s(A',f) \le s(A',y) \wedge m > n$$

proof

(by structural induction on k = n-m, k≥0)

base case : k=0 ⇒ m=n=f

substituting m=n=f into the body of f1 gives the result:

```
let (A',i,j) = f2(A,f,f,s(A,f))
             = (A,f+1,f-1)                    (by lemma 1)
∀xy.f≤x<f+1 ∧ f-1<y≤f. s(A,x) ≤ s(A,f) ≤ s(A,y)
reduces to s(A,f) ≤ s(A,f) ≤ s(A,f)
```

inductive hypothesis

$$m \le f \le n \Rightarrow f_1(A,m,n) = (A',m',n') \text{ s.t.}$$
$$\forall xy.\, m \le x < m' \wedge n' < y \le n.\ s(A',x) \le s(A',f) \le s(A',y) \wedge m > n$$
for k = n-m

$k \geq 0 \Rightarrow m \leq n$, substituting into the body of f_1 gives the result :

```
let (A',i,j) = f2(A,m,n,s(A,f)) in
if f <= j then f1(A',m,j)
          else if i <= f then f1(A',i,n)
                         else f1(A',i,j)
```

In this proof, let ϕ be the fth element in order in the array A.
There are three possible cases:

case 1 : f <= j

f ≤ j means that $s(A,f) \leq \phi$, since there are at least as man elements in A which are greater than or equal to ϕ than to $s(A,f)$

So Lemma 4 in this case gives :

$\forall x.\ j<x\leq n.\ \ s(A,x) \geq \phi$

Now by the induction hypothesis

$f_1(A',m,j) = (A'',p,q)$ s.t.

$\forall xy.\ m\leq x<p \wedge q<y\leq j.\ s(A'',x) \leq s(A'',f) \leq s(A'',y) \wedge p>q$

$\Rightarrow \forall xy.\ m\leq x<p \wedge q<y\leq j.\ s(A'',x) \leq \phi \leq s(A'',y) \wedge p>q$

Together with lemma 4 gives :

$f_1(A',m,n) = (A',p,q)$ s.t.

$\forall xy.\ m\leq x<p \wedge q<y\leq n.\ s(A',x) \leq \phi \leq s(A',y) \wedge p > q$

$\Rightarrow \forall xy.\ m\leq x<p \wedge q<y\leq n.\ s(A',x) \leq s(A',f) \leq s(A',y) \wedge p>q$

case 2 : i <= f

This follows a symmetrically similar argument to *case 1*.

case 3 : j < f < i

By lemma 4

$f_2(A,m,n,s(A,f)) = (A',i,j)$ s.t.

$\forall xy.\ m\leq x<i \wedge j<y\leq n \Rightarrow s(A',x) \leq s(A,f) \leq s(A',y)$

Now $j<f<i$ means that $s(A,f)\leq\phi$ *and* $s(A,f)\geq\phi$, which means $s(A,f) = \phi$.

Lemma 4 also means that $s(A',f)=\phi=s(A,f)$ so Lemma 4 means

$f_1(A,m,n) = f_1(A',i,j)$ s.t.

$j<f<i \wedge \forall xy.\ m\leq x<i \wedge j<y\leq n \Rightarrow s(A',x) \leq s(A',f) \leq s(A,y)$

Now $j<f<i \Rightarrow f_1(L,i,j) = (L,i,j)$ so

$f_1(A,m,n) = (A',i,j)$ s.t.

$\forall xy.\ m\leq x<i \wedge j<y\leq n \Rightarrow s(A',x) \leq s(A',f) \leq s(A,y) \wedge i>j$

6.5.2.7 Finally

find(A) =f1(A,1,N)= (A',m,n) s.t.

$\forall xy.\ 1 \le x < m \wedge n < y \le N \Rightarrow s(A',x) \le s(A',f) \le s(A',y) \wedge m > n$

$\Rightarrow \forall xy.\ 1 \le x \le f \le y \le N \Rightarrow s(A',x) \le s(A',f) \le s(A',y)$

6.5.3 Using the Model Function to Alter the Procedural Program

Using Lemma 2 and 3 the "**goto**" statement within the main body of the procedural program can be removed.

This is because the function f1 has been manipulated as a result of lemmas 1 and 2 and when converted back into a "**while**" loop (according to the strategy described in chapter 5) the "**goto**" is no longer required (since the function only has one exit point).

```
f1(A,m,n) = if m <= n
                then
                let (A',i,j) = f2(A,m,n,s(A,f)) in
                if f <= j then f1(A',m,j)
                          else if i <= f then f1(A',i,n)
                                         else f1(A',i,j)
                else A' ;
```

Converting this into a "**while**" loop form gives :

```
while m <= n do
   let (A',i,j) = f2(A,m,n,s(A,f)) ;
   if f <= j then n := j
             else if i <= f then m := i
                            else begin m:=i; n:=j end
```

Where, the "let (A',i,j) = f2(A,m,n,s(A,f))" can be replaced by converting the function f2 back into a while loop. Since f2 has not been manipulated during the proof, it will correspond to the original Algol "**while**" loop from which it was produced.

6.5.4 Comparing the Manipulations and Proofs with the Axiomatic Method

The Axiomatic Method was used by Hoare to construct and prove the program Find in the programming language Algol. The Axiomatic method is the main analysis and proof technique available to procedural programmers.

This section attempts to compare it with the functional modelling strategy.

Manipulation of The Model

The principal difference between the modelling approach an the Axiomatic approach is the difference between a functional and procedural notation.

The functional style is rich in algebraic properties, allowing a programmer to freely manipulate the structure of their program without affecting the values that it computes.

The procedural notation, on the other hand, is far more rigid: algebraic manipulation rules, where they exist, are far more involved due to changes to the "implicit state. If a programmer wants to replace one statement-sequence with an apparently equivalent statement-sequence then a proof may be required, demonstrating that the two statement sequences are, indeed, equivalent.

6.5.4.1 Substitution

The substitution of an expression for a variable with the consequent removal of a let abstraction in the modelling technique is equivalent to the use of the Assignment Axiom of the axiomatic method.

6.5.4.2 Partial Evaluation

Model functions can be partially evaluated, which in the Axiomatic Method corresponds to replacing the procedural code with fresh code. In this situation, using the Axiomatic Method, a proof would be necessary to show equivalence between the replacement code and the code that it replaces.

6.5.4.3 Projective Abstraction

Some values computed by a procedural program fragment may not be returned by the corresponding model function, since they do not contribute towards the evaluation of the tuple onto which the model is projected.

In the Axiomatic Method there is no such thing as Projective Abstraction, all the identifiers assigned values in the program can be referred to in propositions inserted in between statements of the program.

6.5.5 Summary

In general, it appears that the Axiomatic Method is best suited to the construction of correct programs, since, as Hoare shows, the program can be constructed from the assertions in a reasonably intuitive manner.

The Functional Modelling technique seems more appropriate where a program already exists, and requires analysis and proof. The Modelling technique makes manipulation of the program easier. This is an advantage if the programmer is not actually sure what properties of a program are significant.

In such situations the ability to partially evaluate models, to project their results onto sub domains and to use the Fold/Unfold transformation rules and structural induction provides a very flexible and comprehensive analytic tool.

6.6 Correctness Proof for Pascal List Data Type

It is possible to use the modelling technique to demonstrate that the "linked-list" implementation of the list Abstract Data Type in Pascal is correct, in the sense that it obeys the list axioms, and does not produce any extra side-effects. Such a proof allows the programmer to replace the heap function and heap index with the normal "functional" list constructors and selectors. This technique can be used for *any* Abstract Data Type which can be captured by algebraic equalities.

6.6.1 The Pascal Program

Most Pascal textbooks contain a definition of the recursive data type "list". This data type is usually defined in terms of pointers which link together the elements of the list, with the empty list being represented by the "nil" pointer. The definitions may differ in minor detail, but all follow a very similar pattern. A typical implementation of lists is given below:

```
program list ;
const
  Empty = nil ;
type
  list = ^ListRec ;
  listRec = record
              data : integer ;
              next : list
              end ;

function cons(x : integer ; L : list ) : List ;
var result : list ;
begin
  new(result) ;
  result^.data := x ;
  result^.next := L ;
  cons := result
end ;

function head(L : list) : integer ;
begin
  head := L^.data
end ;

function tail(L : list) : list ;
begin
  tail := L^.next
end ;

function IsEmpty(L : list) : boolean ;
begin
  IsEmpty := L = Empty
end ;
```

6.6.2 The Model for the Linked List

In the model a new name must be found for the functions "head", "tail" and "cons", since these names are already used in the model for list constructors. To avoid confusion, the transformed function names corresponding to the procedural program functions and constants shall be prefixed by α.

In this model a list is used to model the record structure of Pascal. u and s are list update and select functions.

```
val αEmpty = -1 ;
fun αcons(x,L,H,hp) =
    let val result = hp ;
        val hp'    = hp + 1 ;
        val H' y   = if y = result then u(H(result),data,x)
                                   else H y
        val H" y   = if y = result then u(H(result),next,L)
                                   else H' y
        in (result,H",hp') ;

fun αhead(L,H) = s(H(L),data) ;
fun αtail(L,H) = s(H(L),next) ;
fun αIsEmpty(L) = L = -1 ;
```

6.6..3 Manipulation of the Model

Manipulation simplifies this model to:

```
val αEmpty = -1 ;
fun αcons(x,L,H,hp) =
  (hp,
   fn y => if y=hp then u(u(H(hp),data,x),next,L) else H(y),
   hp+1) ;

fun αhead(L,H) = s(H(L),data) ;
fun αtail(L,H) = s(H(L),next) ;
fun αIsEmpty(L) = L = -1 ;
```

6.6.4 The List ADT Axioms

The algebraic definition of a list data structure consists of six axioms as follows:

```
i)   IsEmpty(Empty) = true
ii)  ∀x,L.IsEmpty(cons(x,L)) = false
iii) ∀x,L.head(cons(x,L)) = x
iv)  ∀x,L.tail(cons(x,L)) = L
v)   head(empty) = error
vi)  tail(empty) = error
```

6.6.5 Heap and Heap-Pointer Axioms

The following axioms concerning the heap and heap-pointer have been found useful in proving properties of models of pointer-based procedural programs:

A1) $hp \geq 0$
The heap pointer is always non-negative.

A2) $\forall p.pointer(p) \Rightarrow p < hp$
No pointer value is larger than, or equal to, the current value of the heap pointer.

A3) $hp = 0 \Rightarrow \forall x.H(x) = Rubbish$
When the heap pointer has a value of zero, no heap store is allocated.

A4) $p = q \Rightarrow H(p) = H(q)$
This axiom arises directly from the properties of functions.

A5) $s(u(T,f,x),f) = x$

A6) $\forall x.x < 0 \Rightarrow H(x) = Rubbish$

6.6.6 The Proofs

The proof that the procedural functions implement the list Abstract Data Type relies upon showing that the model functions obey the list axioms.

The function αcons returns a 3-tuple, the first element of which is the list pointer and the other two are the heap and heap pointer. It is thus the first value returned by αcons which is the subject of the proof that the list axioms are respected by the model functions.

In the proofs that follow, αcons is thus projected onto the pointer-valued result of the procedural function "cons" which it models.

There is, however, the possibility that the model function αcons respects the list axioms but also produces some other effect upon the heap, so a further proof is required demonstrating that the *only* effect of αcons is to produce the pointer value for the constructed list.

The whole proof consists of two parts: a demonstration that the model functions respect the list axioms and a demonstration that this is all that they do.

```
i)    IsEmpty(Empty) = true
is αAIsEmpty(αEmpty) = αEmpty = -1
                     = -1     = -1
                     = true
```

Proof of List Axiom i)

```
ii)  ∀L,x.αIsEmpty(αcons(x,L)) = false
     this becomes
     let val (v,H',hp') = αcons(x,L,H,hp) in αIsEmpty(v)
     effectively projecting αcons onto v


     let val (v,H',hp') = αcons(x,L,H,hp) in αIsEmpty(v)
   = let val (v,H',hp') =
               (hp,
                fn y => if y = hp
                        then u(u(H(hp),data,x),next,L)
                        else H(y),
               hp+1)
       in αIsEmpty(v)
   = αIsEmpty(hp)
   = false  (by axiom A1)
```

Proof of List Axiom ii)

```
iii) ∀L,x.αhead(αcons(x,L)) = x
     this becomes
     let val (v,H',hp') = αcons(x,L,H,hp) in αhead(v,H')
     (projection of αcons)


   let val (v,H',hp') = αcons(x,L,H,hp) in αhead(v,H')
 = let val (v,H',hp') =
           (hp,
            fn y => if y = hp
                    then u(u(H(hp),data,x),next,L)
                    else H(y),
            hp+1)
    in αhead(v,H')

 = αhead(hp,fn y => if y = hp
                    then u(u(H(hp),data,x),next,L)
                    else H(y))

 = s((fn y => if y = hp
              then u(u(H(hp),data,x),next,L)
              else H(y)) hp,
     data)
 = s(u(u(H(hp),data,x),next,L),data)
 = x.
```

Proof of List Axiom iii)

```
iv) ∀L,x.αtail(αcons(x,L)) = L
     this becomes
     let val (v,H',hp') = αcons(x,L,H,hp) in αtail(v,H')
     (projection of αcons)


    let val (v,H',hp') = αcons(x,L,H,hp) in αtail(v,H')
  = let val (v,H',hp') =
              (hp,
               fn y => if y = hp
                       then u(u(H(hp),data,x),next,L)
                       else H(y),
               hp+1)
     in αtail(v,H')

= αtail(hp,fn y => if y = hp
                   then u(u(H(hp),data,x),next,L)
                   else H(y))

= s((fn y => if y = hp
                   then u(u(H(hp),data,x),next,L)
                   else H(y)) hp,
          next)
= s(u(u(H(hp),data,x),next,L),next)
= L.
```

Proof of List Axiom iv)

Axioms v) and vi) are direct consequences of the pointer Axiom A6) above (§6.6.5).

6.6.7 Proof of Freedom From Other Effects

To be satisfied with an implementation of an A.D.T, it is not enough simply to prove that the functions defined *satisfy* the algebraic description of the A.D.T, it is also crucial to demonstrate that no *other* effects are produced apart from those required to satisfy the axioms.

For example, a proof that a data structure and associated functions implements a "binary tree", that consists of a demonstration that some part of the code satisfies the binary-tree axioms, does not preclude the possibility that one of the functions *also* writes 20 asterisks to the printer.

6.6.8 A Proof of Freedom from Side-Effects for the Linked-List

In these model functions, it is only αcons which returns values other than those demanded by the list axioms. It returns a new heap and heap pointer. It shall thus be necessary to demonstrate that only *one* new pointer is allocated (the one for the list constructed by cons) and that *no other* pointer values are affected when cons is executed.

The new heap pointer returned by αcons is hp+1, so only one *new* pointer can have been created.

To show that no other pointers have *changed* their value after execution of "cons", it is necessary to show that for all integers strictly less than the value of the heap pointer passed to "αcons", the heap function returned contains identical mappings to that passed.

formally:

$$\alpha cons(x,L,H,hp) = (hp,H',hp+1) \quad \text{such that} \quad \forall i < hp.\ H'(i) = H(i)$$

The proof turns out to be trivial:

by definition of "αcons":

```
    H' = fn x => if x = hp
                 then u(u(H(result),data,x),next,L)
                 else H(x)
so
    ∀i < hp.H'(i) = H(i).
```

6.6.9 Summary

The modelling strategy can be used to prove that an Abstract Data Type is correctly implemented in a procedural program.

Such a proof arms the procedural programmer with axioms taken from the A.D.T which may be asserted at the relevant points in their program.

6.7 The "C" Programming Language

In the "C" programming language, side-effects within expressions are commonplace.

For example consider the "C" expression:

```
(x = 4) + (y = x)
```

This expression is "φ1 + φ2", where "φ1 = 4" and "φ2 = x", but, (assuming left-right expression evaluation) the scope of the assignment "x = 4" *includes* the right-hand-side of the overall addition, and so includes the occurrence of "x" in "φ2 = x".

The expression "(x=4) + (y=z)" is thus modelled by the let abstraction list:

```
let val x = 4 ;
    val φ1 = 4 ;
    val y = x ;
    val φ2 = x in φ1 + φ2
```

The resulting (assignment-free) expression can be manipulated by substitution to an expression in terms of the original identifiers, thus removing the extra variables φi, and making the expression and the let abstractions *independent*. For the expression above this would give:

```
let val x = 4 ;
    val y = x in
      8
```

6.7.1 φi Are "Temporary Variables"

This process is similar to compilation of expressions, where the machine code produced evaluates sub-expressions, and stores their value in a temporary location. In this case the "temporary location" is the new identifier introduced to denote the value of a sub-expression which contains a side-effect. Indeed this identifier is *truly* temporary since if can subsequently be "manipulated-out".

6.7.2 A Typical Loop in "C"

It is very common in the "C" programming language to find examples of side-effects in the bodies of loops, a typical example is given below:

```
while ( (c=getchar())!=EOF) if (c=='.') printf("OUCH!");
```

This loop is modelled by the function f below:

```
fun f(Input) = w(hd(Input),Input,[]) ;
fun w(c,Input,Output) =
  let val c = hd(Input) in
    if not(c=EOF)
       then if  c = '.' then w(c,tl(Input),append(Output,"OUCH!"))
                        else w(c,tl(Input),Output)
       else (c,tl(Input),Output) ;
```

This model could now be converted in Pascal (according to the strategy set out in chapter five, for example). The Pascal version would not make use of the side-affecting features offered to the "C" programmer.

Alternatively, the program could also be converted back into a "C" program, written in a style which avoids the use of the language's more "subtle" features.

6.7.3 Another Typical Example

This next example is taken from page 48 of the book "The C Programming Language" by Kernighan and Ritchie [63].

The modelling strategy is good for such programs since all the subtleties and implicit effects of a program are "brought out into the open".

The lines of the program are numbered in order that the program can be discussed. The numbers are not part of the program itself.

```
/* strcat: concatenates t to the end of s; s must be big enough */
/* 1 */    void strcat(char s[], char t[])
/* 2 */    {
/* 3 */        int i,j ;
/* 4 */        i = j = 0;
/* 5 */        while (s[i] != '\0')          /* find end of s */
/* 6 */            i++;
/* 7 */        while ((s[i++] = t[j++]) != '\0')   /* copy t */
/* 8 */            ;
/* 9 */    }
```

6.7.4 Explanation of the Program

Line number 4 contains a simple case of an assignment within an expression. In general the expression could be far more involved. In this case the line has the effect of assigning zero to both the variables "i" and "j".

lines 5 and 6, simply advance the index "i" to the end of the string "s". In line 6, the increment of "i" acts as a statement, although the value "i" is evaluated by the expression "i++" the effect of terminating the expression with a semicolon is to turn the expression into a statement.

On line 7, there is a while loop with no body. This may appear perverse to a programmer unfamiliar with the "C" programming language: One might expect the loop to have *no* observable effect if it has *no* body. However, in "C", it is common to find loops that perform side-effects as a result of evaluating their controlling expressions.

In the case of line 7, the loop's expression evaluates to the test "is t[j] equal to the character with code zero". In addition to evaluating this boolean expression, the side effect is to assign the jth element of the array "t" to the ith element of the array "s" *and* to increase the values of "i" and "j" by one.

Note that although "i" and "j" are incremented, because the side-affecting expression is written postfix, the evaluated result is "i" and "j" and *not* "i+1" and "j+1". Also, the side-effects happen *even* when the loop body is *not* executed (due to the predicate evaluating to false). This means that the zero character that acts as a sentinel for the arrays will also be copied.

As this simple example demonstrates, "a little bit of 'C' goes an awfully long way" !

6.7.5 The Model

All of these subtleties are reflected in the model for "strcat" which is given below:

```
fun strcat(s,t) =
    let val i = w1(s,0) in
      let val s = w2(s,i,0,t) in s
where
    fun w1(s,i) = if select(s,i) <> chr(0)
                  then w1(s,i+1)
                  else i
and
    fun w2(s,i,j,t) =
        if select(t,j) <> chr(0)
           then w2(update(s,i,select(t,j)),i+1,j+1,t)
           else update(s,i,select(t,j)) ;
```

It is interesting, in passing, to note that, in their definition of the "C" language [63], Kernighan and Ritchie have this to say about the order of evaluation of sub-expressions:

> "The order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which side effects take place is undefined. Expressions involving a commutative and associative operator (*, +, &, ¦, ^) may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation a temporary variable must be used." (page 185 of [63]).

This appears to make the facility of including assignments in expressions rather a matter of syntactic brevity as opposed to a way of changing the meaning of an expression dynamically.

The point is, that in the "C" programming language, there *are no* commutative or associative operators (*because* of the presence of side-effects).

If the evaluation strategy of the code produced by a particular compiler is known, however, then a modelling strategy can yield a program for which these commutativity and associativity properties do hold, in *all* cases.

6.8 Parallel Execution Paths

Functions in a pure-functional notation are free from any effects other than the evaluation of their result in terms of their arguments, and therefore are considered ideally suited to parallel evaluation [1].

Since a model is a pure-functional version of a procedural program it is highly likely that the modelling approach will expose paths in the program which will facilitate parallel execution of the program.

Consider the program below:

```
program P1 ;

type T  = array [1..1000] of integer ;

var A : T ; i : integer ;

procedure process ;
var j : ineteger ;
    total : integer;
begin
  total := 0 ;
  for j := 1 to A[i] do total := total + A[i] ;
  A[i] := total
end ;


begin
  for i := 1 to 1000 do process
end.
```

The array, "A", is modelled using a list of 1000 elements. The model for this program, projected onto the array "A", is given below:

```
fun P1(A) = fori(A,1) ;
fun fori(A,i) = if i <= 1000 then fori(process(A,i))
                             else A ;
fun process(A,i) = forj(A,i,0,1) ;
fun forj(A,i,total,j) = if j <= A(i)
                        then forj(A,i,update(A,i,total+A(i)),j+1)
                        else A ;
fun update(A,i,v) = if i = 1 then v::tail(A)
                             else head(A)::update(tail(A),i-1,v) ;
```

Next some simple manipulation is performed upon this model:

The function "forj" can be simplified, since:

```
forj(A,i,total,j) = forj'(A(i),total,j)
where fun forj'(n,total,j) = if j<=n then forj'(n,total+n,j+1)
                                     else update(A,i,total) ;
```

The application of update can be distributed through the call to "forj'" in the model, giving:

```
fun P1(A) = fori(A,1) ;
fun fori(A,i) = if i <= 1000 then fori(update(A,i,process(A(i)))
                             else A ;
fun process(n) = forj'(n,0,1) ;
fun forj'(n,total,j) = if j <= n
                       then forj'(n,total+n,j+1)
                       else n ;
```

The function "fori" can now be rewritten as a call to the function map ("map" is an example of a function that is directly amenable to parallel evaluation).

```
fun fori(A) = map(process,A)
where fun map(f,A) = if null(A) then []
                                else f(head(A)) map(f,tail(A)) ;
```

6.8.1 Another Efficiency Improvement

Of course, the call: process(n,0,1) evaluates to n^2, as could be proved by simple structural induction, so a further efficiency improvement could be achieved by putting:

```
fun fori(A) = map(fn z => z*z, A) ;
```

Converting this manipulated model back into the procedural notation, leads to a more efficient procedural program:

```
program P1 ;
type t  = array [1..1000] of integer ;
var A :  T ;
    i :  integer ;
begin
  for i  := 1 to 1000 do A[i] := A[i] * A[i]
end.
```

CHAPTER SEVEN

SEMANTIC FOUNDATIONS

7.1 Introduction

This chapter investigates the semantic foundations of this thesis.

§7.2 defines what it means for a syntactic substitution rule to be valid, and goes on to define what it means for a programming language to be referentially transparent.

§7.3 looks at a few simple example languages and describes their substitution rules with respect to referential transparency.

§7.4 uses a non-referentially transparent language as an example of how a modelling strategy can be constructed and proved-correct in terms of the semantic description of the language.

In §7.5, an important property called "Algebraic Closure" is introduced.

This property demands that a language allow syntactic substitution of all abstract syntactic constructs which have equivalent meaning. The absence of this property is characterised by irritating special cases in the description of a programming language. These special cases may be generalised "out of existence" by application of the definition of "Algebraic Closure" in a generative mode.

In §7.6, an attempt is made to define the semantic domains and mappings that make a language "functional".

In §7.7, an attempt is made to define the semantic domains and mappings that make a language "procedural".

§7.8 considers the interleaving of input and output events in procedural programs and functional models and shows that the Henderson Lazy Stream approach [30] is a form of *projected* model which *does not* represent interleaving of input and output events.

7.2 Valid Substitution Rules and Referential Transparency

7.2.1 The Literature on Referential Transparency of Programming Languages

There are many (and differing) descriptions of referential transparency, each of which usually gives a counter example, or mentions a feature of a programming language which will lead to the language being considered non-referentially transparent.

This section summarises the views put forward on the subject in various books concerning functional programming [16,66,18,65,21,78,72,19,20].

These accounts fall into categories, according to which of the four statements below is mentioned in the account.

i) Languages with assignment statements are non-referentially transparent (sometimes "non-referentially transparent" is also termed "referentially opaque").

ii) Languages which allow global variables to be used by more than one procedure/function are non-referentially transparent.

iii) Languages which do not respect algebraic laws, such as commutativity of addition, are non-referentially transparent.

iv) A Language is referentially transparent if the meaning of whole expressions can be inferred from the meanings of its component expressions alone.

Of these four accounts, only the fourth *defines* referential transparency, the others give properties which must necessarily be absent for a language to be referentially transparent.

account i) is mentioned in [78] (page 12), [72] (page 3), [16] (page 6) and [78] (page 72).

account ii) is mentioned in [21] (page 11) and [66] (page 10).

account iii) is mentioned in [78] (page 71), [21] (page 11), [65] (page 7) and [18] (page 269).

account iv) is mentioned in [20] (page 3), [78] (page 72), [65] (page 7) and [18] (page 10).

The fact that the meaning of an expression must depend solely on the meaning of its sub expressions can be described by delimiting the allowable substitutions that may be performed in the language.

Demanding that algebraic laws must be preserved depends upon the definition of the algebraic laws to be preserved. An algebraic law can also be described by a substitution rule, so the specification of which algebraic laws must be preserved will be treated hereinafter as a choice of valid substitution rules.

On its own, the assignment statement does *not* alter the substitution rules of a language any more than the let-abstraction of functional languages does (this is demonstrated in chapter 3, §?), so account i) on its own must be discarded as unreasonable.

It appears that when a programmer claims that a language is "referentially transparent" what they are asserting is that a particular set of substitution rules are applicable to programs in the language.

Given the semantics of a language, one can thus investigate the truth of such a claim.

If this "coalesced" version of what constitutes the property of referential transparency is accurate, it would certainly account for the differing perspectives placed upon "referential transparency" in the literature; each author may have a slightly different set of substitution rules in mind.

7.2.2 Valid Substitution Rules

The validity of a substitution rule depends upon the semantics of the language. The definition of validity presented here is sufficiently general to apply, not just to programming languages, but to languages in general.

Before the definition can be presented, it is necessary to briefly clarify the terms of reference: the semantics of a language and the substitution rules that may be applied to programs in the language.

Having done this the definition is obvious and trivial.

7.2.3 The Semantics Of A Programming Language

Using the Denotational Description technique [16], the meaning of a program is defined by a function which maps each program of the language to some "mathematical object". The phrase "mathematical object" is intentionally vague. It is up to the language designer to *model* the effect of program-execution by choosing a suitable semantic domain.

The choice of a semantic domain *prescribes* what type of questions a programmer will be *able* to ask the language designer about the execution of programs.

For example, if the meaning of a program is described solely in terms of how many beeps a computer executing the program will make, then questions about what values appear on the screen connected to the computer will clearly be unanswerable.

If the semantics is described in terms only of the values stored in program variables, i.e. an environment mapping, then it will not be possible to ask how long a program takes to execute, or in what order those values are stored.

The choice of a suitable semantic domain in which to describe the meaning of a program is thus a *choice* as to what attributes of a program's execution are to be modelled. This choice represents an important act of abstraction on the part of the language designer.

Having chosen a domain for the meaning of a program, the designer defines the meaning of the language as a mapping from the sentences of the language into this semantic domain.

The domains chosen for the semantic description thus prescribe the kinds of questions that can be answered by the description.

The particular mappings defined, dictate the answers to these questions.

7.2.4 Semantic Descriptions And Syntactic Equivalence

A semantic description of a language is a mapping from sentences of the language into some semantic domain.

A semantic description of a language thus partitions the sentences of the language into a set of *syntactic* equivalence classes. That is, those sentences of a language that are mapped to the same semantic object will be in the same equivalence class.

Given a sentence, S1, and a sentence, S2, if S2 occurs in the same equivalence class as S1, then the sentence S2 my be used in place of the sentence S1 *without a change of meaning*. That is, S2 may be *substituted* for S1.

Clearly, in performing substitutions it would be desirable to find a *sub string* within a sentence that may be substituted for, rather than replacing an *entire* sentence with a different one.

A substitution rule is *valid* if it is meaning preserving. That is, if one sentence can be transformed into another, by application of a substitution rule, then both sentences must be assigned *identical* values by the semantic mapping.

7.2.5 An Example: Strings of Digits

Consider a language, D, which consists of all non-empty, finite strings of digits.

In order to give a meaning to D it is first necessary to decide upon a semantic domain.

Let the semantic domain be the integers, $\mathbb{N}$.

In semantic descriptions, members of $\mathbb{N}$ will be written in bold typeface to avoid confusion with members of D.

Now, in order to define the meaning of the language, it is necessary to define a mapping, $\mathcal{F}$, which assigns a member of $\mathbb{N}$ to each string in D.

7.2.5.1 Notation For describing $\mathcal{F}$

Of course, in order to describe the mapping, $\mathcal{F}$, one has to use some form of notation. Clearly the *meaning* of *this* notation is assumed to be *understood*, otherwise no progress will be made towards the description of a semantics for the language, D.

This is an important issue, but it is not of concern here, since the semantic function, $\mathcal{F}$, is of only of interest inasmuch as it defines equivalence classes in the syntactic domain.

For the language, D, two different meaning functions will be defined: $\mathcal{F}_1$ and $\mathcal{F}_2$.

The members of D will be written using list notation, where "*cons*" and "*append*" have their usual meaning, and where "Λ" denotes the empty list. The members of $\mathbb{N}$ will be constructed from the addition function, which has its usual mathematical meaning and is written with the infix symbol "+". "L" will be used to denote arbitrary members of D and "x" will be used to denote arbitrary digits.

7.2.5.2 A Meaning, $\mathcal{F}_1$, For D

The meaning is a function $\mathcal{F}_1$: D → $\mathbb{N}$

Let the mapping be that a member of D is mapped to its length.

Formally, the meaning of a string,d is the (unique) solution to the following recursion equation, defined by cases below:

$$\mathcal{F}_1\ulcorner\Lambda\urcorner = 0$$
$$\mathcal{F}_1\ulcorner cons(x,L)\urcorner = 1 + \mathcal{F}_1\ulcorner L\urcorner$$

According to $\mathcal{F}_1$ the strings ⌜1234⌝, ⌜2134⌝, ⌜9999⌝ and ⌜0000⌝ are all in the same syntactic equivalence class (because they all *mean* 4, that is, are mapped to 4 by $\mathcal{F}_1$). In this particular equivalence class there are 10^4 different members.

In general, $\mathcal{F}_1$, defines an equivalence class for each N in $\mathbb{N}$, containing 10^N members of D.

It is possible, given $\mathcal{F}_1$, to describe transformation rules which will allow "substitution of equivalent sub stings". All that is required is to verify that any substitution performed on a sentence is valid (with respect to $\mathcal{F}_1$).

7.2.5.3 A Valid Substitution Rule, Sub_1, For D w.r.t. $\mathcal{F}_1$

An example of a valid substitution rule is that any digit may be replaced by any other digit in any string.

7.2.5.4 An Invalid Substitution Rule, Sub_2, For D w.r.t. $\mathcal{F}_1$

An example of an invalid substitution rule is that leading zeros in a sentence may be removed.

7.2.6 A Different meaning function $\mathcal{F}_2$

Now, a different, perhaps more intuitively agreeable, meaning is ascribed to D. This meaning treats each member of D as a *numeral* which is mapped to the corresponding member of $\mathbb{N}$.

$$\mathcal{F}_2\ulcorner cons(0,\Lambda)\urcorner = 0$$
$$\mathcal{F}_2\ulcorner cons(1,\Lambda)\urcorner = 1$$
$$\mathcal{F}_2\ulcorner cons(2,\Lambda)\urcorner = 2$$
$$\mathcal{F}_2\ulcorner cons(3,\Lambda)\urcorner = 3$$
$$\mathcal{F}_2\ulcorner cons(4,\Lambda)\urcorner = 4$$
$$\mathcal{F}_2\ulcorner cons(5,\Lambda)\urcorner = 5$$
$$\mathcal{F}_2\ulcorner cons(6,\Lambda)\urcorner = 6$$
$$\mathcal{F}_2\ulcorner cons(7,\Lambda)\urcorner = 7$$
$$\mathcal{F}_2\ulcorner cons(8,\Lambda)\urcorner = 8$$
$$\mathcal{F}_2\ulcorner cons(9,\Lambda)\urcorner = 9$$
$$\mathcal{F}_2\ulcorner append(L,cons(x,\Lambda))\urcorner = 10 * \mathcal{F}_2\ulcorner L\urcorner + \mathcal{F}_2\ulcorner cons(x,\Lambda)\urcorner$$

7.2.6.1 An Invalid Substitution Rule, Sub_1, For D w.r.t. $\mathcal{F}_2$

Any digit may be replaced by any other digit in a member of D.

7.2.6.2 A Valid Substitution Rule, Sub_2, For D w.r.t. $\mathcal{F}_2$

Leading "0" characters may be deleted.

7.2.7 Referential Transparency

In order to decide whether or not a set of substitution rules are valid one has to consult the semantic mapping.

Referential transparency appears to be the property that all "normal algebraic" substitution rules are valid.

Whether or not a language exhibits the property of referential transparency will thus be expressed "with respect to the semantics of the language and a set of substitution rules".

7.2.8 Another Example

Clearly, one could define the semantics in such a way as to allow any set of substitution rules to be valid. Consider a language made up of a sequence of assignments of the form "light := on" or "light := off". In B.N.F, this is :

E ::= on ¦ off

S ::= Light := E; S ¦ Λ

7.2.8.1 A Semantic Description of the Language

Any sequence ending in "Light := on" shall be mapped to 1, and any other sequence shall be mapped to 0.

7.2.8.2 Substitution Rules

Now, given *this* meaning, the substitution rule "all but the last assignment to the variable 'Light' may be deleted" is valid, however, the substitution rule "the order of assignments may be swapped around" is invalid.

7.2.8.3 Referential Transparency Again

So if a programmer *expects* to be able to delete all but the last assignment, then the programmer will conclude that the language is referentially transparent. However, a programmer who expects to be able to rearrange the order of the assignments will describe the language as non-referentially transparent.

Clearly the decision as to whether a language is referentially transparent or not depends upon what substitution rules the programmer *expects* the language to permit.

7.2.9 Altering the Semantics to Fit the Substitution Rules

It is clearly possible to make *any* substitution rule valid by an appropriate choice of semantics. For example, the number of occurrences of the sub string "Light := on" could be defined to be the *meaning* of a sentence. Using *this* definition of meaning, the rule "the order of assignments may be swapped around" is valid, but the rule "all but the last assignment to the variable 'Light' may be deleted" is invalid.

In this example there is no problem with a language designer deciding upon the semantics of the language, and so the issue as to whether or not the language is referentially transparent becomes a matter of choice.

For *programming* notations however, it is not possible to simply *decide* what the semantics of a language should be; the semantic mapping is restricted to those mappings which correspond to realisable and desirable implementations. Such is the case with functional languages.

7.3 Functional Languages and Referential Transparency

In this section various functional-style languages are described.

Each is given a syntactic definition using B.N.F and a semantic definition using the Denotational Method.

Starting with an incredibly simple language, the discussion proceeds by the addition of various features to the language.

After each addition the discussion focuses on the substitution rules which are enjoyed by the language. The substitutions sought are those which render the language referentially transparent in respect of the normal algebraic laws of substitution (e.g. commutativity, associativity and the substitution of values for identifiers).

7.3.1 An Extremely Simple Functional Language

The functional language below contains only one syntactic formation rule to form an expression from an addition symbol and two numerals.

7.3.1.1 Syntax

$$E ::= D_1 + D_2$$

7.3.1.2 Semantics

The meaning of a program written in this language is given by the semantic function $\mathcal{E}$.

$$\mathcal{E} : E \to \mathbb{N}$$

$$\mathcal{E}\ulcorner D_1 + D_2 \urcorner = \mathcal{F}_2\ulcorner D_1 \urcorner + \mathcal{F}_2\ulcorner D_2 \urcorner$$

$\mathcal{F}_2$ is the semantic function described in §7.2.6, D is the syntactic domain of non-empty, finite strings of digits (defined in §7.2.5) and "+" is the normal mathematical addition function over natural numbers $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Although exceedingly simple, the semantic function $\mathcal{E}$ does split up the language into an infinite number of sets of equivalent sentences, each of which contains an infinite number of sentences of the language.

7.3.1.3 Substitution Rules

Clearly, Sub_2, the substitution rule defined in §7.2.6.2, which allows leading zeros to be deleted from numerals, is valid (due to the use of $\mathcal{F}_2$).

Also, the commutative property of addition is enjoyed by programs in E, since $\forall D_1, D_2 . \mathcal{E}\ulcorner D_1 + D_2 \urcorner = \mathcal{E}\ulcorner D_2 + D_1 \urcorner$.

Thus, it can be said that "E is referentially transparent with respect to $\mathcal{E}$ and with respect to all the normal algebraic equalities that one would expect to find within the syntax of E".

7.3.2 Non Termination

The issue of non-termination of programs and the meaning of "undefined" which is given to such programs, severely complicates semantic discussions.

In order to avoid complexity, the issue of termination is addressed now, and then subsequently ignored.

A syntactic construct, *nt*, is introduced which, when executed, causes non-termination to occur. After the issue has been briefly investigated, the construct which causes non-termination is removed.

Such an explicit means of creating non-termination is obviously unrealistic, but it is perfectly adequate for illustrative purposes.

7.3.2.1 Syntax

V ::= D ¦ *nt*

E ::= V_1 + V_2

7.3.2.2 Semantics

The semantic domain of the language must now include the undefined element $\perp$ (in [57] this is called ω). This domain is called $\mathbb{N}_\perp$ to distinguish it from the set of natural numbers $\mathbb{N}$.

Since $\perp$ is included as an argument to the addition function used in the semantic description, it now becomes necessary to describe the result of "$\perp$+n", "n+$\perp$" and "$\perp$+$\perp$" for natural numbers, n.

To begin with "+" is chosen to be strict in both arguments, that is, "$\perp$+n" = "n+$\perp$" = "$\perp$+$\perp$" = $\perp$.

The semantic consequence of this choice for the meaning of E, is as follows:

$$\mathcal{E} : E \rightarrow \mathbb{N}_{\perp}$$
$$\mathcal{E}\ulcorner D \urcorner = \mathcal{F}_2\ulcorner D \urcorner$$
$$\mathcal{E}\ulcorner nt \urcorner = \perp$$
$$\mathcal{E}\ulcorner V_1 + V_2 \urcorner = \mathcal{E}\ulcorner V_1 \urcorner + \mathcal{E}\ulcorner V_2 \urcorner$$

As can readily be verified, the language still respects commutativity of addition, in particular : $\mathcal{E}\ulcorner nt + D \urcorner \equiv \mathcal{E}\ulcorner D + nt \urcorner \equiv \mathcal{E}\ulcorner nt + nt \urcorner \equiv \mathcal{E}\ulcorner nt \urcorner$.

However, if *all* binary functions are defined to be strict in both their arguments (or in either of their arguments for that matter), then some algebraic identities will *not* be valid.

Consider, for example, the introduction of a multiplication operator into the language E :

7.3.2.3 Syntax

$$V ::= D \mid nt$$
$$E ::= V_1 + V_2 \mid V_1 * V_2$$

7.3.2.4 Semantics

$$\mathcal{E} : \mathbb{N}_{\perp}$$
$$\mathcal{E}\ulcorner D \urcorner = \mathcal{F}_2\ulcorner D \urcorner$$
$$\mathcal{E}\ulcorner nt \urcorner = \perp$$
$$\mathcal{E}\ulcorner V_1 + V_2 \urcorner = \mathcal{E}\ulcorner V_1 \urcorner + \mathcal{E}\ulcorner V_2 \urcorner$$
$$\mathcal{E}\ulcorner V_1 * V_2 \urcorner = \mathcal{E}\ulcorner V_1 \urcorner * \mathcal{E}\ulcorner V_2 \urcorner$$

7.3.2.5 The Semantics of *

Now the issue of the strictness (or otherwise) of the multiplication function becomes paramount if the language is to be referentially transparent with respect to algebraic properties of multiplication.

If "*" is defined, like "+", to be strict, then $\mathcal{E}\ulcorner 0*nt\urcorner \equiv \mathcal{E}\ulcorner nt*0\urcorner \equiv \mathcal{E}\ulcorner nt*nt\urcorner \equiv \mathcal{E}\ulcorner nt\urcorner$. But, if one argument to multiplication is zero and the other is some *unknown* natural number, it is *still* possible to decide that the result is zero.

A strict multiplication function is therefore not suitable.

Instead a semantics is required where the multiplication of two arguments is defined to be zero if *either* argument is zero.

This is a case of defining the desired substitution rules and then attempting to construct an implementation whose semantics respects these substitution rules (c.f. §7.2.9). As stated earlier (in §7.2.9), this is *not* always possible since some desirable semantics are unimplementable (or so inefficient as to be impractical).

7.3.2.6 Evaluation Strategies

The issue is thus: "is it possible to provide an *implementation* of the language which allows the correct (i.e. algebraic) interpretation to be placed upon the multiplication operator?"

Fortunately the answer is "yes":

The processor, when evaluating the multiplication operator, alternately performs one "evaluation step" of one argument to the operator and then the other, and halts with result zero if *either* argument evaluates to zero after a particular evaluation step.

If either argument is $\ulcorner nt\urcorner$ then the evaluation step has no effect and the processor performs an evaluation step for the other argument.

Clearly then, if both arguments are $\ulcorner nt\urcorner$, then the computation of the expression fails to terminate. If, however, *either* argument evaluates to zero then computation immediately halts.

For such an implementation the semantics of "*" will be as follows :

$\forall x \in \mathbb{N}_{\perp}.\ 0*x = x*0 = 0$ *and*

$\perp*\perp = \perp$ *and*

$\forall n,m \in \mathbb{N},\ n,m \neq 0.$ n*m is normal multiplication of natural numbers.

The languages Hope[51], ML[41], Miranda[52] and many others define "*" to be strict, so these languages cannot be said to be completely referentially transparent with respect to normal algebraic properties. This can cause problems when using the structural induction proof technique since it will be possible to show f(x)*0 = 0 for all recursion equations, f. This is not true, since the recursion equation, f, may not terminate.

7.3.2.7 Impossible Substitutions

The theory of computability dictates that certain substitution rules will be unrealisable (for example, being able to substitute "true" or "false" for expressions involving equality of functions will not be possible in all cases).

7.3.3 A Functional Language With Identifiers

The ability to write non-terminating programs is now abandoned, so that attention can be focused upon other issues.

The language E, described earlier in §7.3.1, is extended by allowing a program to refer to identifiers.

Although identifiers are allowed in the language, at this stage no mechanism has been included to *bind* values to identifiers; the identifiers effectively stand for constants, the particular value of which is defined by the state in which the program is evaluated.

7.3.3.1 Syntax

$$V ::= I \mid D$$

$$E ::= V_1 + V_2$$

Programs are expressions which are evaluated in a state, σ, which maps identifiers in I, to expressible values in $\mathbb{N}$.

7.3.3.2 Semantics

$$\mathcal{E} : E \rightarrow (I \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$$

$$\mathcal{E}\lceil V_1 + V_2 \rceil\sigma = \mathcal{E}\lceil V_1 \rceil\sigma + \mathcal{E}\lceil V_2 \rceil\sigma$$

$$\mathcal{E}\lceil I \rceil\sigma = \sigma\lceil I \rceil$$

$$\mathcal{E}\lceil D \rceil\sigma = \mathcal{F}_2\lceil D \rceil$$

7.3.3.3 Substitution Rules

Clearly the language described in §7.3.1 is a subset of E both syntactically and semantically. All the substitution rules described in §7.3.1.3 apply to E. Also the commutative property of addition extends to arguments which are identifiers.

Once again, it is possible to say that E is referentially transparent with respect to $\mathcal{E}$ and with respect to all the normal algebraic properties.

7.3.4 A Functional Language with Inadequate Syntax

The next example language shows how consideration of substitution rules of a language can reveal shortcomings in the language definition.

The language described above in §7.3.3 is extended further, to include syntax for *binding* values to identifiers, thus changing the environment in which a program is evaluated *as the evaluation proceeds.*

However, as will be seen, some of the algebraic substitutions that one would expect to find are not permitted (simply because the syntax is not rich enough). This issue is discussed further in §7.5.

7.3.4.1 Syntax

V ::= I ¦ D

E ::= V_1 + V_2

F ::= let val I = E in F ¦ E

7.3.4.2 Semantics

The meaning of a program F, is described by the semantic function $\mathcal{F}$.

$$\mathcal{E} : E \rightarrow (I \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$$

$$\mathcal{F} : F \rightarrow (I \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$$

$$\mathcal{E}\ulcorner V_1 + V_2 \urcorner\sigma = \mathcal{E}\ulcorner V_1 \urcorner\sigma + \mathcal{E}\ulcorner V_2 \urcorner\sigma$$

$$\mathcal{E}\ulcorner I \urcorner\sigma = \sigma\ulcorner I \urcorner$$

$$\mathcal{E}\ulcorner D \urcorner\sigma = \mathcal{F}_2\ulcorner D \urcorner$$

$$\mathcal{F}\ulcorner \text{let val } I = E \text{ in } F \urcorner\sigma = \mathcal{F}\ulcorner F \urcorner(\sigma[\mathcal{E}\ulcorner E \urcorner\sigma \ / \ \ulcorner I \urcorner])$$

7.3.4.3 Some Example Programs

$$P_1 \equiv \ulcorner \text{let val } I_1 = 5 \text{ in let val } I_2 = 6 \text{ in } I_1 + I_2 \urcorner$$

$$P_2 \equiv \ulcorner \text{let val } I_1 = 5 + 6 \text{ in let val } I_2 = 7 \text{ in } I_1 + I_2 \urcorner$$

7.3.4.4 Substitution Rules

Expressions bound to identifiers by let abstractions may often be substituted into the expressions in which they are used.

For example :

$\mathcal{E}\ P_1 = \mathcal{E}\ulcorner 5 + 6 \urcorner = \mathcal{E}\ulcorner 11 \urcorner = 11$

Thus P_1 can be replaced by, amongst others, $\ulcorner 5 + 6 \urcorner$ or $\ulcorner 11 \urcorner$.

$\mathcal{E}\ P_2 = \mathcal{E}\ulcorner 11 + 7 \urcorner = \mathcal{E}\ulcorner 18 \urcorner$.

Thus P_2 can be replaced by, amongst others, $\ulcorner 11 + 7 \urcorner$ or $\ulcorner 18 \urcorner$.

However, although one would expect to be able to substitute $\ulcorner 5 + 6 \urcorner$ for $\ulcorner I_1 \urcorner$ in P_2, one cannot since $\ulcorner 5 + 6 + I_2 \urcorner$ is *not a member* of the syntactic class F. Thus the language is *not* referentially transparent with respect to normal algebraic substitution rules because the syntax does not allow such substitutions.

7.3.4.5 The Beta Reduction Rule

The language F, is sufficiently rich to begin a discussion of evaluation of expressions in F into a "normal form". That is, F, can be regarded as a term in a term-rewriting system. The normal forms of the system are those members of D with no leading zeros.

As the examples above show, rewriting an expression in F until a normal form is reached is analogous to the application of the β-reduction rule of the Lambda Calculus [12].

7.3.4.6 Inverse of Semantic Functions

In order to discuss the term-rewriting properties of the language F, it will be necessary to use the inverse, $\mathcal{M}^{-1}$, of a semantic function, $\mathcal{M}$.

Of course the inverse of a semantic function, in general, produces a *set* of sentences drawn from a syntactic domain. When the notation $\mathcal{M}^{-1}(x)$ is used in the description of substitution rules, it stands for an arbitrary member of this set.

For example $\mathcal{F}_2^{-1}(3) = \{\ulcorner 3\urcorner, \ulcorner 03\urcorner, \ulcorner 003\urcorner, \ldots\}$.

Therefore, a valid substitution rule is $\forall \ulcorner d\urcorner \in D. \ulcorner d\urcorner$ is substitutable for $\mathcal{F}_2^{-1}(\mathcal{F}_2 \ulcorner d\urcorner)$.

That is, for a string of digits, $\ulcorner d\urcorner$, $\mathcal{F}_2$ maps $\ulcorner d\urcorner$ to a natural number, which in turn, may be mapped by $\mathcal{F}_2^{-1}$ to the set of all other strings of digits which are mapped to the same natural number by $\mathcal{F}_2$.

Another is $\forall \ulcorner e\urcorner \in E$, given a state, $\sigma \in (I \to \mathbb{N})$, $\ulcorner e\urcorner$ is substitutable for $\mathcal{F}_2^{-1}(\mathcal{E} \ulcorner e\urcorner \sigma)$. This example of a "semantic inverse" is used later.

The β-reduction rule depends upon the notion of a "free variable":

7.3.4.7 Free Variables

The definition of free variables is the natural one [12].

free(I,F) denotes the fact that the variable, I, is free in the expression, F. The definition of *free* is:

free(I,F) $\Leftrightarrow$ I *does not occur in the left hand side of a* $\ulcorner$let val ...$\urcorner$.

7.3.4.8 The Beta Reduction Rule R_β

$$R_\beta : free(\ulcorner x \urcorner, F) \Rightarrow \ulcorner \text{let val x = E in F} \urcorner \equiv \ulcorner F \urcorner [\ulcorner E \urcorner / \ulcorner x \urcorner]$$

7.3.4.9 The Alpha Reduction Rule R_α

$$R_\alpha : free(\ulcorner x \urcorner, F) \Rightarrow \ulcorner \text{let val x = E in F} \urcorner \equiv \ulcorner \text{let val y = E in} \urcorner \ulcorner F \urcorner [\ulcorner y \urcorner / \ulcorner x \urcorner]$$

R_α and R_β are syntactic substitution rules, so it is possible to check if they are valid. That is, is the language referentially transparent with respect to $(\mathcal{F}, R_\alpha)$ and $(\mathcal{F}, R_\beta)$?

The answer is that R_α is valid, but R_β is invalid because, for example :

$$P_2 \not\equiv \ulcorner \text{let val } I_2 = 7 \text{ in } 5 + 6 + I_2 \urcorner$$

Instead, the similar rule R, below is valid.

R is similar to the λ-Calculus β-reduction rule and to R_β, but is complicated by the involvement of the *meaning* of an expression.

R allows for an identifier to be replaced by a numeral which has the same *meaning* as the expression bound to the identifier in the let abstraction.

The rule can only be expressed in terms of the state of an identifier since the *meaning* of an expression involves the state.

$$R : free(I,F) \Rightarrow \mathcal{E} \ulcorner \text{let val I = E in F} \urcorner \sigma = \mathcal{E}(\ulcorner F \urcorner [\mathcal{F}_2^{-1}(\mathcal{E} \ulcorner F \urcorner \sigma) / \ulcorner I \urcorner])\sigma$$

The rule, R, is a rule which effectively says "if the programmer knows what value an expression in E evaluates to, then any numeral which evaluates to this value may be substituted for I in F".

This example shows that, for purely syntactic reasoning to be achievable, the syntax of the language has to be rich enough to cater for *all* algebraic identities. If this is not the case, then substitution rules will, *of necessity*, appeal to the semantic definition of the language.

In §7.5 a property called "Algebraic Closure" is defined. Languages which posses this attribute will have a sufficiently rich syntax for all expressions with an equivalent meaning to be substituted for one another.

7.3.4.10 Altering the Language to Allow a Syntactic Beta Reduction Rule

If the syntax of the language is enriched to include an arbitrary number of addition operators within an expression, then the rule R_β becomes valid:

7.3.4.10.1 Syntax

$$E ::= I \mid D \mid E_1 + E_2 \mid \text{let val } I = E_1 \text{ in } E_2$$

7.3.4.10.2 Semantics

$$\mathcal{E} : E \to (I \to \mathbb{N}) \to \mathbb{N}$$

$$\mathcal{E}\ulcorner I \urcorner\sigma = \sigma\ulcorner I \urcorner$$

$$\mathcal{E}\ulcorner D \urcorner\sigma = \mathcal{F}\ulcorner D \urcorner$$

$$\mathcal{E}\ulcorner E_1 + E_2 \urcorner\sigma = \mathcal{E}\ulcorner E_1 \urcorner\sigma + \mathcal{E}\ulcorner E_2 \urcorner\sigma$$

$$\mathcal{E}\ulcorner \text{let val } I = E_1 \text{ in } E_2 \urcorner\sigma = \mathcal{E}\ulcorner E_2 \urcorner(\sigma[\mathcal{E}\ulcorner E_1 \urcorner\sigma / \ulcorner I \urcorner])$$

In this language, not only do the R_α and R_β rules hold as valid substitution rules but the associativity law of addition holds.

This section concludes the preliminary investigation of the consequences of particular semantic descriptions of functional languages upon the substitution rules that these languages enjoy.

For more involved languages the substitution rules will be richer and more numerous, however, analysis may proceed in exactly the manner.

7.4 The Modelling strategy and the Implicit State

The pure functional language just described in §7.3.4 is now "corrupted" by the addition of syntax which, when executed causes output to occur. The corrupted language is called "Cor" for reference.

"Cor" is modelled by a language called "Mod" (the programs of which, when executed, do *not* cause output to occur).

The modelling strategy is described by definition of a function, $\mathcal{T}$, which maps syntactic elements of the class, F (programs of "Cor"), to E′ (programs of "Mod").

The syntactic mapping, $\mathcal{T}$, is then proved to be meaning-preserving thus proving that the modelling strategy is correct.

7.4.1 The Language "Cor"

7.4.1.1 Syntax

V ::= D ¦ I

E ::= E_1 + E_2 ¦ V ¦ let val I = E_1 in E_2

F ::= E ¦ print val I = E in F

7.4.1.2 Semantics

The function, $\mathcal{F}$, describes the meaning of the syntactic class, F, in terms of a mapping which includes *not only* the bindings for identifiers but the output produced thus far by execution of the program.

The semantics of a program in the language is described by a mapping $\mathcal{M}$.

Some new semantic functions are required to describe Cor: "<>" for appending two lists and the distfix functions "[...,...]" and "(...,...)" for constructing lists and tuples respectively. These functions have their normal mathematical properties.

$\mathcal{V} : V \to (I \to \mathbb{N}) \to \mathbb{N}$

$\mathcal{E} : E \to (I \to \mathbb{N}) \to \mathbb{N}$

$\mathcal{F} : ((I \to \mathbb{N}) \times list\ \mathbb{N}) \to (\mathbb{N} \times list\ \mathbb{N})$

$\mathcal{M} : (I \to \mathbb{N}) \to (\mathbb{N} \times list\ \mathbb{N})$

$\mathcal{V}\ulcorner D \urcorner\sigma = \mathcal{F}_2\ulcorner D \urcorner$

$\mathcal{V}\ulcorner I \urcorner\sigma = \sigma\ulcorner I \urcorner$

$\mathcal{E}\ulcorner E_1 + E_2 \urcorner\sigma = \mathcal{E}\ulcorner E_1 \urcorner\sigma + \mathcal{E}\ulcorner E_2 \urcorner\sigma$

$\mathcal{E}\ulcorner V \urcorner\sigma = \mathcal{V}\ulcorner V \urcorner\sigma$

$\mathcal{E}\ulcorner \text{let val I} = E_1 \text{ in } E_2 \urcorner\sigma = \mathcal{E}\ulcorner E_2 \urcorner(\sigma[\mathcal{E}\ulcorner E_1 \urcorner\sigma\ /\ \ulcorner I \urcorner])$

$\mathcal{F}\ulcorner E \urcorner(\sigma,\mathcal{L}) = (\mathcal{E}\ulcorner E \urcorner\sigma,\ \mathcal{L})$

$\mathcal{F}\ulcorner \text{print val I = E in F} \urcorner(\sigma,\mathcal{L}) = \mathcal{F}\ulcorner F \urcorner(\sigma'\ ,\mathcal{L}')$

where $\sigma' = \sigma[\mathcal{E}\ulcorner E \urcorner\sigma\ /\ \ulcorner I \urcorner]$

and $\mathcal{L}' = \mathcal{L} \diamond [\mathcal{E}\ulcorner E \urcorner\sigma\ ]$

$\mathcal{M}\ulcorner F \urcorner\sigma = \mathcal{F}\ulcorner F \urcorner(\sigma,\ [])$

7.4.2 The Model Notation, "Mod"

"Mod" is the language described in §7.3.4, with the addition of 2-tuples and lists.

7.4.2.1 Syntax

V ::= D ¦ I

⟨mt⟩ ::=

⟨LE⟩ ::= ⟨mt⟩ ¦ E′ , ⟨LE⟩

$E' ::= E'_1 + E'_2 = ¦ V ¦ \text{let val } I = E'_1 \text{ in } E'_2 ¦$
$(E'_1 , E'_2) ¦ E'_1 \diamond E'_2 ¦ [\ ⟨LE⟩\ ]$

7.4.2.2 Semantics

The semantics of "Mod" are described by the meaning function $\mathcal{E}'$.

Note, $\mathcal{V}$ is the meaning function for the syntactic class V, described earlier in §7.4.1.2.

$Ev = \mathbb{N} + list\ Ev + (Ev \times Ev)$

$\mathcal{E}' : E' \to (I \to Ev) \to Ev$

$\mathcal{E}'\ulcorner E'_1 + E'_2\urcorner\sigma = \mathcal{E}'\ulcorner E'_1\urcorner\sigma + \mathcal{E}'\ulcorner E'_2\urcorner\sigma$
$\mathcal{E}'\ulcorner V\urcorner\sigma = \mathcal{V}\ulcorner V\urcorner\sigma$
$\mathcal{E}'\ulcorner \text{let val } I = E'_1 \text{ in } E'_2\urcorner\sigma = \mathcal{E}'\ulcorner E'_2\urcorner(\sigma[\mathcal{E}'\ulcorner E'_1\urcorner\sigma\ /\ \ulcorner I\urcorner])$
$\mathcal{E}'\ulcorner E'_1 \diamond E'_2\urcorner\sigma = \mathcal{E}'\ulcorner E'_1\urcorner\sigma \diamond \mathcal{E}'\ulcorner E'_2\urcorner\sigma$
$\mathcal{E}'\ulcorner (E'_1 , E'_2)\urcorner\sigma = (\ \mathcal{E}'\ulcorner E'_1\urcorner\sigma,\ \mathcal{E}'\ulcorner E'_2\urcorner\sigma\ )$
$\mathcal{E}'\ulcorner [E'_1 , \ldots , E'_n]\urcorner\sigma = [\ \mathcal{E}'\ulcorner E'_1\urcorner\sigma , \ldots , \mathcal{E}'\ulcorner E'_n\urcorner\sigma\ ]$

7.4.3 The Model Function $\mathcal{T}$

The language "Cor" is modelled, using the language "Mod", by the syntactic function $\mathcal{T}$.

$\mathcal{T}$ maps members of F to members of E′.

$\mathcal{T}$ is defined by cases of the syntactic structure of F.

$\mathcal{T}$ uses two auxiliary functions $\mathcal{T}'$ and $\mathcal{T}''$ to translate elements of F and elements of E respectively.

$$\mathcal{T}'' : E \rightarrow E'$$
$$\mathcal{T}' : F \rightarrow E'$$
$$\mathcal{T} : F \rightarrow E'$$

$$\mathcal{T}''(D) = D$$
$$\mathcal{T}''(I) = I$$
$$\mathcal{T}''(E + E) = \mathcal{T}''(E) + \mathcal{T}''(E)$$
$$\mathcal{T}''(\text{let val } I = E \text{ in } E) = \text{let val } I = \mathcal{T}''(E) \text{ in } \mathcal{T}''(E)$$

$$\mathcal{T}'(\text{print val } I = E \text{ in } F) = \text{let val output} = \text{output} \diamond [\mathcal{T}''(E)] \text{ in}$$
$$\text{let val } I = \mathcal{T}''(E) \text{ in}$$
$$\mathcal{T}'(F)$$

$$\mathcal{T}'(E) = (\mathcal{T}''(E), \text{output})$$

$$\mathcal{T}(F) = \text{let val output} = [] \text{ in } \mathcal{T}'(F)$$

7.4.4 Examples

A couple of examples help to demonstrate the strategy employed by $\mathcal{T}$, and the advantage of using it.

7.4.4.1 Example 1

In this example, the sentence to be modelled is:

```
⌜let val x = 2 + 4 in
    let val y = x + 2 in
      x+y⌝
```

In this first example, the program causes *no* output to be produced, when executed. Thus its model is very simple.

The resulting model is built up from the applications of $\mathcal{T}''$ and $\mathcal{T}'$, to demonstrate the method used by the function $\mathcal{T}$ (this will be useful for the reader who wishes to verify the proof in §7.4.5).

The function $\mathcal{T}''$ maps syntax from F which is formed solely by the syntactic formation rule E. Such syntax does not cause any output and is mapped to identical syntax in E′.

```
T''(let val x = 2 + 4 in                    =    let val x = 2 + 4 in
      let val y = x + 2 in                           let val y = x + 2 in
        x+y )                                          x+y
```

The function $\mathcal{T}'$ reflects the fact that the final result of a program, must include the value of the output list created by execution, in addition to the result of evaluation of the expression in "Cor".

Thus

```
T'(let val x = 2 + 4 in                     =    ((let val x = 2 + 4 in
      let val y = x + 2 in                           let val y = x + 2 in
        x+y )                                          x+y), output)
```

The function $\mathcal{T}$, calls the auxiliary function, $\mathcal{T}'$, to perform the modelling of the various expression constructs of F, all that is then required, is for $\mathcal{T}$ to model the fact that, initially, the output is empty.

Thus

```
T(let val x = 2 + 4 in                      =    let val output = [] in
      let val y = x + 2 in                           ((let val x = 2 + 4 in
        x+y )                                          let val y = x + 2 in
                                                         x+y), output)
```

7.4.4.1.1 Substitution

There are no substitution rules in "Cor" that allow the programmer to infer *anything* about the output of a program. Of course, the programmer can use extra-linguistic reasoning, such as "the program contains no 'prints' and so creates no output".

In "Mod" however, the Output is *explicitly* mentioned (as a list). Nothing is "special" about this list, so the normal substitution rules of the language apply.

That is, reasoning about output can be conducted *within the language itself.*

In this case the model can be manipulated to:

(14 ,[])

This tells the programmer that there is no output and that the program evaluates to 14.

Of course, in this simple example there is little difference between the manipulation of the program to reveal facts about I/O and the extra-linguistic reasoning that could be used with the language "Cor". However, for programs which exhibit more complicated output behaviour, the ability to manipulate a program according to its syntactic rules will be a considerable advantage.

7.4.4.2 Example 2

The sentence to be modelled is:

```
⌜print val x = 2 + 4 in
   print val y = x + 2 in
   x+y⌝
```

This example is identical to the last, except that the values of "x" and "y" are printed out as they are evaluated.

Only those elements of the program which are drawn from the syntax class E are modelled by $\mathcal{T}''$. The following three expressions are the elements of the sentence which are drawn from the syntactic class E.

$\mathcal{T}''$(x+y)	=	x+y
$\mathcal{T}''$(2 + 4)	=	2 + 4
$\mathcal{T}''$(x + 2)	=	x + 2

The function $\mathcal{T}'$ models the side-affecting constructs (i.e. those in the syntactic class F):

```
T'(print val x = 2+4 in                    let val output = output <> [2+4] in
   print val y = x+2 in          =           let val x = 2+4 in
     x+y)                                      let val output = output <> [x+2] in
                                                 let val y = x+2 in
                                                   (x+y, output)
```

The function, $\mathcal{T}$, merely includes the initial definition for the initial value of the output list:

```
T(print val x = 2+4 in                     let val output = [] in
   print val y = x+2 in          =           let val output = output <> [2+4] in
     x+y)                                      let val x = 2+4 in
                                                 let val output = output <> [x+2] in
                                                   let val y = x+2 in
                                                     (x+y, output)
```

This example clearly illustrates that the "cost" of making the implicit parts of the state explicit (i.e. as expressible values): Namely, it makes for larger programs. However, the benefits of the approach far outweigh the initial disadvantage of a textually longer program.

Besides, the *complexities* of the model and the modelled programs are *identical*: they are both mapped to the *same* semantic value. The model program actually *evaluates* to a syntactic representation of this semantic value, whereas the original program does *not*.

This accounts for and justifies the extra notation in the model.

7.4.4.2.1 Substitution

Once again, with the language "Cor", extra-linguistic reasoning could be used to discuss the output of the program.

In the model, however, the "normal form" is:

```
(14, [6,8])
```

This tells the programmer that the "Cor" program evaluates to 14, and produces as output a six followed by an eight.

7.4.5 Correctness Proof For the Model Function $\mathcal{T}$

In this section a proof is presented that the model algorithm embodied in $\mathcal{T}$, correctly models programs in "Cor". This is done by showing that the *meaning* of any program, $\ulcorner f \urcorner$, drawn from the domain F, is identical to the *meaning* of the model $\ulcorner \mathcal{T}(f) \urcorner$.

Such a proof guarantees the correctness of the modelling strategy with respect to the meaning of the language being modelled.

Formally:

$$\forall \ulcorner f \urcorner \in F \;.\; \mathcal{M} \ulcorner f \urcorner = \mathcal{E}' \ulcorner \mathcal{T}(f) \urcorner$$

(provided $\ulcorner f \urcorner$ does not mention the identifier $\ulcorner output \urcorner$)

The proof consists of two lemmas which prove the relevant property for the modelling algorithms $\mathcal{T}''$ and $\mathcal{T}'$, and a main proof which uses these lemmas to justify the initial let abstraction ($\ulcorner$let val output = []$\urcorner$ in ...).

The modelling algorithm, $\mathcal{T}$, is only valid, of course, if the program to be modelled does *not* mention the identifier $\ulcorner output \urcorner$, since this identifier is *introduced* to model the output list. This is not a matter for concern, since the model strategy could be *parameterised* by the name of an (unused) identifier, from which the model would then be constructed. This has not been done here since it clouds the proof, and is, in any case, equivalent to the assumption that $\ulcorner output \urcorner$ is an unused identifier.

Notation

To make the proofs easier to read, a notational shorthand is introduced for a kind of restriction operator, $\lceil$, on states:

$$\forall x.x \neq \ulcorner I \urcorner \Rightarrow \sigma' x = \sigma x$$

will be written:

$$\sigma' \lceil \ulcorner I \urcorner = \sigma$$

Lemma 1

if $\sigma' \lceil \ulcorner \text{output} \urcorner = \sigma$

and $\ulcorner e \urcorner$ does not mention $\ulcorner \text{output} \urcorner$

then $\mathcal{E}' \ulcorner \mathcal{T}''(e) \urcorner \sigma' = \mathcal{E} \ulcorner e \urcorner \sigma$

where $\ulcorner e \urcorner \in E$

This lemma cannot be proved by appealing to the fact that $\mathcal{T}''$ is *syntactically* the identity function, because this is *no guarantee* that $\mathcal{T}''$ is *semantically* the identity function. It is the proof of lemma 1 that gives this semantic guarantee.

The proof is by simple induction on the syntactic cases for $\ulcorner e \urcorner$, which also form the cases in the definition of $\mathcal{T}''$.

Proof of Lemma 1

Two Bases Cases for $\ulcorner e \urcorner$:

1) $\ulcorner e \urcorner \equiv \ulcorner d \urcorner \in D$, in which case

$\mathcal{E} \ulcorner d \urcorner = \mathcal{V} \ulcorner d \urcorner = \mathcal{E}' \ulcorner d \urcorner = \mathcal{E}' \ulcorner \mathcal{T}''(d) \urcorner$

2) $\ulcorner e \urcorner \equiv \ulcorner i \urcorner \in I$, in which case

$\mathcal{E} \ulcorner i \urcorner = \mathcal{V} \ulcorner i \urcorner = \mathcal{E}' \ulcorner i \urcorner = \mathcal{E}' \ulcorner \mathcal{T}''(i) \urcorner$

Induction Hypothesis:

$\sigma' \lceil \ulcorner \text{output} \urcorner = \sigma$

and $\ulcorner e_1 \urcorner$ *and* $\ulcorner e_2 \urcorner$ do not mention $\ulcorner \text{output} \urcorner$

and $\mathcal{E}' \ulcorner \mathcal{T}''(e_1) \urcorner \sigma' = \mathcal{E} \ulcorner e_2 \urcorner \sigma$

and $\mathcal{E}' \ulcorner \mathcal{T}''(e_1) \urcorner \sigma' = \mathcal{E} \ulcorner e_2 \urcorner \sigma$

Where $\ulcorner e_1 \urcorner$ and $\ulcorner e_2 \urcorner$ are arbitrary members of E.

Two Inductive Cases for $\ulcorner e \urcorner$:

1) $\ulcorner e \urcorner \equiv \ulcorner e_1 + e_2 \urcorner$ in which case

$\mathcal{E}\ulcorner e_1 + e_2 \urcorner\sigma$

$= \mathcal{E}\ulcorner e_1 \urcorner\sigma + \mathcal{E}\ulcorner e_2 \urcorner\sigma$ (by definition of $\mathcal{E}$)

$= \mathcal{E}'\ulcorner(\mathcal{T}''(e_1))\urcorner\sigma' + \mathcal{E}\ulcorner(\mathcal{T}''(e_1))\urcorner\sigma'$ (from Induction Hypothesis)

$= \mathcal{E}'\ulcorner(\mathcal{T}''(e_1 + e_2))\urcorner$ (by definition of $\mathcal{E}'$).

2) $\ulcorner e \urcorner \equiv \ulcorner$let val I = e_1 in $e_2\urcorner$ in which case

$\mathcal{E}\ulcorner$let val I = e_1 in $e_2\urcorner\sigma$

$= \mathcal{E}\ulcorner e_2 \urcorner(\sigma[\mathcal{E}\ulcorner e_1 \urcorner\sigma \,/\, \ulcorner I \urcorner])$ (by definition of $\mathcal{E}$)

$= \mathcal{E}\ulcorner e_2 \urcorner(\sigma[\mathcal{E}'\ulcorner\mathcal{T}''(e_1)\urcorner\sigma' \,/\, \ulcorner I \urcorner])$ (from induction hypothesis)

$= \mathcal{E}'\ulcorner\mathcal{T}''(e_2)\urcorner(\sigma'[\mathcal{E}'\ulcorner\mathcal{T}''(e_1)\urcorner\sigma' \,/\, \ulcorner I \urcorner])$ (from induction hypothesis)

$= \mathcal{E}'\ulcorner$let val I = $\mathcal{T}''(e_1)$ in $\mathcal{T}''(e_2)\urcorner\sigma'$ (by definition of $\mathcal{E}'$).

Lemma 2

if $\sigma'\ulcorner$output$\urcorner = \mathcal{L}$

and $\sigma' \lceil \ulcorner$output$\urcorner = \sigma$

and $\ulcorner f \urcorner$ does not mention $\ulcorner$output$\urcorner$

then $\mathcal{F}\ulcorner f \urcorner(\sigma, \mathcal{L}) = \mathcal{E}'\ulcorner\mathcal{T}'(f)\urcorner\sigma'$

where $\ulcorner f \urcorner \in F$

This lemma asserts that provided the model, $\ulcorner\mathcal{T}'(f)\urcorner$, is evaluated in an appropriate state, then the model will be correct. An appropriate state is one where all identifier bindings other than $\ulcorner$output$\urcorner$ have identical values to those in the state of the modelled program, σ, and where the binding for the identifier $\ulcorner$output$\urcorner$ is the value of the output list, $\mathcal{L}$, for the modelled program (i.e. the output produced "so far").

The lemma also requires that the identifier $\ulcorner$output$\urcorner$ is not mentioned in the program $\ulcorner f \urcorner$.

The proof is by induction on the syntactic cases for $\ulcorner f \urcorner$, which also form the cases in the definition of $\mathcal{T}'$.

Proof of Lemma 2

Base Case for $\ulcorner f \urcorner$:

$\ulcorner f \urcorner \equiv \ulcorner e \urcorner \in E$, in which case

$\mathcal{F}\ulcorner e \urcorner(\sigma,\mathcal{L}) = (\mathcal{E}\ulcorner e \urcorner\sigma,\mathcal{L})$

$= (\mathcal{E}'\ulcorner(\mathcal{T}''(e)\urcorner\sigma',\mathcal{L})$ (by lemma 1 and definition of lemma 2)

$= (\mathcal{E}'\ulcorner(\mathcal{T}''(e)\urcorner\sigma',\sigma'\ulcorner output \urcorner)$ (by definition of lemma 2)

$= \mathcal{E}'\ulcorner(\mathcal{T}''(e),output)\urcorner\sigma'$ (by definition of $\mathcal{E}'$)

$= \mathcal{E}'\ulcorner\mathcal{T}'(e)\urcorner\sigma'$ (by definition of $\mathcal{T}'$)

Induction Hypothesis:

$\sigma'\ulcorner output \urcorner = \mathcal{L}$

and $\sigma'\lfloor\ulcorner output \urcorner = \sigma$

and $\ulcorner f \urcorner$ does not mention $\ulcorner output \urcorner$

and $\mathcal{F}\ulcorner f \urcorner(\sigma,\mathcal{L}) = \mathcal{E}'\ulcorner\mathcal{T}'(f)\urcorner\sigma'$

Where $\ulcorner f \urcorner \in F$.

Required to Show:

$\mathcal{F}\ulcorner$print val I = e in f$\urcorner(\sigma,\mathcal{L}) = \mathcal{E}'\ulcorner\mathcal{T}'$(print val I = e in f)$\urcorner\sigma'$

Where $\ulcorner e \urcorner$ is an arbitrary member of the syntactic domain E.

LHS = $\mathcal{F}\ulcorner f \urcorner(\sigma[\mathcal{E}\ulcorner e \urcorner\sigma \,/\, \ulcorner I \urcorner],\ \mathcal{L} \diamond [\mathcal{E}\ulcorner e \urcorner\sigma])$

(by definition of $\mathcal{F}$)

RHS = $\mathcal{E}'\ulcorner$let val output = output $\diamond [\mathcal{T}''(e)]$ in

let val I = $\mathcal{T}''(e)$ in $\mathcal{T}'(f)\urcorner\ \sigma'$

(by definition of $\mathcal{T}'$)

$= \mathcal{E}'\ulcorner\mathcal{T}'(f)\urcorner(\sigma'[\mathcal{E}'\ulcorner\mathcal{T}''(e)\urcorner\sigma' \,/\ulcorner I \urcorner,\ \mathcal{E}'\ulcorner output \diamond [\mathcal{T}''(e)]\urcorner\sigma' \,/\ulcorner output \urcorner])$

(by definition of $\mathcal{E}'$)

NOW (by lemma 1 and induction hypothesis and definition of $\mathcal{E}'$)

$$\mathcal{E}'\ulcorner\mathcal{T}''(e)\urcorner\sigma' = \mathcal{E}\ulcorner e\urcorner\sigma$$

and $\mathcal{E}'\ulcorner output <> [\mathcal{T}''(e)]\urcorner\sigma'$

$= \mathcal{E}'\ulcorner output\urcorner\sigma' <> \mathcal{E}'\ulcorner[\mathcal{T}''(e)]\urcorner\sigma'$

$= \mathcal{L} <> \mathcal{E}\ulcorner e\urcorner\sigma$

SO

RHS $= \mathcal{E}'\ulcorner\mathcal{T}'(f)\urcorner\sigma''$

where $\sigma'' = (\sigma'[\mathcal{E}\ulcorner e\urcorner\sigma / \ulcorner I\urcorner, \mathcal{L} <> [\mathcal{E}\ulcorner e\urcorner\sigma / \ulcorner output\urcorner])$

NOW $\sigma''\ulcorner output\urcorner = \mathcal{L} <> [\mathcal{E}\ulcorner e\urcorner\sigma]$ (by definition of σ')

and $\sigma''\lceil\ulcorner output\urcorner = \sigma[\mathcal{E}\ulcorner e\urcorner\sigma / \ulcorner I\urcorner]$ (since $\ulcorner I\urcorner \not\equiv \ulcorner output\urcorner$)

and $\ulcorner f\urcorner$ does not mention $\ulcorner output\urcorner$ (by definition of lemma 2)

SO

RHS $= \mathcal{F}\ulcorner f\urcorner(\sigma[\mathcal{E}\ulcorner e\urcorner\sigma / \ulcorner I\urcorner], \mathcal{L} <> [\mathcal{E}\ulcorner e\urcorner\sigma])$

(by induction hypothesis)

= LHS as required.

Main Proof

The main proof uses **Lemma 2** to establish:

$\mathcal{M}\ulcorner f\urcorner = \mathcal{E}'\ulcorner\mathcal{T}(f)\urcorner$ provided $\ulcorner f\urcorner$ does not mention $\ulcorner output\urcorner$.

That is, the meaning of a program in $\ulcorner f\urcorner$ is identical to the meaning of its model $\ulcorner\mathcal{T}(f)\urcorner$ provided $\ulcorner f\urcorner$ does not mention $\ulcorner output\urcorner$..

Proof

LHS = $\lambda\sigma.\mathcal{F}\ulcorner f\urcorner(\sigma;[])$ (by definition of $\mathcal{M}$)

RHS = $\mathcal{E}'\ulcorner$let val output = [] in $\mathcal{T}'(f)\urcorner$ (by definition of $\mathcal{T}$)

$= \lambda\sigma.\mathcal{E}'\ulcorner\mathcal{T}'(f)\urcorner\sigma'$

where $\sigma' = \sigma[\ [] \ / \ \ulcorner output\urcorner]$ (by definition of $\mathcal{E}'$)

NOW $\sigma' \lceil \ulcorner output\urcorner = \sigma$ (by definition of σ')

and $\sigma'\ulcorner output\urcorner = []$ (by definition of σ')

and $\ulcorner f\urcorner$ does not mention $\ulcorner output\urcorner$ (by definition)

SO

RHS = $\mathcal{F}\ulcorner f\urcorner(\sigma,\mathcal{L})$ (by **Lemma 2**)

= LHS as required.

7.5 Algebraic Closure

The language described in §7.3.4 demonstrated that it is sometimes not possible to substitute a value that "one would expect to substitute" simply because the syntax of the language is not rich enough to permit such a substitution.

Given a semantic description of the language, it seems that what makes a substitution reasonable is this: "Any two pieces of syntax which have equivalent meaning should be substitutable".

Some care is required in this definition however. As Stoy points out (page five of [16]) it would not be reasonable to substitute "6" for "1+5" in "21+57".

As this example demonstrates, the parser actually *does* play *some* part in the description of the semantics of the language. In semantic descriptions it is normal to assume that sentences in the language have "already" been parsed (according to a grammar), into an abstract syntax.

Now, for any particular abstract syntax, there may be many ways of *parsing* a particular sentence into this abstract syntactic structure.

This is precisely the part played by the parser in the definition of the mapping from sentences of the language to their meaning.

In this discussion, and throughout the literature on Semantics, it is assumed that the mapping from sentences of the language to abstract syntax is not contentious. If elucidation is required, then a grammar will be necessary, in addition to the abstract syntax.

Thus, in the discussion of substitution rules, it is the *abstract* syntactic classes that will be substituted for, in *abstract* syntactic expressions, rather than *concrete* syntactic sub strings within *concrete* syntactic strings.

7.5.1 A Definition of Algebraic Closure

For a semantic function, $\mathcal{E}$, mapping elements of a syntactic class, E, into a semantic domain, S, the set $\forall\mathcal{E}$, shall denote the set of possible semantic values produced by $\mathcal{E}$. That is $\forall\mathcal{E} = \{ \mathcal{E}\ulcorner e\urcorner : \ulcorner e\urcorner \in E \}$.

CHAPTER EIGHT

CONCLUSIONS

CHAPTER EIGHT CONCLUSIONS

Compared to procedural notations, functional notations are rich in algebraic properties and enjoy a variety of flexible proof techniques. Chapter two briefly sets out the differences between the procedural and functional styles of programming.

There are several techniques in print [44,61,79,80] and in the "folklore" of computer programming, which allow a programmer to convert a procedural programming construct into a functional notation.

This thesis shows that these techniques are only valid when the state of computation is simply an environment mapping from identifiers in a program's syntax to the semantic values which it computes.

The thesis makes a distinction between the explicit state (the environment mapping) and the implicit state (every other state-component).

Chapter three describes a general method for modelling the implicit state with an explicit state. Chapter three thus unifies the existing, ad hoc, techniques for modelling and extends the application area of these techniques to include any and every procedural program.

In chapter four a simple technique called "Abstraction Projection" is introduced. This technique allows a programmer to produce many distinct model functions for one procedural program. Each model is projected onto a small set of the semantic values. The technique gives a programmer the ability to focus on a particular aspect of computation and "abstract away" from all other details of computation which are irrelevant.

Chapter five presents some strategies for converting functional models back into a procedural notation.

The Functional Modelling Approach thus allows a procedural programmer to use functional programs to analyse *any* semantic value computed by *any* procedural program.

Functional programs are ideal for analysis due to their rich algebraic properties and since Projection Abstraction allows *any* semantic value to be the result of a model, the strategy can be applied to a wide variety of application areas. Some of the possible applications of the strategy are demonstrated and discussed in chapter six. Specifically these are:

Structural alteration (reverse engineering)
Error-detection and removal
Efficiency improvement
Specification generation and proof construction
Language conversion

Many other application areas may be discovered in the future. Some of these are briefly described in chapter nine:

Complexity analysis
Compile-time Garbage collection
Parallel execution path analysis (chapter six contains an example)

The application areas of the modelling strategy clearly include any form of compile-time analysis that may be performed upon a procedural program.

The semantic foundations of this thesis rest upon the notion of "Referential Transparency" and the algebraic treatment of programs. The first six chapters also implicitly assume a definition of what exactly constitutes "procedural programming" and "functional programming".

Chapter seven discusses these semantic assumptions, turning them from assumptions into formal definitions.

The essential contributions are:

A formal definition of "Functional Programming Language".
A formal definition of "Procedural Programming Language".
A formal definition of "Valid Substitution Rule", in the context of which a discussion of "Referential Transparency" is presented.
A formal definition of a new concept: "Algebraic Closure", which can be used to "iron out" substitutive "special cases" in a programming language.

Chapter Nine

FUTURE WORK

9.1 Modelling Strategies

Chapter seven shows how a programmer could use the semantic description of a procedural programming language to construct a modelling strategy for the language and to prove this strategy correct with respect to the semantic description.

Sadly, there are no semantic descriptions available for commonly used procedural languages such as C and Pascal.

One task for future work is to describe a semantics for such programming languages and to construct and prove modelling strategies for them.

9.2 CASE Tools for Reverse Engineering

Many of the techniques described in this thesis, such as model production and Projection Abstraction may be performed automatically.

Implementation of these automatable techniques would create a powerful and flexible CASE tool for the analysis and manipulation of procedural programs.

9.3 Application Areas

There are many application areas of this work which have not yet been fully examined. Some of these are listed below:

9.3.1 Compile-Time Garbage Collection

By choosing the heap and heap pointer as the semantic domain onto which a model is projected, a programmer will be able to analyse the way in which the heap store is allocated, used and disposed of.

This analysis may lead to improvements in the use of store in a similar manner to those described by Darlington in [91].

9.3.2 Parallel Execution Paths

Functional languages are deemed amenable to parallel evaluation [1,58].

As such, the modelling strategy may be used to expose the paths available within the program which can be executed in parallel.

A simple example of this possibility is described in chapter six (§6.7).

9.3.3 Homogenisation of Data Structures

One analysis that has not been performed in the examples contained in chapter six, is that of homogenisation of data structuring.

Large programs which are badly designed, often have no organised approach to the storage of data.

With the automation of modelling techniques will come the ability to analyse the underlying data structures demanded by a program, and ought to lead to the creation of a set of techniques for homogenisation of the storage requirements of a program.

9.4 Semantic Foundations

It is surprising that terminology such as "functional language", "procedural language" and the like are used so freely, when such terms are so ill defined.

This lack of definition is all the more surprising when one considers the vast body of work on semantics of programming language which could be used to resolve the inadequacy.

Chapter seven attempts a formal definition of the notion of "functional" and "procedural" programming, but more work is required to complete such definitions.

A similar approach could also be used in the definition of other undefined terms in common parlance, such as "Object Oriented Programming", "Low Level language", "More/Less Expressive" and so on.

The are many advantages of a formal semantic approach to the definition of these terms (over and above the certainty that such definitions would produce).

For example, a language which meets the requirements set out in the definition of the term "functional", will also be the subject of any theorems that can be derived from these requirements. Thus a formal semantic definition of "functional" offers similar benefits to the formal description of an "Abstract Data Type". One could speak of an "Abstract Linguistic Type". Theorems would be constructed in terms of the abstract linguistic definition and will be applicable to all programs which satisfy the definition.

9.5 Language Design

It seems, from the work presented in chapter three, that many procedural languages only lack algebraic properties of substitution because of the implicit state. Perhaps it might be possible to define a language which is as efficient as any procedural language, but which does not sacrifice algebraic flexibility in order to achieve this efficiency.

APPENDIX A1

SUBSET OF ML USED IN THIS THESIS

A1 The SubSet Of ML Used

This appendix describes the subset of ML used in modelling. The notation used is Extended Backus Naur Form. The syntax described here is a very small subset of the total ML syntax. Once a model has been created in ML, of course, then manipulation can be used to produce a modified model, which may use *any* (possibly larger set) of the ML syntax.

A syntax description of the entire ML language can be found in [41].

A brief tutorial on the subset of ML used here for modelling is given in §2.2.

The subset of ML used depends upon the expression syntax, E, of the procedural language.

```
<ML>           ::=  <declarations> <definitions>
<definitions>  ::=  <definition> ¦ <definition> <definitions>
<definition>   ::=  fun <identifier> (<formals>) = <Exp> ;
<Exp>          ::=  <E> ¦
                    if <E> then <Exp> else <Exp>  ¦
                    let val <identifier> = <E> in <Exp>  ¦
                    let fun <identifier>( <formals> )  =
                          <Exp> in <Exp>                    ¦
                    <identifier> ( <actuals> )   ¦
                    let val (<formals>) =
                           <identifier>(<actuals>) in <Exp>  ¦
                    <Built_in> ( <actuals> )
<formals>      ::=  <Empty> ¦ <formal> , <formals>
<actuals>      ::=  <Empty> ¦ <actual> , <actuals>
<Empty>        ::=
<formal>       ::=  <identifier>
<actual>       ::=  <E>
<Built_in>     ::=  cons ¦ empty ¦ Null ¦ Head ¦ Tail ¦ Append
<declarations> ::=  <Empty> ¦ <declaration> ; <declarations>
<declaration>  ::=  val <identifier> = <E>
```

APPENDIX A2

IMPLEMENTATION

A2 Implementation

In order to make any firm conclusions about the suitability of the modelling technique for analysing and altering the structure of large programs it will be necessary to implement a "CASE tool" which will embody the modelling strategy, and will also provide various "standard" manipulation techniques.

Such a tool could also contain various strategies like those outlined in chapters three, four and five.

It is envisaged that as the system is used, extra heuristic strategies for analysis and manipulation will be added to the system's repertoire.

A small prototype program has been implemented on the SUN Microsystems which converts Fortran IV into ML, according to the strategy outlined in chapter three.

The program is written in compiled Hope [51].

However, much more work is needed to implement a more general system which will treat all procedural languages in a unified manner, and which has manipulation and converting back strategies.

Work towards this goal has been undertaken [74], but more time and resources are required.

References

1. J. Darlington, M.D. Cripps, A.J. Field, P.G. Harrison & M.J. Reeve : "The Design and Implementation of ALICE : A Graph-reduction machine".
Data flow and Reduction Architectures. ed Thakker. IEEE Press 1987.

2. J. Darlington & R.M. Burstall : "A Transformation System for Developing Recursive Programs".
Journal of the ACM 24(1), 44-67. 1977.

3. C.A.R Hoare : "An Axiomatic Basis for Computer Programming".
Communications of the ACM 12, 576-580. 1969.

4. A.M. Turing : "Checking a Large Routine".
Conference on High-Speed Automatic Calculating Machines, Cambridge, 1949.

5. P. Naur : "Proof of Programs by General Snap-Shots".
BIT 6, 310-316. 1966.

6. R.W. Floyd : "Assigning Meanings to Programs".
Proceedings of Symposia in Applied Mathematics 19, 19-32. 1967.

7. C.A.R. Hoare, E.W. Dijkstra & O.J. Dahl : "Structured Programming".
Academic Press, New York. 1972.

8. K.R. Apt : "Ten Years of Hoare's Logic".
Communications of the ACM 3,4,431-483. 1981.

9. P.J. Landin : "A Formal Description of Algol 60".
Formal Language Description Languages, ed T.B. Steel, North-Holland Publishing, 1966.

10. J. M^{C}Carthy : "A Formal Description of a Subset of Algol 60".
Formal Language Description Languages, ed T.B. Steel, North-Holland Publishing, 1966.

11. C. Strachey : "Towards a Formal Semantics".
Formal Language Description Languages, ed T.B. Steel, North-Holland Publishing, 1966.

12. A. Church : "The Calculi of Lambda Conversion".
Annals of Mathematical Study 6. Princeton University Press, 1941.

13. D. Scott : "Outline of a Mathematical Theory of Computation".
Proceedings of the fourth Princeton Conference on Information Sciences and Systems. Princeton 1970.

14. D. Scott & C. Strachey : "Toward a Mathematical Semantics for Computer Programs".
Technical Monograph PRG 6. Oxford University Programming Research Group. 1971.

15. D. Scott : "Data Types as Lattices".
SIAM Journal of Computing 5, 1976.

16. J.E. Stoy : "Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory".
MIT Press, 1977.

17. S. Abramski & C.J. Hankin : "Abstract Interpretation of Declarative Languages".
Ellis-Horwood, 1987.

18. C. Reade : "Elements of Functional Programming".
Addison-Wesley, 1989.

19. C. Hankin, H. Glaser & G. Till : "Principals of Functional Programming".
Prentice-Hall, 1984.

20. J. Darlington, D. Turner & P. Henderson (eds) : "Functional Programming : Applications and Implementations".
Cambridge University Press, 1982.

21. P.G. Harrison & A.J. Field : "Functional Programming".
Adison-Wesley, 1988.

22. J. Darlington : "Program Transformation".
in [20].

23. R.M. Burstall : "Proving Properties of Programs Using Structural Induction".
The Computer Journal 12, 41-48, 1969.

24. S. Eisenbach & C.Sadler : "Why Functional Programming?".
in [78]

25. D. Friedman and D. Wise : "CONS Should Not Evaluate its Arguments".
In "Automata, Languages and Programming", Edinburgh University Press, 1976.

26. P. J. Landin : "A Correspondence Between Algol-60 and Church's Lambda Calculus", Communications of the ACM, 8, 3

27. M. Harman & S. Danicic : "SOL1 : A First Prototype Statistically Oriented Language".
Internal Publication of the School of Computer Science, North London Polytechnic, December 1991. (#2).

28. C. A. R. Hoare : "Communicating Sequential Processes".
Prentice Hall, 1985.

29. D. Bull, I Morrey & J Pugh : "Software Engineering : A Programming Approach".
Pentice-Hall, 1987.

30. P. Henderson : "Purely Functional Operating Systems".
in [20].

31. M. Attkinson : "Reverse Engineering".
Internal Publication of the School of Computer Science, North London Polytechnic, 1990. (#1)

32. D.A. Turner : "Recursion Equations as a Programming Language". in [20].

33. C.A.R. Hoare : "A Proof of the Program FIND". Communications of the ACM 14,1, 1971.

34. C. A. R. Hoare : "Procedures and Parameters : An Axiomatic Approach". Lecture Notes in Mathematics, vol. 188, Springer Verlag, NY, 1971.

35. J.E. Stoy : "Mathematical Aspects of Computer Programs" . in [20].

36. M.J.C. Gordon : "The Denotational Description of Programming Languages, An Introduction".
Springer-Verlag, 1979.

37. M. Harman & S. Danicic : "Programming Languages for Statistical Computation".
CompStat Conference on Computational Statistics 1990. Proceedings Physica Verlag, Berlin, 1990.

38. B. Gilchrist & A. Scallan : "Funigirls : A Prototype Functional Language for the Analysis of Generalized Linear Models".
Compstat Conference on Computational Statistics, 1988. Proceedings Physica Verlag, Berlin, 1988.

39. H Abelson & G.J Sussman : "Structure and Implementation of Computer Programs".
MIT Press, 1985.

40. J. Darlington, P. Harrison. H. Khoshnevisan, L. McLaughlin, N. Perry, H. Pull, M. Reeve, K. Sephton, L. While & H. Wright : "A Functional Programming Environment Supporting Execution, Partial Evaluation and Transformation".
Parle Conference on Parallel Computation 1989.

41. A Wikstrom : "Functional Programming Using Standard ML".
Prentice Hall Series in Computer Science, 1987.

42. C.A.R. Hoare & N. Wirth : "An Axiomatic Definition of the Programming Language PASCAL".
Acta Informatica, Vol 2, p335-355, 1973.

43. E.W. Dijkstra : "A Discipline of Programming".
Prentice-Hall, 1976.

44. J. M^{C}Carthy : "Towards a Mathematical Theory of Computation".
Proceedings of the International Foundations of Information Processing Congress, Munich, Germany, C.M Poppelworth (ed). 1962.

45. C. Strachey & C. Wadsworth : "Continuations : A Mathematical Semantics for Handling Full Jumps".
Technical Report, PRG-11, Oxford University Computer Lab., Programming Research Group, 1974.

46. R. DeMillo, R. Lipton & A Perlis : "Social Processes and Proofs of Theorems and Programs".
Communications of the ACM Vol. 22, No. 5. 1979.

47. R Mathews : "The Chip With a Sting in its Tail".
New Scientist, 13th. July 1991. No. 1777.

48. J. Fetzer : "Program Proving : The Very Idea".
Communications of the ACM Vol. 31, No. 9. 1988.

49. K. Gödel : "Uber Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter System I".
Monatstschefte Math. Phys. 38, p173-98.1931.
English Translation in : "The Undecidable", Raven New York. M. Davis (ed). 1965.

50. J. Vuillemin : "Proof Techniques for Recursive Programs".
Ph. D. Thesis, Computer Science Dept. Stanford, California.

51. R.M. Burstall, D.B. M^{ac}Queen & D.T. Sanella : "HOPE : an Experimental Applicative Language".
Internal Report, Dept. of Computer Science. University of Edinburgh. 1980.

52. D.A. Turner : "An Overview of Miranda". SIGPLAN Notices 21, 158-166. 1986.

53. D.A. Harrison : "Report of use of Functional Languages".
Usenet : comp.lang.functional (D.A.Harrison@uk.ac.newcastle).

54. I. Moor : "Realistic Functional Programming".
in [78].

55. D. Scott : "Models for Various Type-Free Calculi".
Proceedings of the Fourth International Conference on Logic, Methodology and the Philosophy of Science, Bucharest, P. Suppes *et al* (eds). North Holland, Amsterdam. p157-187. 1973.

56. H.P. Barandregt : "The Lambda Calculus : Its Syntax and Semantics"
North Holland Studies in Logic and Foundations of Mathematics, vol 103. North Holland, Amsterdam. 1984.

57. Z. Manna : "Mathematical Theory of Computation".
McGraw-Hill 1974.

58. M. Cripps, T. Field & M. Reeve : "An Introduction to ALICE : A Multiprocessor Graph-Reduction Machine".
in [78].

59. S.C. Kleene : "Introduction to Mathematics".
Van Norstrand, N.Y. 1950.

60. Z. Manna, S. Ness & J. Vuillemin : "Inductive Methods for Proving Properties of Programs".
Communications of the ACM Vol. 16, No. 8. 1973.

61. J. McCarthy : "A Basis for a Mathematical Theory of Computation".
in "Computer Programming and Formal Systems". P. Brafford & D. Hirschuerg (eds). North Holland.

62. N. Wirth : "Algorithms + Data Structures = Programs".
Prentice-Hall, New Jersey. 1976.

63. B. Kernighan & D.M. Ritchie : "The C Programming Language".
Prentice Hall, Software Series, 1978 (Reprinted Second Edition 1988).

64. P.J. Landin : "The Next 700 Programming Languages".
Communications of the A.C.M. 9, pp157-164. 1966.

65. M. C. Henson : "Elements of Functional Languages".
Blackwell, 1987.

66. B. J. MacLennan : "Functional Programming : Practice and Theory".
Addison Wesley, 1990.

67. H Shildt : "Modula-2 Made Easy"
Osbourne McGraw-Hill.

68. R.L. While : "Behavioural Aspects of Term Rewriting Systems".
Ph.D Thesis. Functional Programming Research Group. Imperial College. 1987.

69. W. Stoye : "Message-Based Functional Operating Systems".
Science of Computer Programming, vol 6, No 3. 291-311.

70. S Abramski : "Reasoning About Concurrent Systems".
in "Distributed Computing System Programme". I.E.E. Digital Electronics & Computing, Series 5. D. Duce(ed), 1984.

71. N Perry : "Hope$^+$C : A Continuation extension for Hope$^+$"
Internal Report of the Imperial College Functional Programming Research Group. ref IC/FPR/LANG/2.5.1/21. 1987.

72. R. Bailey : "Functional Programming with Hope".
Ellis-Horwood, 1990.

73. D.E. Knuth : "The Art of Computer Programming".
Addison-Wesley, 1968. (page 353).

74. G. Karakitsos : "From MLT to SAM".
Internal Publication of the School of Computing, North London Polytechnic. (#4).

75. Conference on "Formal Semantics of Programming Languages", September 14th - 16th, 1970.
Proceedings ed. R.Rustin, Prentice-Hall, New Jersey.

76. D. Scott : "Lattice Theory, Data Types and Semantics".
p65 of [75].

77. E. W. Dijkstra : "Go To Statement Considered Harmful".
Communications of the ACM, 11, 3. Page 147. 1968.

78. S. Eisenbach (ed.) : "Functional Programming : Languages, Tools and Architectures".
Ellis Horwood, 1987.

79. J. H. Morris : "Real Programming in Functional Languages".
in [20].

80. J. H. Morris, Jr. : "A Correctness Proof Using Recursively Defined Functions".
in [75].

81. C. Strachey : "Varieties of Programming Languages".
Oxford University Technical Report : PRG-10. Oxford Uni. Computer Lab., Programming Research Group, 1972.

82. D. Scott : "Logic and Programming Languages".
Communications of the ACM, 20,9, 1977.

83. T. B. Steel (ed) : "Formal Language Description Languages".
North-Holland Publishing, Amsterdam, 1966.

84. A. Blikle & A Tarlecki : "Naive Denotational Semantics".
Conference on Foundations of Information Processing, 1983.
Proceedings published by North-Holland publishing, R Mason (ed.).

85. J. M. Cadiou : "Recursive Definitions of Partial Functions and their Computations".
Ph. D. Thesis, Computer Science Dept., Stanford University, California.

86. J. H. Morris : "Lambda Calculus Models of Programming Languages".
Ph. D. Thesis. Project MAC, Technical Report TR-57. MIT, Cambridge, Mass.

87. D. McCracken : "A Guide to Fortran IV Programming".
Wiley, New York, 1965.

88. C. Bohm & G. Jacopini : "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules".
Communications of the ACM, 9, 5, 1966.

89. E. Ashcroft & Z. Manna : "The Translation of 'goto' Programs into 'while' Programs".
Proceedings of the IFIP Congress. North-Holland, Amsterdam, 1971.

90. J. Backus : "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
Communications of the ACM, 21,8. 1978.

91 J. Darlington & R.M. Burstall : "A System Which Automatically Improves Programs". Acta Informatica 6, 41-60. 1976.