STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM - 218

STAN-CS-73-393

AD772063

PROOF TECHNIQUES FOR RECURSIVE PROGRAMS

BY

JEAN E. VUILLEMIN

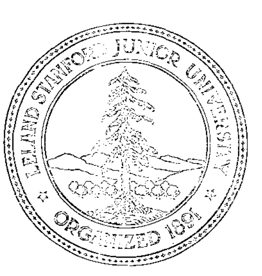
SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 2494
PROJECT CODE 3D30

OCTOBER 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151



Proof Techniques for Recursive Programs

Jean Vuillemin

Abstract

The concept of least fixed-point of a continuous function can be considered as the unifying thread of this dissertation.

The connections between fixed-points and recursive programs are detailed in Chapter 2, providing some insights on practical implementations of recursion. There are two usual characterizations of the least fixed-point of a continuous function. To the first characterization, due to Knaster and Tarski, corresponds a class of proof techniques for programs, as described in Chapter 3. The other characterization of least fixed points, better known as Kleene's first recursion theorem, is discussed in Chapter 4. It has the advantage of being effective and it leads to a wider class of proof techniques.

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

This research was supported by the Advanced Research Projects Agency, Dept. of Defense under contract DAHC 15-73-C-0435

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151.

Acknowledgments

First of all, I am grateful to Dana Scott, Robin Milner, and David Park who, by their respective works, made this thesis possible. I am deeply indebted to:

Donald Knuth for his reading of the manuscript; his criticisms of Chapter 2 led to rewarding improvements in the generality of the results.

Zohar Manna for his constant encouragement and help; he has been a model adviser throughout my work.

Robin Milner for all the things I learned from him, and the many interesting discussions we had.

I also want to thank my friends Jean Marie Cadiou, Ashok Chandra, Cyril Grivet, Gilles Kahn, Lockwood Morris, Steve Ness, Mark Smith, and Phyllis Winkler who all helped me in their many different ways.

Table of Contents

Introduction 1

Chapter 1. Scott's Theory of Computation 3

1. Data-Types 4

2. Computable Functions Over Data-Types 9

3. Fixed-Points 13

Chapter 2. Fixed Points and Recursion 15

1. Computations of Recursively Defined Functions 15

1.1 Description of Lang S and Lang P 16

1.2 Conventions and Notations 18

1.3 Computation Rule 20

1.4 Computation Lattice of a Program 22

2. Correct Implementations of Recursion 33

2.1 Incorrect Computation Rules 33

2.2 Safe Computation Rules 34

3. An Optimal Implementation of Recursion in Lang S 41

3.1 Never Do Today What You Can Put Off Until Tomorrow 41

3.2 Optimality of the Delay Rule 45

3.3 Sequential Functions 46

Chapter 3. Proofs Based Upon Monotonicity 53

1. A Formal System for the Time Being 53

1.1 Syntax 53

1.2 Semantics 54

1.3 Axioms and Rules of Inference 54

1.4 Soundness 56

1.5 Pragmatics 57

1.6 A Possible Weakness of the System 61

2. Justification of Some Proof Techniques 64

2.1 Description of a Flowchart Language 64

2.2 The Inductive Assertions Technique 67

2.3 Termination of Programs 68

Chapter 4. Proofs Based Upon Continuity 70

1. Description of LCF 71

1.1 Syntax 71

1.2 Axioms and Rules 72

1.3 Some Remarks About the Logic 74

1.4 Some Examples of Proofs 76

2. Modelling Some Proof Techniques Within LCF 81

2.1 Structural Induction 81

2.2 Truncation Induction 88

Conclusion 94

References 95

Introduction

The goal of this work was to study and hopefully compare in a precise way the various techniques for proving properties of programs existing in the literature. It soon turned out that nothing interesting could be said if one did not state precisely what the various methods really are within a common logical system. A perfectly adequate system for doing so was the Logic for Computable Function of Milner [18], which is based on the work of Scott [29] and [30].

In this framework, proof techniques fall rather nicely into two classes: for the first class, which includes the methods of Burstall [1], Floyd [7], Hoare [9], Manna-Pnueli [16], the semantics needed for validating the techniques only demand that programs be interpreted as monotone functions in the sense of Scott [29]; for methods in the second class, such as those of Scott [30] and Morris [23], programs must be interpreted as continuous functions.

The methods in the second class are then "more powerful" in that they can be used for justifying the other techniques; furthermore, provided that all methods are expressed within the same logical system, we can exhibit properties of programs which are provable with the proof-techniques in the second class, and not provable with the techniques in the first class, and not vice-versa.

Before studying the various proof techniques, we present a minimal background in Scott's Theory of Computation in Chapter 1. One of the points of the theory which we thought needed clarification was the relations between the abstract notion of least fixed-point and the

concrete notion of trace of a program. Chapter 2, which is the most original part of this thesis, is devoted to this question. We believe that Theorems 1, 3 and 4 are new while Theorem 2 is a generalization of a result by Cadiou [2].

In Chapter 3, we study the proof-technique in the first class. The formal system used is original, although a mere adaptation of Milner's LCF to a different semantic domain. Reduction of the proof techniques presented to the rule of fixed-point induction are due to Park [26].

In Chapter 4, we describe reductions of some methods to the rule of induction of Scott [30]; some of these reductions are also used, implicitly or explicitly in deBakker-Scott [6], Scott [30], Milner [18], and Milner-Weyrauch [21].

Chapter 1. SCOTT'S THEORY OF COMPUTATION

In this chapter, we shall present an overview of Scott's theory of computation, whose goal was to give a "mathematical" as opposed to "operational" semantics for high-level programming languages. Only the parts of the theory which are relevant to this dissertation will be described. In particular, one of Scott's most impressive achievements was to construct a model for the λ-calculus, which in turn provides a mathematical semantics for programming peculiarities such as self-modifying machine codes or procedures taking other procedures as arguments. We shall not concern ourselves with this problem, and the kind of procedure we are willing to consider has a definite type -- a function from individuals to individuals, or a functional from functions to functions, etc. Limited as it is, the theory that we shall describe is nevertheless powerful enough not only to describe the semantics of non-trivial subsets of any programming language, but also to justify all the existing proof techniques for those languages. The presentation of this chapter, whose only purpose is to make the thesis more or less self-contained, is based on Scott [29] except for some minor technical details.

We assume that the reader has some knowledge of elementary lattice and recursion theories.

1. Data Types

As a first step, let us consider some examples of what one would like to call data types:

(a) the boolean values true and false;

(b) the set of integers;

(c) the n-dimensional arrays of integers;

(d) the set of subsets of integers;

(e) the set of computable partial functions over some data-type;

(f) the set of non-negative real numbers.

Some of those sets contain as elements objects like total functions or irrational real numbers, which we shall call "infinite elements". They cannot be described entirely, but one can give better and better finite approximations to what they really are. For example, the intervals $[3,4]$, $[3.1,3.2]$, $[3.14,3.15]$, ... form a sequence of approximations of π.

This suggests that data-types ought to be partially ordered sets. The notation $x \sqsubseteq y$ means that x approximates y, and $\sqsubseteq$ must therefore be a reflexive, transitive and antisymmetric relation over the data-type. For example, if A and B are some subsets of the integers, $A \sqsubseteq B$ means that A is a subset of B. Similarly, for any two intervals $[x,x']$ and $[y,y']$ of non-negative real numbers $[x,x'] \sqsubseteq [y,y']$ will mean that $x \leq y$ and $y' \leq x'$, i.e., $[y,y']$ gives us a better idea of where the real number lies than $[x,x']$.

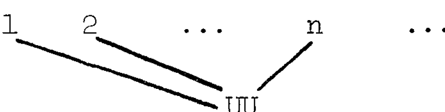
Considering now two integers k and ℓ, we do not wish to say that one is an approximation of the other. However, it may be the case that k is not explicitly known, but has to be determined as

the result of some computation. As we all know, this computation may never terminate, in which case k is said to be <u>undefined</u>; we denote this by $k \equiv UU$ and clearly $UU \sqsubseteq \ell$ for any ℓ. We use a different equality sign " $\equiv$ " in order to avoid confusions with the regular equality " $=$ " over the integers. Here, $x \equiv y$ means that $x \sqsubseteq y$ and $y \sqsubseteq x$, while $x = y$ is true whenever x and y are the same integer. For example, $1 = 1$ and $1 \equiv 1$ are both true, while $UU \equiv 1$ is false and $UU = 1$ is undefined. To be precise, one should write $(UU_I = 1) \equiv UU_B$ where the subscripts are here to remind us that UU_I is an undefined integer, while UU_B is an undefined boolean.

To clarify those ideas, it is helpful to describe more precisely the partial orderings over our favorite data types.

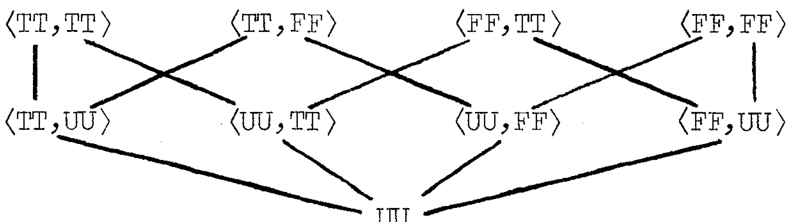
(a) For the boolean values, the data type looks like TT ╲ UU ╱ FF, where b ╲ a means that b covers a, i.e., $a \sqsubseteq b$ with $a \not\equiv b$ and $a \sqsubseteq c \sqsubseteq b$ for some c implies either $a \equiv c$ or $c \equiv b$.

(b) Although there are infinitely many integers, the corresponding data type is not much richer:



Data types of this kind, where elements are either completely specified or undefined will be called <u>discrete</u>.

(c) The data type of pairs of Boolean has already a richer structure:



(d) In the data type of subsets of some set, $A \sqsubseteq B$ means that A is a subset of B ; the least element UU is the empty set.

(e) As indicated before, the elements of the data type of real numbers are closed intervals $[x,x']$ with $0 \leq x \leq x'$ and $[x,x'] \sqsubseteq [y,y']$ whenever $x \leq y$ and $y' \leq x'$. It is convenient to complete the real line with an element ∞ , thus allowing $[7.1 , \infty]$ for example, to be a real number. The interval $[0,\infty]$ reflects a complete lack of information and should therefore be identified with the undefined real UU .

(f) If $\mathscr{D}$ is a data type partially ordered by $\sqsubseteq_{\mathscr{D}}$, the partial functions mapping $\mathscr{D}$ into $\mathscr{D}$ are ordered by:

$$f \sqsubseteq g \quad \text{iff} \quad f(x) \sqsubseteq_{\mathscr{D}} g(x) \quad \text{for all } x \text{ in } \mathscr{D} .$$

The minimal element $UU_{\mathscr{D}\to\mathscr{D}}$ is the partial function which is everywhere undefined, i.e., $UU(x) \equiv UU$ for all x in $\mathscr{D}$.

Infinite Elements as Limits

Let us contemplate again the sequence $[3,4] , [3.1 , 3.2] , [3.14 , 3.15] , \ldots$. We would like to be able to

define π as the "limit" of these intervals. Abstractly, this will require that any chain [*]

$$x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_i \sqsubseteq x_{i+1} \sqsubseteq \cdots$$

has a limit y in the data type $\mathcal{D}$, which is the <u>least-upper bound</u> of the x_i's, that is, $x_j \sqsubseteq y$ for every j and, for any z in the data type, $x_j \sqsubseteq z$ for every j implies $y \sqsubseteq z$. We write $y \equiv \bigsqcup_{i \geq 0} x_i$.

According to this notation, in the data-type of real numbers $[1,2] \equiv \bigsqcup_{i \geq 0} [i/(i+1),(2i+1)/i]$ and for sets of integers, $\{k|k \text{ is odd}\} \equiv \bigsqcup_{i \geq 0} \{1,3,\ldots,2i+1\}$. Let us define the constant function <u>one</u> as <u>one</u>$(x) \equiv 1$ for any integer x, while one(UU) $\equiv$ UU ; this function can also be defined as a limit of partial functions

$$\underline{\text{one}} \equiv \bigsqcup_{i \geq 0} [\lambda x.\ \underline{\text{if}}\ x \leq i\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ \text{UU}] .$$

Computability

Asking that the infinite object $\bigsqcup_{i \geq 0} x_i$ be computable will require that the x_i themselves be computable. We therefore postulate the existence of an effectively given subset E of the data type $\mathcal{D}$, such that any element of $\mathcal{D}$ is the limit (not necessarily effective) of some chain of elements of E. Such a set E will be called a recursive basis of $\mathcal{D}$. For example, a data-type in which there are no infinite ascending chains (booleans, integers, arrays) is its own

---

[*] Strictly speaking, we only need denumerable chains to have a limit. However, when data-types have a denumerable basis (see below), requiring that countable chains have limits implies that any chain (and in fact directed set) also has a limit.

basis provided that it is recursive. The finite sets of integers constitute a basis for the set of subsets of the integers. Similarly, the set of functions which are undefined for all but a finite number of arguments is a basis for the data type of partial functions. Finally, a basis for the real numbers is the set of rational-end-point intervals.

We can remark that the recursive basis of a data type $\mathcal{D}$ must be denumerable. Consequently, all of its elements being obtained as limits of denumerable chains in the basis, $\mathcal{D}$ itself has at most a continuum number of elements. In particular, since there are at most denumerably many computable objects (i.e., objects defined as limits of effectively given chains), a non-denumerable data-type will possess many non-computable elements.

We can summarize the above discussion by the postulate

> A data-type is a partially ordered set with a minimal element, possessing a recursive basis and in which every ascending chain has a limit.

Note: This notion of data-type is slightly different from the one advocated by Scott [29], namely that data-types ought to be complete lattices. The main technical reason for this choice was the difficulty which seems to arise for defining our notion of sequential function in Chapter 2, with complete lattices.

2. Computable Functions over Data Types

The next step is to consider programs as functions mapping data types into data types, and to derive some mathematical properties of such functions.

Programs as Monotone Mappings

Let f be a partial function computed by some program. Whenever the input x is less defined than the input y, the output $f(x)$ must be less defined than $f(y)$, i.e., $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. This motivates the hypothesis that functions computed by programs are monotonic mappings over the data type.

Examples

— The successor function $[\lambda x.\ x+1]$ over the integers is monotone if we choose $UU+1 \equiv UU$.

— The conditional if p then x else y where

if UU then x else y $\equiv$ UU

if TT then x else y $\equiv$ x

if FF then x else y $\equiv$ y

is monotone with respect to p, x and y. (A function of several variables is monotone when it is monotone in each of its arguments.)

— As for sets, the functions $A \cup B$ and $A \cap B$ are both monotone in A and B.

— The following definition of division over the reals makes it a monotone function:

$$[x,y]/[x',y'] \equiv [\frac{x}{y'}, \frac{y}{x'}] \quad \text{where}$$

$$\frac{x}{\infty} = 0 \quad \text{and} \quad \frac{x}{0} = \infty \quad \text{for all} \quad x \in [0,\infty] .$$

Programs as Continuous Mappings

As it stands now, the theory is already quite adequate for expressing and proving properties of programs, and Chapter 3 describes some results which can be derived from the assumption that mappings between data-types are monotone functions.

However, we are still missing an essential property of computable functions. Knowing the values of a monotone function over the basis of a data-type does not determine in general its values over the data-type. For example, the function

$$\text{funny-union}(A,B) \equiv \begin{cases} A \cup B & \text{if } A \text{ or } B \text{ is finite} \\ N & \text{if } A \text{ and } B \text{ are infinite} \end{cases}$$

where A and B are two subsets of N, is monotone but clearly not computable.

Intuitively, the value $f(x)$ of a computable function f at an infinite object x should be obtained as the limit of the values $f(x_i)$ over the finite approximation x_i of x. More precisely, let us consider an arbitrary chain

$$e_0 \sqsubseteq e_1 \sqsubseteq \cdots \sqsubseteq e_n \sqsubseteq e_{n+1} \sqsubseteq \cdots$$

of elements in the basis of the data type. Since f is monotone, the set $\{i \geq 0 \mid f(e_i)\}$ is also a chain

$$f(e_0) \sqsubseteq f(e_1) \sqsubseteq \cdots \sqsubseteq f(e_n) \sqsubseteq f(e_{n+1}) \sqsubseteq \cdots$$

and the computability of f demands that

$$\underline{f(\bigsqcup_{n\geq 0} e_n) \equiv \bigsqcup_{n\geq 0} f(e_n)} \qquad \text{(a)}$$

A monotone function satisfying equation (a) for arbitrary chains will be called continuous. We shall therefore postulate that

<u>Computable functions are continuous mappings between data-types</u>.

Again, a function of several arguments is continuous if it is continuous in each of its arguments.

<u>Examples</u>

— The function $[\lambda p,x,y.$ <u>if</u> p <u>then</u> x <u>else</u> $y]$ is continuous. Addition of two integers, union of two sets, division of reals are also continuous operations. The functional $[\lambda F.[\lambda x.$ <u>if</u> $x = 0$ <u>then</u> 1 <u>else</u> $x.F(x-1)$ over the data-type of natural numbers is continuous, both in F and in x.

— Let us define the mappings $\exists x\ p(x)$ and $\forall x\ p(x)$ which associate a boolean to each function p from natural numbers to booleans as follows:

— $\exists x\ p(x)$ is equal to TT if $p(n) \equiv$ TT for some natural number n and equal to UU otherwise.

— $\forall x\ p(x)$ is equal to TT if $p(n) \equiv$ TT for all natural numbers $n \not\equiv$ UU and equal to UU otherwise.

We shall verify that $[\lambda p.(\exists x)p(x)]$ is continuous while $[\lambda p.(\forall x)p(x)]$ is monotone but not continuous in general. Let $p_0 \sqsubseteq \cdots \sqsubseteq p_i \sqsubseteq p_{i+1} \sqsubseteq \cdots$

be a chain of partial predicates over the natural numbers. We easily verify that $(\bigsqcup_{i \geq 0} p_i)(x) \equiv \bigsqcup_{i \geq 0} (p_i(x))$. Now, if $(\bigsqcup_{i \geq 0} p_i)(x) \equiv \bigsqcup_{i \geq 0} p_i(x) \equiv TT$ for some x , there must exist an i_0 such that $i \geq i_0$ implies $p_i(x) \equiv TT$; otherwise, either $(\bigsqcup_{i \geq 0} p_i)(x) \equiv FF$ and again there is an i_0 such that $p_{i_0}(x) \equiv FF$ or $(\bigsqcup_{i \geq 0} p_i)(x) \equiv UU$ and $p_i(x) \equiv UU$ for all i . In all cases we have $(\exists x)(\bigsqcup_{i \geq 0} p_i)(x) \equiv \bigsqcup_{i \geq 0} (\exists x) p_i(x)$ and $\exists$ is indeed continuous. One shows that $\forall$ is monotone in a similar way and the chain $p_i(x) \equiv (x < i)$ provides a counterexample to the continuity of $\forall$.

Let us now discuss some properties of continuous functions. First of all, it is possible to define a topology over data-types such that a function is continuous in the above sense if and only if it is continuous in the topological sense (see Scott [31]). Without describing the topology, we can nevertheless say that a subset X of the data-type $\mathcal{D}$ is directed if for all $x,y \in X$, there exists a $z \in X$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$. Together with the existence of a denumerable basis for $\mathcal{D}$, the fact that continuous functions preserve limits of denumerable chains implies that continuous functions also preserve least-upper-bounds of directed sets. Continuous functions do not however preserve least-upper-bounds or greatest-lower-bounds (when they exist) of arbitrary sets.

3. Fixed Points

Let f be a function over a data-type $\mathcal{D}$. We say that $x \in \mathcal{D}$ is a fixed-point of f if $x \equiv f(x)$; we say that y is the <u>least-fixed-point</u> of f if $y \equiv f(y)$ and $y \sqsubseteq x$ for any other fixed-point x. Note that, whenever it exists, the least-fixed-point of f must be unique; we shall denote it either by $\mu x.f(x)$ or by x_f.

<u>Theorem</u> (Kleene). <u>Any continuous function over a data-type $\mathcal{D}$ has a least-fixed-point x_f and</u>

$$\underline{x_f \equiv \bigsqcup_{n \geq 0} f^n(\mathrm{UU})} \quad .$$

<u>Proof</u>. Here $f^n(\mathrm{UU})$ means $f(f(\ldots(f(\mathrm{UU}))\ldots))$ (n times) and, by monotonicity of f, the set $\{f^n(\mathrm{UU})\}$ for $n \geq 0$ is indeed a chain. We first prove that $\bigsqcup_{n \geq 0} f^n(\mathrm{UU})$ is a fixed point of f. This is easy since

$f(\bigsqcup_{n \geq 0} f^n(\mathrm{UU})) \equiv \bigsqcup_{n \geq 0} f^{n+1}(\mathrm{UU}) \equiv \bigsqcup_{n \geq 0} f^n(\mathrm{UU})$ by continuity of f.

We now prove that $\bigsqcup_{n \geq 0} f^n(\mathrm{UU})$ must be minimal. Let y be an arbitrary fixed-point of f, i.e., $y \equiv f(y)$. It is easy to prove by induction that $f^n(\mathrm{UU}) \sqsubseteq y$ for any n. The conclusion $\bigsqcup_{n \geq 0} f^n(\mathrm{UU}) \sqsubseteq y$ follows immediately.

□

<u>Examples</u>

— In any data type, $\mathrm{UU} \equiv [\mu y.y]$ and $x \equiv [\mu y.x]$.

If $\tau \equiv \lambda f.[\lambda x.$ <u>if</u> $x = 0$ <u>then</u> 1 <u>else</u> $x.f(x-1)]$

and $\sigma \equiv \lambda f.[\lambda x.$ <u>if</u> $x > 100$ <u>then</u> $x-10$ <u>else</u> $f(f(x+11))]$ over the natural numbers,

then $\tau^{n+1}(UU) \equiv [\lambda x.\ \underline{if}\ x \leq n\ \underline{then}\ x!\ \underline{else}\ UU]$

and $\sigma^{n+1}(UU) \equiv [\lambda x.\ \underline{if}\ x > 100\ \underline{then}\ x-10$

$\underline{else}\ \underline{if}\ x-100 > -n\ \underline{then}\ 91\ \underline{else}\ UU]$;

therefore, $f_\tau \equiv [\lambda x.x!]$ and $f_\sigma \equiv [\lambda x.\ \underline{if}\ x > 100\ \underline{then}\ x-10\ \underline{else}\ 91]$.

From these examples, the reader may already suspect that there must be a relation between recursively defined functions and least fixed points. The next chapter will be entirely devoted to this question.

Chapter 2. FIXED-POINTS AND RECURSION

The object of this chapter is to detail the connections between fixed-points of continuous functionals and recursively defined functions in a very simple programming language. We first illustrate that the semantics of recursively defined functions will depend on the implementation. A careless implementation of recursion will introduce unnecessary computations, which may even prevent the program from terminating. A general criterion for the correctness of an implementation will be proved. We then describe an implementation of recursion which is both correct and optimal in a general class of sequential languages and therefore constitutes an attractive alternative to both "call by value" and "call by name".

1. Computations of Recursively Defined Functions

Before defining a computation rule, we must describe two programming languages, *lang* S and *lang* P . Although those two languages were chosen for their extreme simplicity, their use of recursion is as general as any, and the results of this chapter provide some insight into semantics and implementation of more complex programming languages.

Lang S permits only *sequential* computations, and corresponds precisely to a certain "typed" subset of Algol or LISP.

Lang P requires some parallel operations, and thus departs from more classical programming languages, although we could undoubtedly write an interpreter for lang P in any of those classical languages.

1.1 Description of lang S and lang P

Syntax

Both languages have the same syntax:

$$\langle\text{program}\rangle ::= F(X_1,\ldots,X_n) \Leftarrow \langle\text{term}\rangle$$

$$\begin{aligned}\langle\text{term}\rangle ::= {} & A_1 | A_2 | \ldots \\ & | X_1 | \ldots | X_n \\ & | G_1(\langle\text{term } 1\rangle,\ldots,\langle\text{term } p_1\rangle) \\ & \vdots \\ & | G_k(\langle\text{term } 1\rangle,\ldots,\langle\text{term } p_k\rangle) \\ & | F(\langle\text{term } 1\rangle,\ldots,\langle\text{term } n\rangle) \quad .\end{aligned}$$

We limited ourselves to a single recursive equation, the extension of the results in this chapter to systems of mutually recursive equations being straightforward.

Here, $A_1,A_2,\ldots,G_1,\ldots,G_k$ denote fixed constants and functions respectively. It is convenient to use a more standard syntax, e.g., $F(X) \Leftarrow \text{IF } X = 0 \text{ THEN } 1 \text{ ELSE } X.F(X-1)$ instead of $F(X) \Leftarrow G_1(P_1(X,A_0),A_1,G_2(X,F(G_3(X))))$.

The meaning of a program will be a continuous mapping in $[\mathcal{D}_1 x \ldots x \mathcal{D}_n \to \mathcal{D}]$ where each $\mathcal{D}_i$ and $\mathcal{D}$ are some data-types; for simplicity, the $\mathcal{D}_i$'s will be identical to $\mathcal{D}$ unless explicitly specified.

Semantics of terms in lang P

The meaning of a $\langle\text{term}\rangle$ is a (continuous) functional $\lambda f.\lambda x_1,\ldots,x_n \mathcal{J}(\langle\text{term}\rangle)$ where the semantic function $\mathcal{J}$ is defined inductively as follows:

(i) $\mathcal{J}(A_1) \equiv a_i$ where $a_i \in \mathcal{D}$

(ii) $\mathscr{A}(X) \equiv x_i$

(iii) $\mathscr{A}(G_k(\langle \text{term } 1\rangle, \ldots, \langle \text{term } p_k\rangle) \equiv g_k(\mathscr{A}(\langle \text{term } 1\rangle), \ldots, \mathscr{A}(\langle \text{term } p_k\rangle))$

where g_k is some continuous function in $[\mathcal{D}^{p_k} \to \mathcal{D}]$.

(iv) $\mathscr{A}(F(\langle \text{term } 1\rangle, \ldots, \langle \text{term } n\rangle)) \equiv f(\mathscr{A}(\langle \text{term } 1\rangle), \ldots, \mathscr{A}(\langle \text{term } n\rangle))$.

Here we have to prove that this is continuous, i.e., that continuous functions are closed under composition, λ-abstraction and fixed-point operation. The reader can find these proofs either in Scott [30] or in Milner [19].

Semantics of Terms in lang S

The semantics of lang S is defined in precisely the same way as that of lang P , the difference lying in restrictions on the interpretation of base functions. In lang S , we require functions to be sequential, i.e., roughly that their arguments can be computed in sequence. We shall give later a precise definition of this notion. For expository purposes, however, we shall limit ourselves for the moment to studying a particular sequential language.

The data-types on which our particular lang S is computing are discrete, i.e., they look like:

In what follows, we use ω instead of $uu_{\mathcal{D}}$ and Ω in place of $uu_{\mathcal{D}\to\mathcal{D}}$ in order to help the eye avoid type confusions. Among the base functions, we point out a particular one, denoted IF-THEN-ELSE whose interpretation is the usual conditional, i.e., <u>if</u> uu <u>then</u> x <u>else</u> y $\equiv \omega$, <u>if</u> tt <u>then</u> x <u>else</u> y $\equiv$ x and <u>if</u> ff <u>then</u> x <u>else</u> y $\equiv$ y .

All other base functions are required to be strict, i.e., $g_i(\ldots,\omega,\ldots) \equiv \omega$: they are undefined as soon as at least one of their arguments becomes undefined. They are meant to correspond to the "hardware" functions: add , addone , test-for-equality , ...

It will be shown that all functions definable in lang S are sequential. The symmetric OR defined by the table:

x OR y

x \ y	uu	tt	ff
uu	uu	tt	uu
tt	tt	tt	tt
ff	uu	tt	ff

or the symmetric multiply * where $0*x \equiv x*0 \equiv 0$ are not sequential, and are therefore not definable in lang S , nor in Algol for that matter.

Semantics of Programs in both lang S and lang P

The functional $\tau \equiv \lambda f.\lambda x_1,\ldots,x_n \mathscr{S}(\langle\text{term}\rangle)$ as defined in lang S or lang P can be shown to be continuous. It must therefore have a least fixed-point f_τ and it would be nice to define the meaning $\mathcal{M}$ of the corresponding program as $\mathcal{M}(\langle\text{program}\rangle) \equiv f_\tau$.

This is unfortunately not true for all implementations of recursion, and our goal will be to characterize the implementations for which the computed function is equal to this least fixed-point.

1.2 Conventions and Notations

The reader has already noticed that syntactic entities are denoted by upper case letters, while the associated semantic objects are represented by the corresponding lower-case letters. We shall keep this convention throughout this chapter. For example, if T is the term

IF X = 0 THEN 1 ELSE X.F(X-1) , then its meaning t is

$\lambda f.\lambda x$ <u>if</u> $x = 0$ <u>then</u> 1 <u>else</u> $x.f(x-1)$, where $=$ in this last expression means the equality function over the natural numbers, 0 the number 0 , etc.

From now on, we use upper case letters other than A , D , X , F and G to denote (syntactic) terms. If T and S are terms, we denote by $T\{S/X_i\}$ the result of replacing all occurrences of the letter X_i by the term S in T . By $T\{P/F\}$, we mean the term obtained by replacing in T all subterms of the form $F(T_1,\ldots,T_n)$ by $P\{T_1/X_1,\ldots,T_n/X_n\}$. For example,

$$\text{if} \quad T = G_1(F(X_1,F(X_1,X_2)),X_1) \quad \text{and} \quad P = G(F(X_2,X_1))$$

$$\text{then} \quad T\{P/F\} = G_1(G(F(G(F(X_2,X_1)),X_1)),X_1) \quad .$$

Whenever we only wish to substitute P for some occurrences of F in T , we rename, say F_1 , the occurrences that we shall substitute and F_2 the others. The result of the substitutions is then $T\{P/F_1,F/F_2\}$. The same kind of notation also applies to semantic terms.

We use $F(\bar{X})$ and $f(\bar{x})$ as abbreviations for $F(X_1,\ldots,X_n)$ and $f(x_1,\ldots,x_n)$ respectively.

Also, it will be convenient to consider only programs $F(\bar{X}) \Leftarrow P$ where P is of the form $G(P_1,\ldots,P_p)$ with the additional restriction that each of the letters F , $X_1,\ldots,X_n$ occurs at least once in P . That is, P is required not to ignore any of its program variables, to depend upon F (i.e., to be recursive) and not to be of the uninteresting form $F(X) \Leftarrow F(T_1,\ldots,T_n)$. The main results of this chapter generalize without this restriction, but the proofs are made longer by an addition of special cases.

1.3 Computation Rule

A computation rule $\mathcal{C}$ is an algorithm for selecting some occurrences of the letter F in each term. For any such rule and input $\bar{D}$, we construct the computation sequence $T_0, T_1, \ldots, T_n, \ldots$ of the term T by the program $F(\bar{X}) \Leftarrow P$ as follows: $T_0 = T\{\bar{D}/\bar{X}\}$ and T_{i+1} is the result of substituting P for the F's chosen by $\mathcal{C}$ in T_i. For example, if P = IF X < 2 THEN X ELSE F(X-1) + F(X-2), the computation sequence of F(X) according to "call-by-value" for input X = 2 is:

$$T_0 = \underline{F}(2)$$

$$T_1 = \text{IF } 2 < 2 \text{ THEN } 2 \text{ ELSE } \underline{F}(1) + F(0)$$

$$T_2 = \text{IF } 2 < 2 \text{ THEN } 2 \text{ ELSE } (\text{IF } 1 < 2 \text{ THEN } 1 \text{ ELSE } F(0) + F(-1)) + \underline{F}(0)$$

$$\begin{aligned} T_3 = \text{IF } 2 < 2 \text{ THEN } 2 \\ &\text{ELSE } (\text{IF } 1 < 2 \text{ THEN } 1 \text{ ELSE } F(0) + F(-1)) + \\ &\quad \text{IF } 0 < 2 \text{ THEN } 0 \text{ ELSE } F(-1) + F(-2) \quad . \end{aligned}$$

$$T_4 = T_5 = \cdots = T_3 \quad .$$

(Here, F(1) is in fact an abbreviation for F(2-1), etc.)

In T_n, we underline the F's selected by the computation rule for substitution. It is interesting to see precisely how the underlined F is selected in this last example. For this purpose, we must introduce the notion of simplification. The simplification mechanism is discussed at length in Cadiou [2], and we refer the interested reader to this work. In our particular example, it is possible to define a simplification mechanism $\lambda T\ \text{simpl}(T)$ such that

$$\text{simpl}(T_0) = F(2)$$

$$\text{simpl}(T_1) = F(1) + F(0)$$

$$\text{simpl}(T_2) = 1 + F(0)$$

$$\text{simpl}(T_3) = \text{simpl}(T_4) = \dots = 1$$

(Note that now, $F(1)$ is no longer an abbreviation since $\text{simpl}(2-1) = 1$.)

The rule "call-by-value" then selects the leftmost-innermost occurrence of F in simplified terms. Similarly, "call-by-name" selects the "leftmost-outermost" one.

In its most general form, simplification can be an extremely powerful computation tool. For example, if our program is $F(X) \Leftarrow \text{IF } X = 0 \text{ THEN } 0 \text{ ELSE } F(X-1)$ it is perfectly all right to use $F(X) \to 0$ as a simplification rule over the natural numbers, and there is no room left for substitutions! Our purpose however is to study computations which are performed by substitutions and <u>not</u> by simplifications.

We must therefore restrict the power of simplifications which we allow, and, for this purpose, we merely borrow Cadiou's notion of <u>standard simplifications</u> (see Cadiou [2] for a precise definition). Roughly, standard simplifications force us to know everything about base functions, and nothing a priori about the recursively defined function F , since simplifications of the type $F(\bar{D}) \to A_i$ are not permitted. In effect, we have to compute without any "built in" value of the recursively defined function, stored for example in memory from a previous computation.

We will not study standard simplifications in <u>lang P</u> , since this would require describing completely the data-type on which computations

are performed but we will describe them in <u>lang S</u> .

For all constants $A_{i1},\ldots,A_{ip}$ and base function G_p there exists a standard simplification of the type

$$G_p(A_{i1},\ldots,A_{ip}) \to A_j \quad .$$

In effect, this says that the values of the base-functions over the domain are known, and these functions are total. Accordingly, the conditional admits the simplifications

IF TRUE THEN B ELSE C $\to$ B and

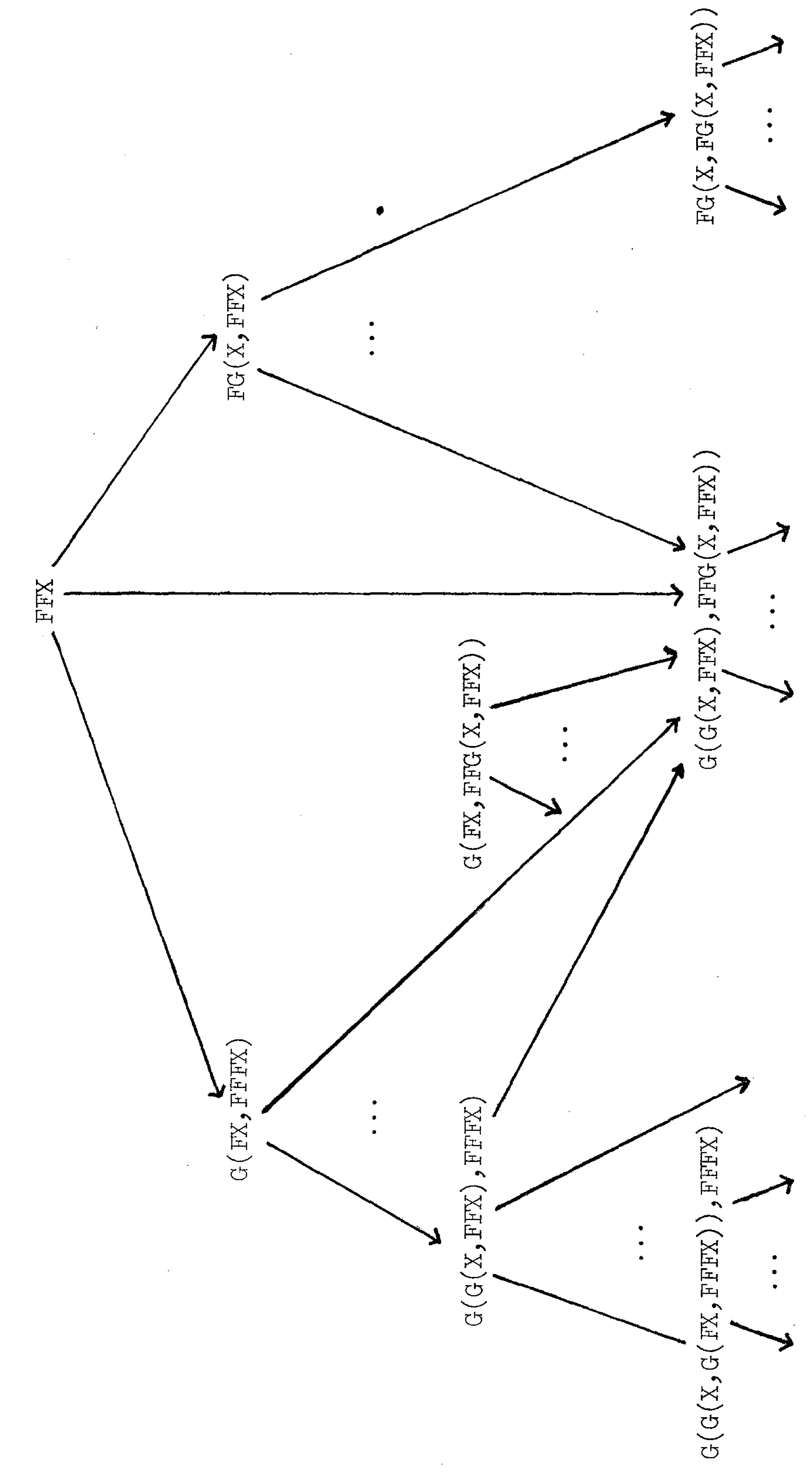
IF FALSE THEN B ELSE C $\to$ C .

These are the only standard simplifications in lang S and we say that a term is simplified when all of its subterms have been simplified.

1.4 Computation Lattice of a Program

Instead of considering computation sequences for each input and computation rule, we can apprehend the set of all possible computations in one infinite diagram.

For example, the computation diagram of the term F(F(X)) by the program F(X) <= G(X,F(F(X))) looks like



FFX
FG(X,FFX)
G(FX,FFFX)
FG(X,FG(X,FFX))
G(G(X,FFX),FFG(X,FFX))
G(FX,FFG(X,FFX))
G(G(X,FFX),FFFX)
G(G(X,G(FX,FFFX)),FFFX)

A computation rule is then an algorithm for selecting a path in such a graph for each input. This computation diagram has a very rich structure which we shall now study.

<u>Computation of a term according to P</u>

We say that $B \underset{P}{\rightarrow} C$ or simply $B \rightarrow C$ whenever C can be obtained by substituting P for some occurrences of F in B.

The notation $B \overset{*}{\underset{P}{\rightarrow}} C$ or $B \overset{*}{\rightarrow} C$ means that there exists a finite sequence of terms $D_0, D_1, \ldots, D_m$ such that $D_0 = B$, $D_m = C$ and $D_i \underset{P}{\rightarrow} D_{i+1}$ for $0 \leq i < m$.

<u>Definition</u>

<u>The computation diagram of T by P is the set of terms U such that $T \overset{*}{\underset{P}{\rightarrow}} U$, partially ordered by $\leq$ where $B \leq C$ whenever $B \overset{*}{\underset{P}{\rightarrow}} C$.</u>

It is clear that $\leq$ is reflexive and transitive. In order to prove that it is also antisymmetric, we notice that, if $B \underset{P}{\rightarrow} C$, the size $\|C\|$ (where size is, say the number of symbols) of the term C is strictly larger than the size of B if at least one substitution has been performed (this is due to our restriction on P). It follows that $B \overset{*}{\rightarrow} C$ and $C \overset{*}{\rightarrow} B$ implies $B = C$.

Clearly, the computation diagram of T by P has the Church-Rosser property of the λ-calculus. (This follows from the work of Rosen [28] for example.) However, it also has a property which is <u>not</u> true of the λ-calculus, namely:

Theorem 1

The computation diagram of T by P is a lattice under the ordering $\leq$, and we shall name it the computation lattice of T by P .

Proof.[*] In order to study the structure of the computation diagram of a term T_0 by a program P , we need to relate the structure of C to that of B when $B \overset{*}{\underset{P}{\rightarrow}} C$.

Lemma 1

(i) $A_i \overset{*}{\rightarrow} C$ if and only if $C = A_i$ and $X_j \overset{*}{\rightarrow} C$ if and only if $C = X_j$.

(ii) $G_i(B_1,\ldots,B_{p_i}) \overset{*}{\rightarrow} C$ if and only if $C = G_i(C_1,\ldots,C_{p_i})$ and $B_i \overset{*}{\rightarrow} C_i$ for $1 \leq i \leq p_i$.

(iii) $F(B_1,\ldots,B_n) \overset{*}{\rightarrow} C$ if and only if $C = F(C_1,\ldots,C_n)$ with $B_i \overset{*}{\rightarrow} C_i$ for $1 \leq i \leq n$ or $P\{B_1/X_1,\ldots,B_n/X_n\} \overset{*}{\rightarrow} C$.

Proof. Claims (i) and (ii) are easy and we only prove (iii).

If $B = F(B_1,\ldots,B_n) \overset{*}{\rightarrow} C$ and C is not of the form $F(C_1,\ldots,C_n)$, there must be a point in the computation $B \overset{*}{\rightarrow} C$ where the outermost F of B is substituted, i.e., $F(B_1,\ldots,B_n) \overset{*}{\rightarrow} F(B'_1,\ldots,B'_n) \overset{*}{\rightarrow} P\{B''_1/X_1,\ldots,B''_n/X_n\} \overset{*}{\rightarrow} C$ with $B'_i \rightarrow B''_i$ (and therefore $B_i \overset{*}{\rightarrow} B''_i$) for any $1 \leq i \leq n$.

It follows from our definitions that $B_i \overset{*}{\rightarrow} B''_i$ for $1 \leq i \leq n$ implies $P\{B_1/X_1,\ldots,B_n/X_n\} \overset{*}{\rightarrow} P\{B''_1/X_1,\ldots,B''_n/X_n\}$ and consequently $P\{B_1/X_1,\ldots,B_n/X_n\} \overset{*}{\rightarrow} C$, as claimed in (iii). In order to get the

---

[*] I am grateful to Jean-Marie Cadiou for his help with this proof.

other part of the implication (iii), we simply notice that $F(B_1,\ldots,B_n) \to P\{B_1/X_1, \ldots, B_n/X_n\}$ by substituting P for the outer F in $F(B_1,\ldots,B_n)$.

□

If $B \leq C$, we can define a distance $\text{dist}(B,C)$ between B and C as follows:

(i) if $B = A_i$ or $B = X_j$ then $C = B$ and $\text{dist}(B,C) = 0$;

(ii) if $B = G_i(B_1,\ldots,B_{p_i})$ then $C = G_i(C_1,\ldots,C_{p_i})$ with $B_i \leq C_i$ for $1 \leq i \leq p_i$ and $\text{dist}(B,C) = \max_{1 \leq j \leq p_i} \{\text{dist}(B_j,C_j)\}$;

(iii) if $B = F(B_1,\ldots,B_n)$ then (by Lemma 1), either $C = F(C_1,\ldots,C_n)$ and $\text{dist}(B,C) = \max_{1 \leq i \leq n} \{\text{dist}(B_i,C_i)\}$ or

$P\{B_1/X_1, \ldots, B_n/X_n\} \leq C$ and $\text{dist}(B,C) = 1 + \text{dist}(P\{B_1/X_1,\ldots,B_n/X_n\},C)$

It is easily seen that the distance between any two terms $B \leq C$ is finite.

Lemma 2

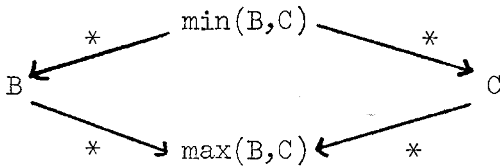
_If $B = F(B_1,\ldots,B_n)$, $C = F(C_1,\ldots,C_n)$, $B' = P\{B_1/X_1,\ldots,B_n/X_n\}$ and $C' = P\{C_1/X_1, \ldots, C_n/X_n\}$ then $B \leq C$ implies $B' \leq C'$ and $\text{dist}(B',C') \leq \text{dist}(B,C)$._

Proof. By a straightforward induction on $\|P\|$, one proves that $\text{dist}(P\{B_1/X_1, \ldots, B_n/X_n\},P\{C_1/X_1, \ldots, C_n/X_n\}) \leq \max_{1 \leq i \leq n} \{\text{dist}(B_i,C_i)\}$, hence $\text{dist}(B',C') \leq \text{dist}(B,C)$.

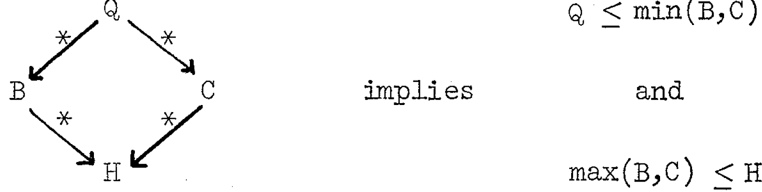
□

We now start the proof of Theorem 1:

For any two terms B, C in the computation diagram of T by P, we must show the existence of $\min(B,C)$ and $\max(B,C)$ such that



and for any Q and H



.

Existence of $\max(B,C)$

We shall describe an algorithm for computing $\max(B,C)$ and then prove the correctness of this algorithm: let $\sigma(B,C)$ be defined recursively as

(i) $\sigma(B,B) = B$,

(ii) $\sigma(G_i(B_1,\ldots,B_{p_i}),G_i(C_1,\ldots,C_{p_i})) = G_i(\sigma(B_1,C_1),\ldots,\sigma(B_{p_i},C_{p_i}))$,

(iii) $\sigma(F(B_1,\ldots,B_n),F(C_1,\ldots,C_n)) = F(\sigma(B_1,C_1),\ldots,\sigma(B_n,C_n))$,

(iv) $\sigma(F(B_1,\ldots,B_n),G(C_1,\ldots,C_p)) = \sigma(P\{B_1/X_1, \ldots, B_n/X_n\},G(C_1,\ldots,C_p)) =$
$\sigma(G(C_1,\ldots,C_p),F(B_1,\ldots,B_n))$,

(v) in all the other cases, $\sigma(B,C)$ yields an error symbol, (say a German gothic letter) which is not part of our set of letters.

We shall prove that $\sigma(B,C) = \max(B,C)$ in two parts:

Part 1. For any terms T , B , C

The proof is by induction on couples $\langle \mathrm{dist}(T,B) + \mathrm{dist}(T,C), \|T\| \rangle$ ordered lexicographically by $<$. Assuming the result to be true for all triples T' , B' , C' with $\langle \mathrm{dist}(T',B') + \mathrm{dist}(T',C'), \|T'\| \rangle <$ $\langle \mathrm{dist}(T,B) + \mathrm{dist}(T,C), \|T\| \rangle$, we prove it for T , B , C by a case analysis on the structure of T .

Case 1. $T = A_i$ or $T = X_j$.

By Lemma 1, $T \xrightarrow{*} B$ and $T \xrightarrow{*} C$ implies $T = B$ and $T = C$; hence $B = C = \sigma(B,C)$ and indeed $B \xrightarrow{*} \sigma(B,C)$ and $C \xrightarrow{*} \sigma(B,C)$.

Case 2. $T = G_i(T_1, \ldots, T_{p_i})$.

By Lemma 1, $B = G_i(B_1, \ldots, B_{p_i})$ and $C = G_i(C_1, \ldots, C_{p_i})$, with $T_i \xrightarrow{*} B_i$ and $T_i \xrightarrow{*} C_i$ for $1 \leq i \leq p_i$. Since $\mathrm{dist}(T_i,B_i) + \mathrm{dist}(T_i,C_i) \leq$ $\mathrm{dist}(T,B) + \mathrm{dist}(T,C)$ and $\|T_i\| < \|T\|$ for any $1 \leq i \leq p_i$, the induction hypothesis tells us that $B_i \xrightarrow{*} \sigma(B_i,C_i)$ and $C_i \xrightarrow{*} \sigma(B_i,C_i)$ for each $1 \leq i \leq p_i$. Regrouping everything, the conclusion $B \xrightarrow{*} \sigma(B,C)$ and $C \xrightarrow{*} \sigma(B,C)$ then follows from the definition $\sigma(G_i(B_1, \ldots, B_{p_i}), G_i(C_1, \ldots, C_{p_i})) = G_i(\sigma(B_1,C_1), \ldots, \sigma(B_{p_i},C_{p_i}))$.

Case 3. $T = F(T_1, \ldots, T_n)$.

By symmetry, we only need consider the subcases:

Case 3.1. $B = F(B_1,\ldots,B_n)$ and $C = F(C_1,\ldots,C_n)$

The proof is similar to that of Case 2.

Case 3.2. $B = F(B_1,\ldots,B_n)$ and $C = G(C_1,\ldots,C_p)$.

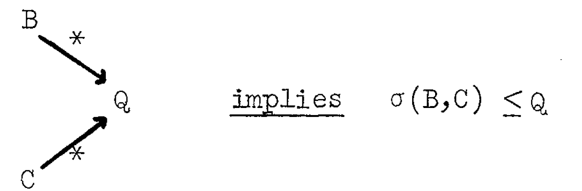
Let $T' = P\{T_1/X_1,\ldots,T_n/X_n\}$ and $B' = P\{B_1/X_1,\ldots,B_n/X_n\}$.

By Lemma 1, we know that $T' \overset{*}{\rightarrow} C$ and $T_i \overset{*}{\rightarrow} B_i$ for $1 \leq i \leq n$, hence $T' \overset{*}{\rightarrow} B'$. By Lemma 2, we know that $\text{dist}(T',B') \leq \text{dist}(T,B)$. Since $\text{dist}(T',C) < \text{dist}(T,C)$, we can apply the induction hypothesis to the terms T' , B' , C , i.e., $B' \overset{*}{\rightarrow} \sigma(B',C)$ and $C \overset{*}{\rightarrow} \sigma(B',C)$. Since $B \rightarrow B'$ and $\sigma(B,C) = \sigma(B',C)$ by definition of σ , we have established that $B \overset{*}{\rightarrow} \sigma(B,C)$ and $C \overset{*}{\rightarrow} \sigma(B,C)$.

Case 3.3. $B = G(B_1,\ldots,B_p)$ and $C = G(C_1,\ldots,C_p)$.

Let $T' = P\{T_1/X_1,\ldots,T_n/X_n\}$. By Lemma 1, we know that $T' \overset{*}{\rightarrow} B$ and $T' \overset{*}{\rightarrow} C$. Since $\text{dist}(T',C) < \text{dist}(T,B)$ and $\text{dist}(T',C) < \text{dist}(T,C)$, we can use the induction hypothesis in order to get $B \overset{*}{\rightarrow} \sigma(B,C)$ and $C \overset{*}{\rightarrow} \sigma(B,C)$.

Part 2. For any terms B , C , Q



The proof is by induction on $\langle \text{dist}(B,Q) + \text{dist}(C,Q), \|Q\| \rangle$.

Case 1. $Q = A_i$ or $Q = X_j$.

Then $Q = B = C = \sigma(B,C)$ and $\sigma(B,C) \overset{*}{\rightarrow} Q$.

Case 2. $Q = F(Q_1, \ldots, Q_n)$ or $Q = G_i(Q_1, \ldots, Q_{p_i})$ where G_i is not G .

The proof goes mutatis-mutandis as that of Part 1, Case 2.

Case 3. $Q = G(Q_1, \ldots, Q_p)$

We only need consider the cases:

Case 3.1. $B = G(B_1, \ldots, B_p)$ and $C = G(C_1, \ldots, C_p)$.

Back to Case 2.

Case 3.2. $B = F(B_1, \ldots, B_n)$ and $C = G(C_1, \ldots, C_p)$.

Let $B' = P\{B_1/X_1, \ldots, B_n/X_n\}$. Since $dist(B',C) < dist(B,Q)$, we know by the induction hypothesis that $\sigma(B',Q) = \sigma(B,C) \overset{*}{\rightarrow} Q$.

Case 3.3. $B = F(B_1, \ldots, B_n)$ and $C = F(C_1, \ldots, C_n)$.

Let $B' = P\{B_1/X_1, \ldots, B_n/X_n\}$ and $C' = P\{C_1/X_1, \ldots, C_n/X_n\}$. The induction hypothesis tells us that $\sigma(B',C') \overset{*}{\rightarrow} Q$. One then proves by induction on $||P||$ that $\sigma(B',C') =$
$\sigma(P\{B_1/X_1, \ldots, B_n/X_n\}, P\{C_1/X_1, \ldots, C_n/X_n)\} = P\{\sigma(B_1,C_1)/X_1, \ldots, \sigma(B_n,C_n)/X_n\}$.

We conclude the proof by noticing that $\sigma(B,C) \rightarrow \sigma(B',C')$ since $\sigma(B,C) = F(\sigma(B_1,C_1), \ldots, \sigma(B_n,C_n)) \rightarrow P\{\sigma(B_1,C_1)/X_1, \ldots, \sigma(B_n,C_n)/X_n\} =$
$\sigma(B',C')$.

Existence of $\min(B,C)$

For any terms B , C in the computation diagram of T by P the set $\{L \mid L \leq B, L \leq C\}$ of lower bounds of B and C is not empty because $T \leq B$ and $T \leq C$ and it is finite. We know from elementary lattice theory that, if any two elements in a partially ordered set have a least-upper-bound, any non-empty finite subset also has a least-upper-

bound. We then define min(B,C) as $\max\{L \mid L \leq B, L \leq C\}$ and verify easily that min has all the desired properties.

□

<u>Relation Between the Computation Lattice and the Data-type of Continuous Functions over $\mathcal{D}$</u>

In order to characterize computed partial functions in terms of the semantic interpretation of a given computation lattice, we notice that

<u>Lemma C</u>

<u>For any terms B, C in the computation lattice of T by P, $B \leq C$ implies $b(\Omega) \sqsubseteq c(\Omega)$.</u>

<u>Proof</u>. The proof is straightforward by induction on $\|B\|$:

If $B = A_i$ or $B = X_j$ then $B = C$ and $b(\Omega) \equiv c(\Omega)$.

If $B = G_i(B_1, \ldots, B_{p_i})$, then $C = G_i(C_1, \ldots, C_{p_i})$ and we know by induction that $b_j(\Omega) \sqsubseteq c_j(\Omega)$ for $1 \leq j \leq p_i$. Since $[\lambda x_1, \ldots, x_{p_i}, g_i(x_1, \ldots, x_{p_i})]$ is monotone with respect to any of its arguments, $b(\Omega) \equiv g_i(b_1(\Omega), \ldots, b_{p_i}(\Omega)) \sqsubseteq g_i(c_1(\Omega), \ldots, c_{p_i}(\Omega)) \equiv c(\Omega)$. Finally, if $B = F(B_1, \ldots, B_n)$ then $b(\Omega) \equiv \Omega \sqsubseteq c(\Omega)$.

□

In particular, to any computation sequence $T_0 \to T_1 \to \ldots T_n \to T_{n+1} \to \ldots$ according to some rule $\mathcal{C}$ and input $\bar{D}$, we associate the chain $t_0(\Omega)(\bar{d}) \sqsubseteq t_1(\Omega)(\bar{d}) \sqsubseteq \ldots \sqsubseteq t_n(\Omega)(\bar{d}) \sqsubseteq t_{n+1}(\Omega)(\bar{d}) \sqsubseteq \ldots$

<u>The corresponding computed partial function $\mathcal{C}_p$ is therefore characterized as: $\mathcal{C}_p \equiv \lambda\bar{d} \bigsqcup_{n \geq 0} t_n(\Omega)(\bar{d})$.</u>

From these definitions follows an easy generalization of a theorem of Cadiou [2]:

Theorem 2 (Cadiou)

Any fixed-point of the equation $f \equiv p(f)$ is an extension of any function computed by the program $F \Leftarrow P$.

Proof. For any natural number m, let P^m be defined as $P^0 = F(\bar{X})$ and $P^{m+1} = P\{P^m/F\}$. It is easily seen that $p^i(\Omega) \equiv p(p(\ldots p(\Omega)\ldots))$ (i times). Since Cadiou [2] proved that for any computation sequence $T_0, T_1, \ldots, T_n$ where $T_0 = F(\bar{X})$ we have $T_i \leq P^i$ for all natural numbers i, it follows from Lemma C that $t_i(\Omega) \sqsubseteq p^i(\Omega)$ for all i. The function p being continuous, $f_p \equiv \bigsqcup_{i \geq 0} p^i(\Omega)$, hence $t_i(\Omega) \sqsubseteq f_p$ for any i. It follows that $\mathcal{C}_p \equiv \bigsqcup_{i \geq 0} t_i(\Omega) \sqsubseteq f_p$ and, since $f_p \sqsubseteq f$ for any fixed-point f of p, the conclusion $\mathcal{C}_p \sqsubseteq f$ holds. □

2. Correct Implementation of Recursion

In this section, we try to characterize the computation rules $\mathcal{C}$ such that $\mathcal{C}_p \equiv f_p$ for any program $F \Leftarrow P$, called fixed-point computation rules.

Here are some computation rules we shall consider, both in lang S and lang P :

(1) Call by value: substitute for the leftmost-innermost occurrence of F after simplifications.

(2) Call by name: substitute for the leftmost-outermost occurrence of F after simplifications.

(3) Parallel innermost: substitute for the occurrences of F having all of their arguments free of F's .

(4) Parallel outermost: substitute for all the F's which do not occur in any argument of another F .

(5) Free argument: substitute for all the occurrences of F having at least one of their arguments free of F's after simplifications.

(6) Full substitution: substitute for all the occurrences of F .

2.1 Incorrect Computation Rules

Proposition 1.

In lang P , the rules (1), (2), (3) and (5) are incorrect.

Proof. Consider the program $F(X,Y) \Leftarrow$ IF $X = 0$ THEN 0 ELSE $F(X+1,F(X,Y))*F(X-1,F(X,Y))$ where $*$ is the parallel multiplication function $0*x \equiv x*0 \equiv 0$. The least fixed-point over the integers

(considered as a discrete data-type) of the corresponding functional is the zero function $\lambda x,y$ if $x = \omega$ then ω else 0 . The computation of $F(1,0)$ using (1), (2) or (3) is infinite. As for rule (5), we can take the program $F(X) \Leftarrow X.F(F(X))$ in the data-type of sequences of letters as a counter-example. □

Proposition 2 (Morris [23])

In lang S the rules (1) and (3) are incorrect.

Proof. Consider $F(X,Y) \Leftarrow$ IF X = 0 THEN 0 ELSE $F(X-1,F(X,Y))$. The corresponding least fixed-point over the non-negative integers is again the constant function 0 while the computation of $F(1,0)$ using rules (1) or (3) is infinite. □

2.2 Safe Computation Rules

We now define the class of safe computation rules, and show that they correspond to "correct" implementations of recursion.

Let $\mathcal{C}$ be a computation rule and B an arbitrary term in the computation lattice of T by P . In order to describe the effect of $\mathcal{C}$ on B , we rename F_1 the occurrences of F selected for substitution by $\mathcal{C}$ in B for some input $\bar{D}$, and F_2 the others.

Definition

We say that $\mathcal{C}$ is a safe computation rule if, for any term $B\{F/F_1, F/F_2\}$ in the computation lattice of T by P and for any input $\bar{D}$, $b\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv b\{\Omega/f_1, \Omega/f_2\}(\bar{d})$.

Intuitively, the computation is safe if the values of the F's which are notsubstituted (renamed F_2) are insufficient: as long as more information is not obtained about the other arguments (the F_1's), the information about B cannot be improved.

In order to clarify this definition, let us prove the safeness of some of our computation rules.

Proposition 3

In lang S , the rules (2), i.e., call-by-name and (5), i.e., free argument are safe.

Proof. By induction on $\|C\|$ where $C = \text{simpl}(B)$: we first notice that, because of the semantic definition of lang S , if F occurs in C then $c(\Omega)(d) \equiv \omega$ (remember that C has been simplified and, when a simplified term has the form IF C_1 THEN C_2 ELSE C_3 , we must have F occurring in C_1).

Case $C = A_i$ then any rule is safe.

Case $C = G_i(C_1,\ldots,C_{p_i})$. The letter F occurs necessarily in C , otherwise we could simplify further. Since both rules select at least one F on such terms, we know by our previous remark that

$$c\{\Omega/f_1 , f_p/f_2\}(\bar{d}) \equiv \omega \equiv c\{\Omega/f_1 , \Omega/f_2\}(\bar{d}) \quad .$$

Case $C = F(C_1,\ldots,C_n)$. The safeness of rule (2) is straightforward since the outermost F is substituted. For the same reason, rule (5) is safe if at least one of the C_i is constant. If none of the C_i's is constant, then $c_i\{\Omega/f_1 , f_p/f_2\}(\bar{d}) \equiv \omega$ for $1 \leq i \leq n$ and we must prove that $f_p(\omega,\ldots,\omega) \equiv \omega$. This is ensured by imposing in lang S

that all program variables $X_1,\ldots,X_n$ occur in simpl(P) hence $f_p(\omega,\ldots,\omega) \equiv p(f_p)(\omega,\ldots,\omega) \equiv \omega$. □

Proposition 4

The rules (4), i.e., parallel outermost and (6), i.e., full substitution are safe in both lang S and lang P .

Proof. By induction on $\|B\|$.

Case $B = A_i$. Any rule is safe.

Case $B = G_i(B_1,\ldots,B_{p_i})$. By induction, $b_i\{\Omega/f_1 , f_p/f_2\}(\bar{d}) \equiv b_i\{\Omega/f_1 , \Omega/f_2\}(\bar{d})$ for $1 \leq i \leq p$ in both cases, hence safeness is also satisfied on b .

Case $B = F(B_1,\ldots,B_n)$. Both rules select the outermost F hence $b\{\Omega/f_1 , f_p/f_2\}(\bar{d}) \equiv \omega \equiv b\{\Omega/f_1 , \Omega/f_2\}(\bar{d})$. □

Note that the computation rules that we already recognized as incorrect are all unsafe. In order to prove that safe rules are correct, we need the following technical lemma:

Lemma S

If $\mathcal{C}$ is safe, then $B \overset{\mathcal{C}}{\to} C$ and $\underline{\min}(B,Q) = \underline{\min}(C,Q)$ imply $q(\Omega)(\bar{d}) \sqsubseteq b(\Omega)(\bar{d})$ for any terms B , C and Q in the computation lattice of T by P , and input $\bar{D}$.

Proof. Let us first determine some properties of the min of two terms:

Lemma 3

(i) $\underline{\min}(G_i(B_1,\ldots,B_{p_i}),G_i(C_1,\ldots,C_{p_i})) = G_i(\underline{\min}(B_1,C_1),\ldots,\underline{\min}(B_{p_i},C_{p_i}))$.

(ii) $\underline{\min}(P\{B_1/X_1,\ldots,B_n/X_n\},G(C_1,\ldots,C_p)) = P\{M_1/X_1,\ldots,M_n/X_n\}$ where $M_1,\ldots,M_n$ are such that $F(M_1,\ldots,M_n) = \underline{\min}(F(B_1,\ldots,B_n),G(C_1,\ldots,C_p))$.

Proof. Property (i) is easy and property (ii) follows from the fact that $P\{M_1/X_1,\ldots,M_n/X_n\} \overset{*}{\rightarrow} M' \overset{*}{\rightarrow} P\{B_1/X_1,\ldots,B_n/X_n\}$ with $M_i \overset{*}{\rightarrow} B_i$ for $1 \leq i \leq n$ implies that $M' = P\{M'_1/X_1,\ldots,M'_n/X_n\}$ where $M_i \overset{*}{\rightarrow} M'_i \overset{*}{\rightarrow} B_i$ for $1 \leq i \leq n$.

□

We now prove Lemma S: Let us rename F_1 the occurrences of F selected by $\mathcal{C}$ in B and F_2 the others. Let $M = \underline{\min}(B,Q) = \underline{\min}(C,Q)$.

We first prove by induction on $\langle \text{dist}(M,B) + \text{dist}(M,C), \|M\| \rangle$ that $Q \leq B\{F/F_1, P^m/F_2\}$ for some natural number m . (Here P^m means $P\{P^{m-1}/F\}$ for $m > 0$ and $P^0 = F(X_1,\ldots,X_n)$.)

Case $M = A_i$ or $M = X_j$

In this case, $M = B = C = Q$ and we can choose $m = 0$.

Case $M = G_i(M_1,\ldots,M_{p_i})$

By Lemma 1, $B = G_i(B_1,\ldots,B_{p_i})$, $C = G_i(C_1,\ldots,C_{p_i})$ and $Q = G\ (Q_1,\ldots,Q_{p_i})$. By Lemma 3, $M_i = \underline{\min}(B_i,Q_i) = \min(C_i,Q_i)$ for

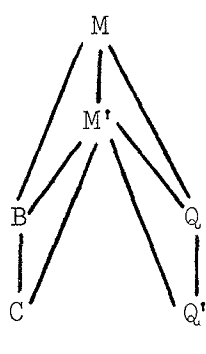
$1 \le i \le p$. It follows by induction that $Q_i \le B_i\{F/F_1 , P^{m_i}/F_2\}$.
We can then choose $m = \sup_{1 \le i \le p_i} \{m_i\}$ in order to get

$Q \le B\{F/F_1 , P^m/F_2\}$.

<u>Case $M = F(M_1,\ldots,M_n)$</u>

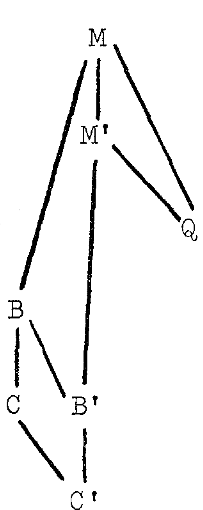
By definition of <u>min</u> , we need only consider the cases:

<u>Case $B = G(B_1,\ldots,B_p)$ and $Q = F(Q_1,\ldots,Q_n)$</u>



Let $M' = P\{M_1/X_1 , \ldots, M_n/X_n\}$ and
$Q' = P\{Q_1/X_1 , \ldots, Q_n/X_n\}$. By Lemma 3,
$M' = \underline{min}(B,\ ') = min(C,Q')$. By Lemma 2,
$dist(M',B) + dist(M',Q') < dist(M,B) + dist(M,Q)$
so we know by induction that
$Q' \le B\{F/F_1 , P^m/F_2\}$ and, a fortiori
$Q \le B\{F/F_1 , P^m/F_2\}$ for some m .

<u>Case $B = F(B_1,\ldots,B_n)$ and $Q = G(Q_1,\ldots,Q_p)$</u>



Since $min(B,Q) = min(C,Q)$, the term C is also of the form $C = F(C_1,\ldots,C_n)$. Let
$M' = P\{M_1/X_1 , \ldots, M_n/X_n\}$, $B' = P\{B_1/X_1 , \ldots, C_n/X_n\}$
and $C' = P\{C_1/X_1 , \ldots, C_n/X_n\}$. By Lemma 3, we know that $M' = min(B',Q) = min(C',Q)$.

By Lemma 2, $\mathrm{dist}(M',B') + \mathrm{dist}(M',Q) < \mathrm{dist}(M,B) + \mathrm{dist}(M,Q)$, and the induction hypothesis tells us that $Q \leq B'\{F/F_1, P^m/F_2\}$. Since the outermost F has not been selected by $\mathcal{C}$ in B then $B' \leq B\{P/F_2\}$. Our last case is then treated since $Q \leq B\{F/F_1, P^{m+1}/F_2\}$.

It is now easy to finish the proof of Lemma S.

For any m , $p^m(\Omega) \sqsubseteq f_p$ implies $b\{\Omega/f_1, p^m(\Omega)/f_2\} \sqsubseteq b\{\Omega/f_1, f_p/f_2\}$. By choosing m large enough, we know that $q(\Omega) \sqsubseteq b\{\Omega/f_1, p^m(\Omega)/f_2\}$ and therefore $q(\Omega) \sqsubseteq b\{\Omega/f_1, f_p/f_2\}$. Since $\mathcal{C}$ is safe, $b\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv b(\Omega)(\bar{d})$ and the conclusion $q(\Omega)(\bar{d}) \sqsubseteq b(\Omega)(\bar{d})$ follows.

□

Theorem 3

Any safe rule is a fixed-point rule.

Proof. In the computation lattice of $T_0 = F(\bar{D})$ by P , let $T_0, T_1, \ldots, T_n, \ldots$ and $S_0, S_1, \ldots, S_n, \ldots$ (where $S_0 = T_0$) be the computation sequences corresponding to respectively some safe rule $\mathcal{C}$ and the full substitution rule. Since $s_n(\Omega) \equiv p^n(\Omega)$ then $\bigsqcup_{n \geq 0} s_n(\Omega) \equiv \bigsqcup_{n \geq 0} p^n(\Omega) \equiv f_p$. We know by Theorem 2 that $\mathcal{C}_p(\bar{d}) \sqsubseteq f_p(\bar{d})$ and it is therefore sufficient to show that $\bigsqcup_{n \geq 0} s_n(\Omega)(\bar{d}) \sqsubseteq \bigsqcup_{n \geq 0} t_n(\Omega)(\bar{d})$, in order to prove $\mathcal{C}_p \equiv f_p$.

Let S_n be an arbitrary term in $S_0, S_1, \ldots$. Since there are only finitely many minorants of S_n in the computation lattice, there exists some m such that $\underline{\min}(T_m, S_n) = \underline{\min}(T_{m+1}, S_n)$. The rule $\mathcal{C}$ being safe, it follows from Lemma S that $s_n(\Omega)(\bar{d}) \sqsubseteq t_m(\Omega)(\bar{d})$, hence

$$\bigsqcup_{n \geq 0} s_n(\Omega)(\bar{d}) \sqsubseteq \bigsqcup_{m \geq 0} t_m(\Omega)(\bar{d}) \ .$$

□

As a corollary, rules (2) and (5) are fixed-point in <u>lang S</u> and rules (4) and (6) are fixed-point rules in both <u>lang S</u> and <u>lang P</u> .

3. An Optimal Implementation of Recursion in lang S

Among the correct implementations of recursion, we now try to determine which ones are efficient. This proves unsuccessful in lang P , but we shall describe an implementation of recursion for lang S which turns out to be optimal.

We already know that, in lang S , "call-by-name" is a fixed-point rule, while "call-by-value" is not. However, "call-by-name" is not an efficient way of computing. For example, in the program $F(X) \Leftarrow \text{IF } X > 0 \text{ THEN } X-1 \text{ ELSE } F(F(X+2))$ the "call-by-name" computation of $F(0)$ would be $\underline{F}(0) \to \underline{F}(F(2)) \to \text{IF } \underline{F}(2) > 0 \text{ THEN } F(2)-1 \text{ ELSE } F(F(F(2)+1)) \to \underline{F}(2)-1 \to 0$.

What happens here is that the term $F(2)$ has been duplicated and subsequently computed twice. We shall describe a computation mechanism, called the delay-rule, which avoids those duplications, and prove its optimality.

3.1 Never Do Today What You Can Put Off Until Tomorrow

A natural way to keep track of duplications of terms is to assign labels to all occurrences of F in a computation sequence, so that copies of the same F will receive the same label. This can be achieved by first labelling differently all F'_s in P ; then, if F is labelled α in T_n and is to be substituted, we label each occurrence of F after substitution by α followed by whatever labelling this particular occurrence had in P . For example, using the same computation as before, and the labelling $\text{IF } X > 0 \text{ THEN } X-1 \text{ ELSE } F_1(F_2(X+2))$ for P , the previous computation can be described as:

$$\underline{F}(0) \rightarrow \underline{F}_2(F_2(2)) \rightarrow \text{IF } F_2(2) > 0 \text{ THEN } F_2(2)-1 \text{ ELSE } F_{11}F_{12}(F_2(2)+2)$$

$$\rightarrow \text{IF } 1 > 0 \text{ THEN } F_2(2)-1 \text{ ELSE } F_{11}F_{12}(F_2(2)+2)$$

simplifies to $\underline{F}_2(2)-1 \rightarrow 0$.

The whole idea of the delay-rule is to modify "call-by-name" so that, whenever some occurrence of F is substituted, all the occurrences having the same label will also be substituted. Hence, the "delay-rule" selects for substitution the leftmost-outermost F in a simplified term, as well as all the other F's having the same label.

Consequently, the delay rule computation of F(0) in the program above is

$$\underline{F}_1(0) \rightarrow \underline{F}_1(F_2(2)) \rightarrow \text{IF } F_2(2) > 0 \text{ THEN } F_2(2)-1 \text{ ELSE } F_{11}F_{12}(F_2(2)+2)$$

$$\rightarrow \text{IF } 1 > 0 \text{ THEN } 1-1 \text{ ELSE } F_{11}(F_{12}(1+2))$$

simplifies to 0 . At this point, it is clear that the "delay rule" is safe (proof similar to that of Proposition 1); what is not clear is that the "delay rule" should be more efficient than "call-by-name" and in fact, in our last example, it was less efficient since it took four substitutions versus three for "call-by-name" in order to obtain its result. When "call-by-name" computed $F_{11}(2)$ twice, the delay rule has been computing it three times! It is a simple exercise in data structuring however to avoid all those recomputations: instead of actually copying various occurrences of some F_α in a term, we simply set some pointers to a unique copy of the term F_α . Whenever any occurrence of F_α is chosen for substitution, the substitution is actually performed in the unique copy of F_α so that <u>all</u> occurrences of F_α are substituted at the price of one substitution.

Going a little bit away from our particular programming language we can sketch an implementation of this idea for, say Algol. The arguments of any procedure should be stored as pointers to formal expressions, together with a tag indicating that those arguments have not yet been computed. Whenever the value of an argument is explicitly needed, (for the evaluation of a conditional or on the right-hand side of an assignment), the tag is tested. If the value of the parameter is already there, we use it; otherwise the corresponding formal expression must be computed, its value kept for further references, and the tag is to be changed. In a machine like the Burroughs B5000 (see, for example, Lonergan-King [12]), the so-called "operand call syllable" would do very nicely: depending on a tag stored with the operand, a load operation on the B5000 gets its argument either directly or through a subroutine call. The delay rule would modify this procedure so that, after the subroutine call, the result would be stored in place of the tagged subroutine descriptor. Of course, one would then have to abandon "side-effects" altogether!

Before proving the optimality of the delay rule let us compare the efficiency of various computation rules on the programs

```
Zer(X) <= IF X > 0 THEN X-1 ELSE Zer(Zer(X+2))
Ack(X,Y) <= IF X = 0 THEN Y+1
                          ELSE IF Y = 0 THEN Ack(X-1,1)
                                        ELSE Ack(X-1,Ack(X,Y-1))
Ble(X,Y) <= IF X = 0 THEN 1 ELSE Ble(X-1,Ble(X-Y,Y))
Fib(X) <= IF X < 2 THEN X ELSE Fib(X-1) + Fib(X-2)
```

over the integers.

	Zer(-2)	Ack(2,1)	Ble(8,2)	Fib(5)
Delay rule	7	14	9	15
Call by name	25	29	9	15
Call by value	7	14	341	15
Free argument	7	23	~ 4000	15
Full substitution */	11	23	~ 10000	15

The entries in this array indicate the number of substitutions required for computing the values at the top of the corresponding column, according to the rules at the left of the rows.

If he has been through those examples, the reader may feel quite disappointed because he can beat the delay-rule in almost all cases. For example, the hand-computation of Fib(5) only requires five substitutions if we are careful never to recompute an argument twice. It would be interesting to study a mechanism in which this type of computation would be possible; namely one could imagine a set of simplification rules which could be augmented dynamically, and allow some computations to be performed by simplifications of the style F(D) → A . In our scheme of things, however, this type of "built-in" values is not possible, since our only means of computation is through substitutions, and we should blame inefficiencies on the program, not on the computation rule.

---

*/ Strictly speaking, we are using the full substitution only on simplified terms, otherwise the computation would always be infinite.

3.2 Optimality of the Delay Rule

So far, we know that the delay rule is safe, and that it never recomputes copies of the same term. Using the same labelling as before, we say that a label F_α is maximal in a term if α is not a proper initial segment of β for any label F_β in the term. A term is simple if all of its labels are maximal. In other words, a term is simple if all computations of various copies of subterms have been pushed to the same point. For example, if $T_0 = F(F(X))$ and $T_0 = G(X,F_1(F_2(X)))$ then $G(G(X,F_1(F_2(X)),F_1(F_2(F(X)))))$ is not simple while $F(G(X,F_1(F_2(X))))$ is simple.

A computation is simple if all F's with the same labels are all treated alike in all substitutions (if one of them is to be substituted, all of them are to be substituted). All terms in a simple computation are necessarily simple. If we are to count for one a substitution of all F's with the same labels, as justified by our previous exercise in data structuring, simple computations are more efficient than others. Namely, if we define $\text{length}(T_0 \overset{*}{\rightarrow} A)$ as the total number of substitutions performed during the computation $T_0 \overset{*}{\rightarrow} A$, we have

Lemma E

For any term A, there exists a simple term $\bar{A}$ with $A \leq \bar{A}$ such that, for any computation $T_0 \overset{*}{\rightarrow} A$ and simple computation $T_0 \overset{*}{\Rightarrow} \bar{A}$, $\text{length}(T_0 \overset{*}{\Rightarrow} \bar{A}) \leq \text{length}(T_0 \overset{*}{\rightarrow} A)$.

Proof. Let $r(C)$ be the number of maximal labels and $s(C)$ be the sum of the lengths of the maximal labels in a term C, while q and p mean respectively the number of occurrences of F in T_0 and P. It

is easily proven by induction on $\text{length}(T_0 \overset{*}{\rightarrow} C)$ that $\text{length}(T_0 \overset{*}{\rightarrow} C) \geq \varphi(C,p,q)$ where $\varphi(C,p,q) = \text{if } p = 1 \text{ then } \frac{s(C)}{q} \text{ else } \frac{r(C)-q}{p-1}$. In a similar way, ($C$ simple) and ($T_0 \overset{*}{\Rightarrow} C$ simple) imply $\text{length}(T_0 \overset{*}{\Rightarrow} C) = \varphi(C,p,q)$.

Given any term A, we can "complete" it into an $\bar{A}$ by substituting P for all occurrences of F with non-maximal labels until there is none left. An $\bar{A}$ constructed in this way will be simple and such that $A \leq \bar{A}$ while $r(A) = r(\bar{A})$. It follows that, for any computation $T_0 \overset{*}{\rightarrow} A$ and simple computation $T_0 \overset{*}{\Rightarrow} \bar{A}$, $\text{length}(T_0 \overset{*}{\Rightarrow} \bar{A}) = \varphi(\bar{A},p,q) = \varphi(A,p,q) \leq \text{length}(T_0 \overset{*}{\rightarrow} A)$.

□

The intuitive meaning of this lemma is very simple: nothing is to be gained by working on individual copies of the same term. At the same price, we get more information by substituting all copies of the same occurrences. In particular, all the computation rules described so far will be improved by "lumping" together occurrences of F with the same labels, thus becoming simple rules. However they may still perform unnecessary substitutions unless

Theorem 4

Any computation rule which is simple, safe and performs at most one substitution at each computation step is optimal.

Proof. Let T_0 be a term, $F(\bar{X}) \Leftarrow P$ a program and $\mathcal{C}$ a safe and simple computation rule performing only one substitution at a time.

Let $T_0 \Rightarrow T_1 \Rightarrow \dots \Rightarrow T_n \Rightarrow T_{n+1} \Rightarrow \dots$ the (simple) computation sequence of T_0 according to $\mathcal{C}$ for some input $\bar{D}$.

If T is a term in the computation lattice of T_0 by P, let us consider an arbitrary computation $T_0 \overset{*}{\rightarrow} T$, and prove that whatever approximation $t(\Omega)(\bar{d})$ of $t_0(f_p)(\bar{d})$ is computed by T will be computed <u>faster</u> by $\mathcal{C}$. For this purpose, we construct $\bar{T}$ as in Lemma E, and consider a simple computation $T_0 \overset{*}{\Rightarrow} \bar{T}$ (the argument in Lemma E not only proves the existence of $\bar{T}$ but also that of a simple computation $T_0 \overset{*}{\Rightarrow} \bar{T}$).

Let i be some natural number such that $T_i \leq \bar{T}$ and $T_{i+1} \not\leq \bar{T}$. Since $\mathcal{C}$ performs only one substitution at the time, this implies $T_i = \underline{\min}(T_{i+1},\bar{T}) = \underline{\min}(T_i,\bar{T})$. By Lemma S, we then know that $\bar{t}(\Omega)(\bar{d}) \sqsubseteq t_i(\Omega)(\bar{d})$. Using Lemmas E and C now, $T \leq \bar{T}$ implies $t(\Omega)(\bar{d}) \sqsubseteq \bar{t}(\Omega)(\bar{d})$ and $\text{length}(T_0 \overset{*}{\Rightarrow} \bar{T}) \leq \text{length}(T_0 \overset{*}{\Rightarrow} T)$. Since both $T_0 \overset{*}{\Rightarrow} \bar{T}$ and $T \overset{*}{\Rightarrow} T_i$ are simple and $T_i \leq \bar{T}$, we have $\text{length}(T_0 \overset{*}{\Rightarrow} T_i) \leq \text{length}(T_0 \overset{*}{\Rightarrow} \bar{T})$ hence $t(\Omega)(\bar{d}) \sqsubseteq t_i(\Omega)(\bar{d})$ while $\text{length}(T_0 \overset{*}{\Rightarrow} T_i) \leq \text{length}(T_0 \overset{*}{\rightarrow} T)$.

□

We shall derive two applications of this theorem.

<u>Corollary 1</u>

<u>The delay rule is optimal in lang S</u>.

<u>Proof</u>. The delay rule has all the properties required by Theorem 4.

□

<u>Corollary 2</u>

<u>In lang S, "call by value" is optimal whenever the least fixed-point f_p corresponding to the program $F(\bar{X}) \Leftarrow P$ is a strict function.</u>

(The function f_p is strict if $f_p(\ldots,\omega,\ldots) \equiv \omega$.)

Proof. Since "call by value" is clearly a simple rule and performs at most one substitution at each step, we only need proving that it is safe whenever f_p is strict. We prove that the substitution $B \to B'$ is safe in that case by induction on $\|C\|$ where $C = \text{simpl}(B)$:

Case $C = A_i$. Any rule is safe.

Case $C = G_i(C_1,\ldots,C_{p_i})$. Same argument as for the safeness of "call by name".

Case $C = F(C_1,\ldots,C_n)$. If F does not occur in any of the C_i's , then the outermost substitution is performed, which is clearly safe. Otherwise, let C_i be the leftmost term in which F occurs. Then, $C_i\{\Omega/f_1 , f_p/f_2\}(\bar{d}) \equiv \omega$ and $C\{\Omega/f_1 , f_p/f_2\}(\bar{d}) \equiv f_p(\ldots,\omega,\ldots) \equiv \omega \equiv C\{\Omega/f_1 , \Omega/f_2\}(\bar{d})$. □

3.3 Sequential Functions

The applications of Theorem 4 given in the previous section do not quite match with the generality of the result. In particular, the data-type on which lang S is computing has no chain of length more than two. What we shall now sketch is a theory of sequential functions, where Theorem 4 finds its full application.

The relevant notion here seems to be

Definition

A function $\lambda x_1,\ldots,x_n.g(x_1,\ldots,x_n)$ in $[D_1 x \ldots x D_n \to D]$ is sequential if, for all $x_1 \in D_1,\ldots,x_n \in D_n$ there exists an $i \in [1,n]$ such

that, for all $y_1,\ldots,y_n$ _such that_ $x_j \sqsubseteq y_j$ _for_ $j\epsilon[1,n]$ _and_ $x_i \equiv y_i$ _we have_ $g(x_1,\ldots,x_n) \equiv g(y_1,\ldots,y_n)$.

Intuitively, g is sequential if, at any given moment, the value of (at least) one of its arguments is crucially needed in order to better approximate the value of the result. For the purpose of our theory, we need to check that sequentiality has the correct closure property, namely

Proposition S

Sequentiality is preserved by composition of functions and fixed-point operators.

Proof.

— _Composition_. If $\lambda z_1,\ldots,z_n g(z_1,\ldots,z_n)$ and $\lambda x_1,\ldots,x_m f_i(x_1,\ldots,x_m)$ for $1 \le i \le n$ are sequential, then

$$\varphi \equiv \lambda x_1,\ldots,x_m g(f_1(x_1,\ldots,x_m),\ldots,f_m(x_1,\ldots,x_m))$$

is also sequential: for any $x_1,\ldots,x_m$ and $i\epsilon[1,n]$, let $z_i \equiv f_i(x_1,\ldots,x_n)$; since g is sequential $z_1,\ldots,z_n$ determines some $i_0\epsilon[1,n]$ and, f_{i_0} being also sequential, $x_1,\ldots,x_m$ determine some $j\epsilon[1,m]$ which can then be used for the sequentiality of φ .

— _Fixed-point operator_. If the functions $\lambda x_1,\ldots,x_n f_i(x_1,\ldots,x_n)$ are sequential for any natural number i , the function

$$\varphi \equiv \lambda x_1,\ldots,x_n \bigsqcup_{i\ge 0} f_i(x_1,\ldots,x_n)$$

is also sequential: for any $x_1,\ldots,x_n$ sequentiality of the f_i's determines a sequence $j_0,j_1,\ldots$ where $j_i\epsilon[1,n]$. At least one of

the j_i's must occur infinitely often in this sequence, and it can be used for proving that φ is sequential.

□

For example, over a discrete data-type, conditional and strict functions are sequential; hence, by Proposition S, all functions definable in lang S are sequential.

In a data-type which is a lattice, the functions $\lambda x,y\ \underline{sup}(x,y)$ and $\lambda x,y\ \underline{inf}(x,y)$ are not sequential in general.

The set Σ^ω of finite or infinite words over some vocabulary Σ becomes a data-type under the partial ordering: $x \sqsubseteq y$ whenever x is an initial segment of y.

In Σ^ω, the functions

$\lambda x.\mathrm{first}(x)$ (take the first letter of x),

$\lambda x.\mathrm{rest}(x)$ (erase the first letter of x),

and $\lambda x,y.x \oplus y$ (append the first letter of x to y) are sequential.*/

This is clear enough for first and rest since any function of one argument is sequential. For $x \oplus y$, if $x \equiv \Lambda$, i.e., x is the empty word, then the first argument is to be chosen for sequentiality since $\Lambda \oplus y \equiv \omega$; otherwise, $x \not\equiv \Lambda$ and any x' such that $x \sqsubseteq x'$ will have the same first letter so that we can use the other argument y for sequentiality.

— Yet another programming language. We define a new language lang GS similar to our previous ones except that all base functions must be sequential.

*/ The relevance of these functions and data-type to parallel programs is shown in Kahn [11].

Let $\mathcal{E}$ be a computation rule, called the generalized delay rule (GDR) defined as follows:

First, using the same type of data-structuring as for the delay rule, $\mathcal{E}$ will be simple.

In any term T, rule $\mathcal{E}$ will select at most one F (or rather set of F's with the same labels) as follows:

If $T = A_i$, no F is chosen.

If $T = G_i(T_1,\ldots,T_{p_i})$, the F will be the F chosen by $\mathcal{E}$ in T_j where j is the index corresponding to the sequentiality of g_i with the arguments $t_1(\Omega)(\bar{d}),\ldots,t_{p_i}(\Omega)(\bar{d})$. Of course, this requires the choice of j to be effective; also, since we want $\mathcal{E}$ to be simple, all F's with the same labels occurring in other subterms are also to be substituted.

If $T = F(T_1,\ldots,T_n)$ the outermost F is selected by $\mathcal{E}$.

We can apply Theorem 4 again in order to prove

Corollary 3

The generalized delay rule is optimal in lang GS.

Proof. Since the GDR is simple and performs at most one substitution at each step, all we need to prove is that it is safe.

The proof is by induction on $\|B\|$ where B is any term in the computation lattice of

$$T_0 = T\{\bar{D}/\bar{X}\} \text{ by } P .$$

The cases $B = A_i$ or $B = F(B_1,\ldots,B_n)$ are easy.

If $B = G_i(B_1,\ldots,B_{p_i})$ and j is the sequentiality index of

$g_i(b_1(\Omega)(\bar{d}),\ldots,b_{p_i}(\Omega)(\bar{d}))$, then $b_j\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv b_j(\Omega)(\bar{d})$ by induction. Since $b_k(\Omega)(\bar{d}) \sqsubseteq b_k\{\Omega/f_1, f_p/f_2\}(\bar{d})$, the very definition of sequentiality gives us $b\{\Omega/f_1, f_p/f_2\}(\bar{d}) \equiv b\{\Omega/f_1, \Omega/f_2\}(\bar{d})$.

□

Conclusion

The results of this chapter generalize quite nicely to a programming language where we introduce assignments, goto's and while statements. What is less clear to the author is how to perform computation in a "typeless" recursive language where procedures can be passed as arguments, say in a full LISP for example. It might also be interesting to study (or prove the non-existence of) optimal computation rules when the simplifications allowed are less restrictive than the ones we chose.

Chapter 3. PROOFS BASED UPON MONOTONICITY

In this chapter, we investigate how far into the theory of computation can one get from the mere hypothesis that programs represent monotone mappings between data-types, thus ignoring continuity.

For this purpose, we introduce a formal system in which the methods of "inductive assertions" and "structural induction" for proving properties of programs can be expressed and justified.

The reader interested in the logic developed here is expected to be familiar with the work of Milner [19]. However, a detailed knowledge of the formalism should not be necessary for understanding the various uses we make of it. In particular, the examples given are described informally, despite the fact that all the proofs can be expressed within the logical system.

1. A Formal System for the Time Being

1.1 Syntax

Terms, which are meant to denote monotone functions of some type, are defined as follows:

(i) Typed identifiers are terms. (We shall almost always omit the type subscript.)

(ii) If s is a term of type $\alpha \to \beta$ and t a term of type α, then $s(t)$ is a term of type β.

(iii) If x is of type α and t of type β, then $[\lambda x.t]$ is a term of type $\alpha \to \beta$.

(iv) If P is a wff, t a term of type α and x a variable, then $[\bigsqcup_{\{x|P\}} t]$ and $[\bigsqcap_{\{x|P\}} t]$ are terms of type α.

A well-formed-formula P is a conjunction of equalities or inequalities between terms of the form $p \sqsubseteq q, r \equiv s, \ldots, u \sqsubseteq t$. A proof is a sequence of implications between wffs $P \vdash Q$, each being derived from the preceding implication by an axiom or a rule of inference.

Variables are bound by λ, $\bigsqcup$ and $\bigsqcap$. We write $s\{t/x\}$ and $P\{t/x\}$ to denote the result of replacing all free occurrences of x in s and P by t, after renaming the necessary bound variables.

1.2 Semantics

A standard model is a denumerable family of complete lattices D_α, one at each type α. Each D_α has a minimal element UU_α and maximal element OO_α. The two base types are I and B. The domain of individuals D_I can be any complete lattice while D_B is $\begin{array}{c}\text{true}\\\downarrow\\\text{false}\end{array}$

If α and β are types, then $\alpha \to \beta$ is also a type and $D_{\alpha\to\beta}$ is the set of monotone mappings from D_α into D_β. It is easily checked that, whenever D_α and D_β are complete lattices, $D_{\alpha\to\beta}$ is itself a complete lattice. Terms of type α are intended to denote elements of D_α.

1.3 Axioms and Rules of Inference

Here x, y, z, f represent variables s, t terms and P, Q, R wffs. Axioms and rules are meant at all syntactically correct types.

(a) Axioms

(Reflexivity)	D1:		$\vdash x \sqsubseteq x$
(Transitivity)	D2:	$x \sqsubseteq y\ ,\ y \sqsubseteq z$	$\vdash x \sqsubseteq z$
(Antisymmetry)	D3:	$x \sqsubseteq y\ ,\ y \sqsubseteq x$	$\vdash x \equiv y$
		$x \equiv y$	$\vdash x \sqsubseteq y\ ,\ y \sqsubseteq x$
(Minimality)	D4:		$\vdash UU \sqsubseteq x$
(Maximality)	D5:		$\vdash x \sqsubseteq OO$
(Monotonicity)	F1:	$x \sqsubseteq y$	$\vdash f(x) \sqsubseteq f(y)$
(λ-conversion)	F2:		$\vdash [\lambda x.s](t) \sqsubseteq s\{t/x\}$
(bottoms-tops)	F3:		$\vdash UU(x) \sqsubseteq UU$
(joins)	F4:	$P\{y/x\}$	$\vdash t\{y/x\} \sqsubseteq \bigsqcup_{\{x \mid P\}} t$
(meets)	F5:	$P\{y/x\}$	$\vdash \bigsqcup_{\{x \mid P\}} t \sqsubseteq t\{y/x\}$
(Inclusion)	W1:	P	$\vdash Q$ (Q is a sub-conjunct of P)

(b) Rules of inference

(Conjunction) R1: $$\frac{P \vdash Q \quad P \vdash R}{P \vdash Q, R}$$

(Cut) R2: $$\frac{P \vdash Q \quad Q \vdash R}{P \vdash R}$$

(Substitution) R3: $$\frac{P \vdash Q}{P\{s/x\} \vdash Q\{s/x\}}$$

(Extensionality) R4: $$\frac{P \vdash f(x) \sqsubseteq g(x)}{P \vdash f \sqsubseteq g}$$ (x not free in P)

(Cases) R5: $$\frac{P\{\underline{false}/x\} \vdash Q \quad P\{\underline{true}/x\} \vdash Q}{P \vdash Q}$$

Here, false and true are abbreviations for UU_B and OO_B respectively.

(meets) R6: $$\frac{Q, P \vdash y \sqsubseteq t}{Q \vdash y \sqsubseteq \bigsqcap_{\{x|P\}} t} \quad (x \text{ not free in } Q)$$

(joins) R7: $$\frac{Q, P \vdash t \sqsubseteq y}{Q \vdash \bigsqcup_{\{x|P\}} t \sqsubseteq y} \quad (x \text{ not free in } Q)$$

1.4 Soundness

In order to establish validity of the axioms and rules of inference, one first ought to make sure that terms without free variables indeed denote elements of the complete lattice of the corresponding type. This is easy for application and λ-abstraction (see Milner [19]). For meets and joins, we have to prove in essence that if for each $i \in I$ the function f_i is monotonic then $\bigsqcap_{i \in I} f_i$ and $\bigsqcup_{i \in I} f_i$ are also monotonic.

Let $x \sqsubseteq y$. For all $i \in I$, we have

$$\bigsqcap_{i \in I} f_i(x) \sqsubseteq f_i(x) \sqsubseteq f_i(y) \sqsubseteq \bigsqcup_{i \in I} f_i(y) \quad .$$

It follows by definition of $\sqcap$ and $\sqcup$ that

$$\bigsqcap_{i \in I} f_i(x) \sqsubseteq \bigsqcap_{i \in I} f_i(y) \quad \text{and} \quad \bigsqcup_{i \in I} f_i(x) \sqsubseteq \bigsqcup_{i \in I} f_i(y) \quad ,$$

and by definition again

$$[\bigsqcap_{i \in I} f_i](x) \sqsubseteq [\bigsqcap_{i \in I} f_i](y)$$

$$[\bigsqcup_{i \in I} f_i](x) \sqsubseteq [\bigsqcup_{i \in I} f_i](y) \quad .$$

Using exactly the same approach as Milner [19], one can then go through the axioms and rules of inference, and justify their validity.

1.5 Pragmatics

We shall use the following abbreviations:

(1) By the Knaster-Tarski theorem, we can characterize the least-fixpoint of $\lambda x.f(x)$ as the greatest-lower-bound of $\{x \mid f(x) \sqsubseteq x\}$. We shall therefore use $\mu x.f(x)$ as an abbreviation for $\underset{\{x \mid f(x) \sqsubseteq x\}}{\sqcap}(x)$. The equivalents of rules F4 and R7 are then:

R8: $\vdash f(\mu x.f(x)) \sqsubseteq \mu x.f(x)$

R9: $f(y) \sqsubseteq y \quad \vdash \mu x.f(x) \sqsubseteq y$

The rule R9 was named fixed-point induction by Park [26].

We shall use the notations $f \Leftarrow \tau(f)$ and f_τ as alternatives to $[\mu f.\tau(f)]$.

(2) One should not confuse the domain D_B: $\begin{matrix}\underline{\text{true}}\\ \downarrow \\ \underline{\text{false}}\end{matrix}$ with the boolean data-type $\begin{matrix}TT & & FF\\ & \searrow\ \swarrow & \\ & \omega & \end{matrix}$. Here D_B should be interpreted as the range of some semi-decision procedure.

Let us now suppose that the domain D_α is characterized by a semi-decision predicate $\lambda x.\mathcal{D}(x)$ mapping D_α into D_B such that $\mathcal{D}(x) \equiv \underline{\text{false}}$ if and only if $x \equiv UU_\alpha$. We can then interpret the logical formula $\forall y \epsilon \mathcal{D}\text{: } P(y)$ as $\underset{\{y \mid \mathcal{D}(y) \equiv \underline{\text{true}}\}}{\sqcap}(P(y))$, where P belongs to $D_\alpha \to D_B$. This justifies using $\forall y \epsilon \mathcal{D}.P(y)$, or, when no confusion can arise, $\forall y.P(y)$ as an abbreviation for $\underset{\{y \mid \mathcal{D}(y) \equiv \underline{\text{true}}\}}{\sqcap}(P(y))$. Similarly, $\exists y.P(y)$ will abbreviate $\underset{\{y \mid \mathcal{D}(y) \equiv \underline{\text{true}}\}}{\sqcup}(P(y))$.

Rules F4, F5, R6 and R7 then translate into the following equivalents to the rules of first-order logic:

(i) $\forall y.P(y) \equiv \underline{true}, \mathcal{D}(a) \equiv \underline{true} \vdash P(a) \equiv \underline{true}$

(ii) $P(a) \equiv \underline{true}, \mathcal{D}(a) \equiv \underline{true} \vdash \exists y.P(y) \equiv \underline{true}$

(iii) $\underline{from}$ $Q, \mathcal{D}(y) \equiv \underline{true} \vdash P(y) \equiv \underline{true}$ (y not free in Q)

$\underline{infer}$ $Q \vdash \forall y.P(y) \equiv \underline{true}$

(iv) $\underline{from}$ $Q, \mathcal{D}(y) \equiv \underline{true} \vdash P(y) \equiv \underline{false}$ (y not free in Q)

$\underline{infer}$ $Q \vdash \exists y.P(y) \equiv \underline{false}$

Examples of Proofs

Example 1. The proof that

$$[\bigsqcup_{\{i|I\}} f(i)](x) \equiv \bigsqcup_{\{i|I\}} f(i)(x)$$

is quite instructive, and we sketch it here:

First $\quad I \vdash f(i) \sqsubseteq \bigsqcup_{\{i|I\}} f(i)$ (F4)

$I \vdash f(i) \sqsubseteq [\bigsqcup_{\{i|I\}} f(i)](x)$ (Appl)

(The rule (Appl) $f \sqsubseteq g \vdash f(x) \sqsubseteq g(x)$ is derivable from F1 and F2.)

$\vdash \bigsqcup_{\{i|I\}} f(i)(x) \sqsubseteq [\bigsqcup_{\{i|I\}} f(i)](x)$ (R7)

then $\quad I \vdash f(i)(x) \sqsubseteq \bigsqcup_{\{i|I\}} f(i)(x)$ (F4)

$I \vdash f(i) \sqsubseteq [\lambda x. \bigsqcup_{\{i|I\}} f(i)(x)]$ (R4)

$\vdash \bigsqcup_{\{i|I\}} f(i) \sqsubseteq [\lambda x. \bigsqcup_{\{i|I\}} f(i)(x)]$ (R7)

$\vdash [\bigsqcup_{\{i|I\}} f(i)](x) \sqsubseteq \bigsqcup_{\{i|I\}} f(i)(x)$ (Appl) and (F2).

Example 2. Let us prove that

(a) $\mu f.s(f,f) \equiv \mu f.s(f,\mu f.s(f,f))$

(b) $\mu f.s(f,f) \equiv \mu f.s(f,s(f,f))$.

In other words, we must establish the equivalence of the following three programs:

$$f \Leftarrow s(f,f)$$

$$g \Leftarrow s(g,f)$$

$$h \Leftarrow s(h,s(h,h)) \quad .$$

Proof of (a). Since $s(f,f) \equiv f$, we know by fixed-point induction that $g \sqsubseteq f$. By monotonicity of s , this implies $s(g,g) \sqsubseteq s(g,f)$. Since $g \equiv s(g,f)$, we have $s(g,g) \sqsubseteq g$ and $f \sqsubseteq g$ follows by fixed-point induction again.

Proof of (b). By definition, $f \equiv s(f,f) \equiv s(f,s(f,f))$ and therefore, $h \sqsubseteq f$ by fixed-point induction.

In order to prove that $f \sqsubseteq h$, let us use the auxiliary program

$$k \Leftarrow s(h,s(h,k)) \quad .$$

Since $s(h,s(h,s(h,h))) \equiv s(h,h)$, the rule of fixed-point induction tells us that

$$k \sqsubseteq s(h,h) \quad ; \qquad (1)$$

but we know by (a) that $k \equiv h$, and (1) becomes $h \sqsubseteq s(h,h)$. By monotonicity of s , this implies $s(h,h) \sqsubseteq s(h,s(h,h))$ which, by definition of h , reduces to $s(h,h) \sqsubseteq h$. One last application of fixed-point induction and we prove $f \sqsubseteq h$.

Example 3. For any functions s and t ,

$$\underline{f_{st} \equiv s(f_{ts})} \quad .$$

That is the programs $f \Leftarrow s(t(f))$ and $g \Leftarrow t(s(f))$ are related by $f \equiv s(g)$ and $g \equiv t(f)$. Since $f_{st} \equiv s(t(f_{st}))$ we have $tf_{st} \equiv tstf_{st}$ and, by fixed-point induction, $\underline{f_{ts} \sqsubseteq tf_{st}}$. By symmetry $f_{st} \sqsubseteq sf_{ts}$ hence $\underline{tf_{st} \sqsubseteq tsf_{ts} \equiv f_{ts}}$.

Example 4. Let $f(x) \Leftarrow g(f(h(x),f(k(x)))$ and $y \Leftarrow g(y,y)$.

We prove that $f(x) \equiv y$. Since $g([\lambda x.y](h(x)) , [\lambda x.y](h(x))) \equiv g(y,y) \equiv y \equiv [\lambda x.y](x)$, we know by fixed-point induction that $f \sqsubseteq [\lambda x.y]$ hence $\underline{f(x) \sqsubseteq y}$. On the other hand, $g(f(UU),f(UU)) \sqsubseteq g(f(h(UU)) , f(k(UU)))$ by monotonicity, and $g(f(UU),f(UU)) \sqsubseteq f(UU)$ follows from $f(UU) \equiv g(f(h(UU),f(k(UU))))$. We conclude $y \sqsubseteq f(UU)$ by fixed-point induction and, since $f(UU) \sqsubseteq f(x)$, we proved that $\underline{y \sqsubseteq f(x)}$.

Example 5. If the two functions $\lambda f.s(f)$ and $\lambda f.t(f)$ commute, i.e., $st \equiv ts$ then Example 2 tells us that $f_{st} \equiv s(f_{st})$ and $f_{ts} \equiv t(f_{ts})$ so that $f_s \sqsubseteq f_{st}$ and $f_t \sqsubseteq f_{st}$. (We can say that f_s and f_t are weakly equivalent.)

The similarity between some of those results and better known ones in linear algebra should not surprise us since linear algebra can be used as a model of our formal system. The base domain D_I will be the set of vector-space over some space $\mathcal{V}$. The natural ordering

is inverted: $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2$ holds whenever $\mathcal{V}_2$ is a subspace of $\mathcal{V}_1$. The minimal element UU corresponds to the space $\mathcal{V}$ itself while the vector space containing only $\vec{0}$ corresponds to OO. Linear transformations over $\mathcal{V}$ are then monotone mappings in $D_I \to D_I$ with respect to that ordering, and, if the dimension of $\mathcal{V}$ is infinite, they are not continuous in general. The least fixed-point of a linear transformation $A \in \mathcal{D}_I \to \mathcal{D}_I$ is then the eigenspace of A having maximal dimension.

1.6 <u>A Possible Weakness of the System</u>

Let us consider the inference rule

$$\mathrm{RT}: \frac{P, x \sqsubseteq g(x) \ \vdash \ f(x) \sqsubseteq g(f(x))}{P \ \vdash \ \mu x.f(x) \sqsubseteq g(\mu x.f(x))} \qquad (x \text{ not free in } P)$$

<u>Is RT provable or not within our system?</u> Although we have not been able to settle this question, we shall be able to show that rule RT must be valid in any standard model of our formal system.

Before doing so, let us point out that fixed-point induction can be derived from RT and that using RT would somewhat simplify the proofs in the previous examples. For instance, the proof that $f \sqsubseteq h$, where $f \equiv \mu x.s(x,x)$ and $h \equiv \mu x.s(x,s(x,x))$ could go as follows: Let us assume $y \sqsubseteq h$ and $y \sqsubseteq s(y,y)$. In order to apply rule RT, we shall prove that

$$y \sqsubseteq h, y \sqsubseteq s(y,y) \ \vdash \ s(y,y) \sqsubseteq h, s(y,y) \sqsubseteq s(s(y,y),s(y,y))$$

and therefore conclude that $\vdash f \sqsubseteq h, f \sqsubseteq s(f,f)$ so, a-fortiori $\vdash f \sqsubseteq h$.

By monotonicity $y \sqsubseteq s(y,y) \vdash s(y,y) \sqsubseteq s(s(y,y),s(y,y))$ and $y \sqsubseteq s(y,y) \vdash s(y,y) \sqsubseteq s(y,s(y,y))$. Therefore, using monotonicity three times again $y \sqsubseteq s(y,y)\,,\, y \sqsubseteq h \vdash y \sqsubseteq s(h,s(h,h))$. But $h \equiv s(h,s(h,h))$ and, putting everything together, we get $y \sqsubseteq h\,,\, y \sqsubseteq s(y,y) \vdash s(y,y) \sqsubseteq h, s(y,y) \sqsubseteq s(s(y,y),s(y,y))$.

We shall now justify the rule. To each monotone function t mapping $\mathcal{D} \to \mathcal{D}$ and ordinal number α , we associate an element $t^{\alpha}(UU) \in \mathcal{D}$ as follows:

(i) $t^{0}(UU) \equiv UU$

(ii) $t^{\alpha+1}(UU) \equiv t(t^{\alpha}(UU))$

(iii) If $\alpha = \lim_{\beta<\alpha}(\beta)$ is a limit ordinal, $t^{\alpha}(UU) \equiv \bigsqcup_{\beta<\alpha} \{t^{\beta}(UU)\}$.

More concisely, $t^{\alpha}(UU) \equiv t(\bigsqcup_{\beta<\alpha} \{t^{\beta}(UU)\})$, if we agree that $\bigsqcup(\emptyset) \equiv UU$.

This sequence has the properties that $\beta < \gamma$ implies $t^{\beta}(UU) \sqsubseteq t^{\gamma}(UU) \sqsubseteq f_t$ for all ordinals β and γ , and $t^{\alpha}(UU) \equiv t^{\alpha+1}(UU)$ implies $t^{\alpha}(UU) \equiv f_t$ for any ordinal α .

Hence, if we choose α to be the first ordinal not embeddable in $\mathcal{D} \to \mathcal{D}$, the sequence $t^{0}(UU), t^{1}(UU), \ldots, t^{\alpha}(UU)$ has "too many" elements and $t^{\alpha}(UU) \equiv f_t$. (See Cadiou [2] or Hitchcock-Park [8].)

Now, from the hypothesis $F \sqsubseteq s(F) \vdash t(F) \sqsubseteq s(t(F))$, we can deduce that, for all ordinals α ,

$$t^{\alpha}(UU) \sqsubseteq s(t^{\alpha}(UU)) \quad . \tag{1}$$

If α is not a limit ordinal, (1) is easy to establish. If α is a limit ordinal $\alpha = \lim_{\beta<\alpha}(\beta)$, then for all $\beta < \alpha$ we know that

$t^{\beta}(UU) \sqsubseteq s(t^{\beta}(UU))$. Since $t^{\beta}(UU) \sqsubseteq t^{\alpha}(UU)$ we know that $t^{\beta}(UU) \sqsubseteq s(t^{\alpha}(UU))$ and therefore $t^{\alpha}(UU) \equiv \bigsqcup_{\beta<\alpha} \{t^{\beta}(UU)\} \sqsubseteq s(t^{\alpha}(UU))$. Choosing α such that $t^{\alpha}(UU) \equiv f_t$ then yields the conclusion of rule RT .

2. Justification of Some Proof Techniques

Suitable choices of the semantic definition of programming languages allow to reduce most of the proof techniques described in the literature to the rule of fixed-point induction. In particular, this applies to the methods described in McCarthy [13], Naur [24], Floyd [7], Manna [14], Manna-Pnueli [16], and Hoare [9]. Since Hoare's technique has been justified in Manna-Vuillemin [17], and the connections between fixed-point induction and the Manna-Pnueli method have been explicited by Park [26], we shall limit ourselves to first indicating how the Floyd-Naur method can be explained within our formal system and then sketch the connections with structural induction. The basic ideas in this section are from Park [26].

2.1 Description of a Flowchart-language

A flowchart is a connected graph, with two distinguished nodes (START) and (HALT). Nodes can be of the type assignment [$X_i \leftarrow F(\bar{X})$] or test ($P(\bar{X})$, with exits T and F). Following Floyd [7], the "meaning assigned" to such a program will be a <u>relation</u> $\psi(\bar{x}_H)$ over the values of the program variables, at the (HALT) node. This output relation is obtained by "carrying along" an input relation $\varphi(\bar{x}_s)$, holding of the program variables at the (START) node. The notation $\psi \equiv \Sigma$ (φ , (START) → [β] → (HALT)) therefore means that, whenever we start

the execution of β with inputs satisfying φ , the outputs, if any, must satisfy ψ .

As in Chapter 2, syntactic objects are represented by upper-case letters and associated semantic objects by the corresponding lower-case letters.

The semantic function Σ is defined recursively as:

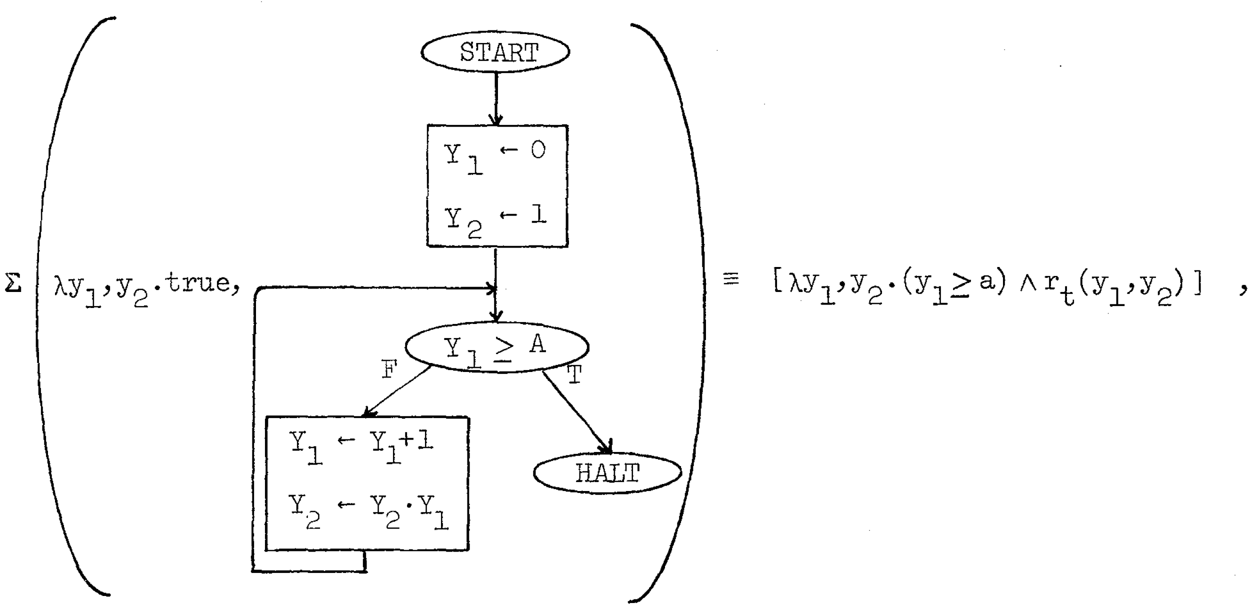
(i) $$\Sigma\left(\pi, \boxed{X_i \leftarrow F(\bar{X})}\right) \equiv \lambda\bar{x}(\exists\bar{y})[\pi(\bar{y}) \wedge x_i = f(\bar{y}) \wedge (\bigwedge_{j \neq i} x_j = y_j)]$$

(ii) $$\Sigma\left(\pi, \begin{array}{c}\boxed{B_1}\\ \downarrow \\ \boxed{B_2}\end{array}\right) \equiv \Sigma\left(\Sigma\left(\pi, \boxed{B_1}\right), \boxed{B_2}\right)$$

(iii) $$\Sigma\left(\pi, \begin{array}{c} \textcircled{Q} \\ T \swarrow \quad \searrow F \\ \boxed{B_1} \quad \boxed{B_2} \end{array}\right) \equiv \underline{\text{if}}\ q\ \underline{\text{then}}\ \Sigma\left(\pi \wedge q, \boxed{B_1}\right)\ \underline{\text{else}}\ \Sigma\left(\pi \wedge \sim q, \boxed{B_2}\right)$$

(iv) $$\Sigma\left(\pi, \begin{array}{c} \textcircled{Q} \\ F \swarrow \quad \searrow T \\ \boxed{B} \end{array}\right) \equiv q \wedge \left[\mu\sigma.\pi \vee \Sigma\left(\sim q \wedge \sigma, \boxed{B}\right)\right]$$

Equation (iv), expressing the semantics of goto's, defines the "minimum valid inductive assertion" described in Manna [14]. There will be essentially one such equation per loop in the program; this may lead to systems of mutually recursive relations, depending on the nature of nesting of the loops. According to this definition, we have for example:



where $t(r)(y_1,y_2) \equiv [(y_1 = 0) \wedge (y_2 = 1)] \vee$

$[\exists x_1,x_2.(x_1 < a) \wedge r(x_1,x_2) \wedge (y_1 = x_1+1) \wedge (y_2 = (x_1+1).x_2)]$.

Note that, in order to simplify our semantic description, we have in effect limited ourselves to considering a flowchart in block-form. If loops do not have this nice nested structure, the description would be slightly more complex, and we would need to express the semantics of ill-nested loops by systems of mutually recursive equations.

2.2 The Inductive Assertions Technique

The meaning of a flowchart program is now a (partial) predicate, defined as the least-fixed point of some equation, say $r \equiv t(r)$. If we can find an "inductive assertion" q such that $t(q) \sqsubseteq q$, the rule of fixed-point induction allows us to infer that $r_t \sqsubseteq q$. This shows that whenever the program terminates, that is, if $r_t(\bar{d}) \equiv \underline{true}$ for some input $\bar{d}$, then we must also have $q(\bar{d}) \equiv \underline{true}$.

This will be best understood by using the same example as above: The expression $t(q) \sqsubseteq q$ is

$$[y_1 = 0) \wedge (y_2 = 1] \vee [\exists x_1, x_2 . (x_1 \neq a) \wedge q(x_1, x_2) \wedge (y_1 = x_1 + 1) \wedge (y_2 = (x_1 + 1) . x_2)]$$

$$\sqsubseteq \ q(y_1, y_2) \quad .$$

Using the inference rules corresponding to those of predicate calculus in Section 1, this formula is equivalent to

$$\vdash \ q(0,1) \equiv \underline{true}$$

and

$$q(y_1, y_2) \wedge y_1 \neq a \equiv true \ \vdash \ q(y_1+1, (y_1+1) . y_2) \equiv true \quad .$$

This last formulation is the direct translation within our formalism of the verification condition derived by Manna [14]. This justification of the method gives us the additional insight that the inductive assertions one may use for proving the partial correctness of some program by the Manna-Floyd method <u>are exactly the fixed-points of some algorithmically constructed functional</u>.

2.3 Termination of Programs

Following Park [26], we shall now prove that the rule of fixed-point induction allows us to derive instances of (mathematical) transfinite induction.

Let $\mathcal{D}$ be a domain, and $\prec$ a partial ordering on $\mathcal{D}$. For any relation R mapping $\mathcal{D}$ into $\begin{matrix}\text{true}\\ \downarrow\\ \text{false}\end{matrix}$, let

$t(R)(x) \equiv [\forall y.\ \underline{\text{if}}\ y \prec x\ \underline{\text{then}}\ R(y)\ \underline{\text{else}}\ \text{true}]$. The least fixed-point of t is then the maximal well-ordered initial segment of the ordering $\prec$ over $\mathcal{D}$. (Note that this is the first time that we use a monotone function which is not continuous.)

Example. Let us consider some orderings over the integers, and the corresponding R_t.

If $\prec$ is $1 \prec 2 \prec 3 \prec \ldots$ then $R_t \equiv t^{\omega}(UU)$ and $R_t(n)$ holds for every n.

If $\prec$ is $\ldots \prec 3 \prec 2 \prec 1$ then $R_t \equiv UU$ never holds.

If $\prec$ is $1 \prec 3 \prec 5 \ldots 2 \prec 4 \prec \ldots$, then $R_t \equiv t^{2\omega}(UU)$ and $R_t(n)$ holds for every n.

If $\prec$ is $1 \prec 3 \prec 5 \ldots \prec 6 \prec 4 \prec 2$, then $R_t \equiv t^{\omega}(UU)$ and $R_t(n)$ holds only of the odd natural numbers.

If $\prec$ is $1 \prec 3 \prec 5 \ldots 2 \prec 6 \prec 10 \prec \ldots 4 \prec 12 \prec 20 \prec \ldots \ldots$, then $R_t \equiv t^{\omega^2}(UU)$ and $R_t(n)$ holds for every n. $\square$

If $\prec$ is a well-founded relation over $\mathcal{D}$, then $R_t(x)$ holds for any element x of $\mathcal{D}$, in which case the "program" $R(x) \Leftarrow t(R)(x)$ can be thought of as defining recursively our domain.

In other words, if

$$WO \equiv \mu R.\lambda <,x.[(\forall y)\ \underline{if}\ y < x\ \underline{then}\ R(y)\ \underline{else}\ true]\quad,$$

the equality $WO(<)(x) \equiv \mathcal{D}(x)$ characterizes the relation $<$ as being well-founded. (See also Hitchcock-Park [8] for a more elegant formulation of this equality.)

No matter what kind of ordering $<$ is, fixed-point induction translates into the following rule:

$$[(\forall y).\ \underline{if}\ y < x\ \underline{then}\ P(y)\ \underline{else}\ true] \sqsubseteq P(x)\ \vdash\ WO(<)(x) \sqsubseteq P(x)\ .$$

And in particular, if $<$ is well founded over $\mathcal{D}$, then $P(x) \equiv true$ will hold for any x in $\mathcal{D}$. Depending on the interpretation of $<$, this is a formulation of structural induction or transfinite induction (see Chapter 4, Section 3).

For example, the termination of the program

$$\begin{cases} F(n) \Leftarrow \underline{if}\ odd(n)\ \underline{then}\ n\ \underline{else} \\ \qquad\qquad \underline{if}\ G(n) = 1\ \underline{then}\ F(\frac{3n}{2})\ \underline{else}\ F(\frac{n}{G(n)} \cdot F(n - \frac{2n}{G(n)}) + \frac{n}{2G(n)}) \\ G(n) \Leftarrow \underline{if}\ even(n)\ \underline{then}\ G(n/2)\ \underline{else}\ n \end{cases}$$

over the natural numbers can be established using the well ordering

$$(1 < 3 < 5 < \ldots) < (2 < 6 < 10 < \ldots) < (4 < 12 < 20 < \ldots) < (\ldots)\ \ldots$$

More examples of applications of this technique will be given in the next chapter.

Chapter 4. PROOFS BASED UPON CONTINUITY

The previous chapter was a first attempt at proving properties of programs, based on a rather weak theory of computation. We shall now use our knowledge that programs are continuous functions, and justify some other proof techniques. The presentation will again be quite informal. However, it should soon be apparent that all the proofs given can be formalized in Milner's Logic for Computable Functions (LCF), as described in Section 1 of this chapter.

Obviously we wish to preserve all the results obtained in the previous chapter. As far as formal systems are concerned, one could achieve this by embedding LCF in the logic described in Chapter 3. In this mixed system, terms would be (syntactically) recognizable as being monotone or continuous, and the appropriate rules of inference could be applied accordingly. The logic would not be very different from the other two we describe in this work. For example, a good candidate for the induction rule would be

$$\text{rule M:}\quad \frac{P \vdash g(UU) \sqsubseteq h(UU) \qquad P, g(x) \sqsubseteq h(x) \vdash g(f(x)) \sqsubseteq h(f(x))}{P \vdash g(\mu x.f(x)) \sqsubseteq h(\mu x.f(x))}$$

where x must not be free in P and g must be continuous, while h and f only need be monotone. (This rule was independently suggested by Hitchcock-Park [8].) Its justification is very similar to that of rule RT in the preceding chapter.

Remarkably enough, there seems to be no real need to get involved in this rather complex mixed system: as long as all the terms used in the proofs denote computable functions, any of the results of Chapter 3

will still hold in LCF. For example, if we restrict ourselves to using only computable assertions, the inductive assertions method can be justified in exactly the same way. The only technique for which this constitutes a real problem is transfinite induction, and we shall give it special attention in Section 2.1.

1. Description of LCF

The formal system that we shall use is, except for some trivial changes, taken from Milner [18]. It is a typed λ-calculus version of a logic designed by Scott [30]. (We assume the reader who is interested in the technical details to be familiar with Milner's work.)

1.1 Syntax

The terms of the logic are intended to denote the computable functions of various types. Each term should therefore be subscripted with its type, but we shall almost always omit this subscript.

Terms are defined recursively as:

(1) Identifiers: $g, p, F, \tau, \sigma, x, y \ldots$ (at each type) or constants: UU (at each type) TT, FF (at the type Boolean) are terms.

(2) If s is of type $\alpha \to \beta$ and t of type α, then $s(t)$ is a term of type β.

(3) If s is of type α, and x of type β, then $[\lambda x.s]$ is a term of type $\beta \to \alpha$.

(4) If p is of type boolean, s and t of type α, then

<u>if</u> p <u>then</u> s <u>else</u> t

is a term of type α.

(5) If f and s are of type α, then $[\mu f.s]$ is a term of type α.

As an alternative to $[\mu f.s]$, we shall also use the notations f_τ, $f \Leftarrow \tau(f)$ and $\tau : f \Leftarrow s$, where $\tau \equiv [\lambda f.s]$.

A wff is a conjunction of equalities $s \equiv t$ or inequalities $s \sqsubseteq t$ between terms, separated by commas.

A proof is a sequence $\Phi_0 \vdash \Psi_0, \ldots, \Phi_n \vdash \Psi_n$ of implications between wffs, each of which is obtained by application of the rules of inference, or use of the axioms.

For any term s or wff Φ, we write $s\{t/x\}$ and $\Phi\{t/x\}$ to designate the result of substituting t for all the free occurrences of x in s and Φ. An occurrence of x is not free if it is bound by λx or μx.

1.2 Axioms and Rules of Inference

In this description, x, y, z, f denote variables, s and t terms, P, Q, R wffs.

(a) Axioms

About the Domains

(Reflexivity)	D1:		$\vdash$	$x \sqsubseteq x$
(Transitivity)	D2:	$x \sqsubseteq y, y \sqsubseteq z$	$\vdash$	$x \sqsubseteq z$
(Antisymmetry)	D3:	$x \sqsubseteq y, y \sqsubseteq x$	$\vdash$	$x \equiv y$
(Minimality)	D4:		$\vdash$	$UU \sqsubseteq x$

About the Functions

(Monotonicity)	F1:	$x \sqsubseteq y \vdash$	$f(x) \sqsubseteq f(y)$
(Fixed point)	F2:	$\vdash$	$f(\mu x.f(x)) \sqsubseteq \mu x.f(x)$
(λ-conversion)	F3:	$\vdash$	$[\lambda x.s](t) \equiv s\{t/x\}$
(bottoms)	F4:	$\vdash$	$UU(x) \sqsubseteq UU$
(conditionals)	F5:	$\vdash$	$\underline{\text{if}}\ UU\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y \equiv UU$
		$\vdash$	$\underline{\text{if}}\ TT\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y \equiv x$
		$\vdash$	$\underline{\text{if}}\ FF\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y \equiv y$

About Formulaes

(Inclusion) W1: $P \vdash Q$ (Q is a subset of P)

(b) Rules of Inference

(Conjunction) R1: $\dfrac{P \vdash Q \quad P \vdash R}{P \vdash Q,R}$

(Cut) R2: $\dfrac{P \vdash Q \quad Q \vdash R}{P \vdash R}$

(Substitution) R3: $\dfrac{P \vdash Q}{P\{s/x\} \vdash Q\{s/x\}}$

(Extensionality) R4: $\dfrac{P \vdash f(x) \sqsubseteq g(x)}{P \vdash f \sqsubseteq g}$ (x not free in P)

(Cases) R5: $\dfrac{P\{UU/x\} \vdash Q \quad P\{TT/x\} \vdash Q \quad P\{FF/x\} \vdash Q}{P \vdash Q}$

(Computation induction) R6: $\dfrac{P \vdash Q\{UU/x\} \quad P,Q \vdash Q\{f(x)/x\}}{P \vdash Q\{\mu x.f(x)/x\}}$ (x not free in P)

1.3 Some Remarks About the Logic

Incompleteness

Using the fact that natural numbers can be defined implicitly within the system, Scott [30] showed that the set of valid implications $P \vdash Q$ is not recursively enumerable, i.e., the logic is incomplete. It also follows directly from the undecidability of equivalence between program schemas that the set of valid theorems $\vdash P$ is not recursively enumerable.

On the other hand, if we just consider terms which correspond to Ianov-schemas (Ianov [10]), the logic becomes complete. (This was proved independently by J. W. deBakker and R. Milner.) Another decidable sub-theory of LCF is described in Courcelles-Kahn-Vuillemin [3].

The Induction Rule is a Generalization of McCarthy's Recursion Induction

We shall use the fixed-point induction formulation of McCarthy's rule: $f(y) \sqsubseteq y \vdash \mu x.f(x) \sqsubseteq y$. This rule is easily derivable from computation induction. In order to show that computation induction *cannot* be derived from fixed-point induction,[*] we shall exhibit a theorem of the logic which cannot be proved by fixed-point induction. One such theorem is:

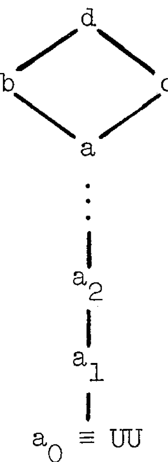
$$\sigma(\tau(x)) \equiv \tau(\sigma(x)), \sigma(UU) \equiv \tau(UU) \vdash \mu x.\sigma(x) \equiv \mu x.\tau(x) \quad .$$

In order to prove that it cannot be derived using only fixed-point induction, notice that after removing the induction rule, neither the

[*] More precisely, if we replace the induction rule of LCF by fixed-point induction, the set of theorems of this modified logic is a strict subset of the theorems of LCF.

axioms, nor the inference rules require continuity in order to be valid. We can thus define the following countermodel:

Terms will denote the hierarchy of monotone functions constructed over the following base domain:



The counterexample to our theorem is provided by the functions f and g defined by

$$f(a_i) \equiv g(a_i) \equiv a_{i+1} \quad ; \quad f(a) \equiv f(b) \equiv b \quad ;$$

$$f(c) \equiv f(d) \equiv g(b) \equiv g(d) \equiv d \quad ; \quad g(a) \equiv g(c) \equiv c \quad .$$

These two functions satisfy the hypothesis but not the conclusion -- $f(UU) \equiv g(UU)$, $fg \equiv gf$ while $\mu x f(x) \not\equiv \mu x g(x)$ -- of our theorem, which is therefore not provable within this system.*/ Actually, the same example can be used to prove that rule RT (see Chapter 3, Section 1.6) is also less powerful than computation example.

The theorem is in itself an interesting one and gives in some cases an elegant way for proving equivalence between programs. For example, the functionals

---

*/ With some slight changes, this counterexample can be used to answer a question raised by Scott [30].

$$P_1(F)(x,y) \equiv \underline{if}\ x = 0\ \underline{then}\ y\ \underline{else}\ F(x-1,y+1)$$

$$P_2(F)(x,y) \equiv \underline{if}\ x = 0\ \underline{then}\ y\ \underline{else}\ F(x-1,y)+1$$

and

$$P_3(F)(x,y) \equiv \underline{if}\ x = y\ \underline{then}\ y\ \underline{else}\ x.F(x+1,y)$$

$$P_4(F)(x,y) \equiv \underline{if}\ x = y\ \underline{then}\ x\ \underline{else}\ y.F(x,y-1)$$

over the natural numbers are such that:

$$P_1(UU) \equiv P_2(UU) \quad , \quad P_1P_2 \equiv P_2P_1 \quad \text{and} \quad P_3(UU) \equiv P_4(UU) \quad , \quad P_3P_4 \equiv P_4P_3 \ .$$

The proofs of equivalence between $F \Leftarrow P_1(F)$, $F \Leftarrow P_2(F)$ and $F \Leftarrow P_3(F)$, $F \Leftarrow P_4(F)$ respectively then follow.

1.4 Some Examples of Proofs

In order to demonstrate some practical aspects of the method, we shall present some examples of proofs by computation induction.

To improve readability, the following conventions will be adopted from now on:

(1) We shall omit the proofs that $f(\ldots,UU,\ldots) \equiv UU$ whenever they are straightforward.

(2) We shall use freely the equality

$$f(\ldots,\ \underline{if}\ p\ \underline{then}\ a\ \underline{else}\ b,\ \ldots) \equiv \underline{if}\ p\ \underline{then}\ f(\ldots,a,\ldots)\ \underline{else}\ f(\ldots,b,\ldots)$$

whenever it is easy to establish that $f(\ldots,UU,\ldots) \equiv UU$.

(3) In the arguments by cases on some variable p , we shall omit the case $p \equiv UU$ whenever it causes no problem.

(4) We shall use the parallel induction rule for systems of mutually recursive definition. Let us describe the situation on the example $\begin{cases} F \Leftarrow \sigma(F,G) \\ G \Leftarrow \tau(F,G) \end{cases}$, the generalization to more complex systems being straightforward. The rule we wish to use is

$$\frac{\begin{array}{l} P \vdash Q\{UU/x\}\{UU/y\} \\ P,Q \vdash Q\{\sigma(x,y)/x\}\{\tau(x,y)/y\} \end{array}}{P \vdash Q\{F/x\}\{G/y\}} \qquad (x\,,\,y \text{ not free in } P)$$

Actually, a more accurate notation would be $F \equiv \mu f.\sigma(f,\mu g.\tau(f,g))$ and $G \equiv \mu g.\tau(\mu f.\sigma(f,g),g)$.

The justification of this rule in the general case can be found in deBakker-Scott [6] or Hitchcock-Park [8].

If F and G happen to have the same type, we can also use the following more intuitive justification of the rule:

Using the pairing function $\pi \equiv \lambda x,y.[\lambda p.\underline{\text{if}}\ p\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y]$, we can define $\mathcal{F} \equiv \pi(F,G)$. The components are then retrieved as $F \equiv \mathcal{F}(TT)$ and $G \equiv \mathcal{F}(FF)$, and $\mathcal{F}$ can be defined by $\mathcal{F} \Leftarrow \pi(\sigma(\mathcal{F}(TT),\mathcal{F}(FF)),\tau(\mathcal{F}(TT),\mathcal{F}(FF)))$. The previous rule is then a direct translation of the ordinary computation induction as applied to $\mathcal{F}$.

(5) For all the examples where computations are meant over some specific data-type -- integer, natural numbers, sets, lists, etc. ... -- we assume implicitly that the axioms for the corresponding data-types are put as premises of the Ways to axiomatize those various domains are described in Milner-Weyrauch [21] and in Newey [25].

Example 1. Let us consider the program schema

$$\tau_n: f(x) \Leftarrow \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f^n(h(x)) \quad ,$$

where $f^n(x) \equiv f(f(\ldots(f(x)\ldots)$ (n times), and $f^0(x) \equiv x$.

We wish to prove that the equality $f_{\tau_n} \equiv f_{\tau_m}$ holds for all natural numbers $n \geq 1$ and $m \geq 1$.

We shall first prove that

$$f_{\tau_n}^{k+1} \equiv f_{\tau_n} \qquad \text{for any } k \geq 0 \quad . \tag{a}$$

Let $P[f]$ be $f_{\tau_n}^k f \equiv f$. We shall prove $P[f_{\tau_n}]$ by computation induction.

Base If $f \equiv UU$ then $P(UU)$ is $f_{\tau_n}^k (UU) \equiv UU$, i.e., $f_{\tau_n}^k (UU(x)) \equiv UU(x)$ which is easily verified, assuming $p(UU) \equiv UU$.

Induction Assuming that $P(f)$ is true,

$$f_{\tau_n}^k (\tau_n(f)(x)) \equiv f_{\tau_n}^k (\underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f^n(h(x)))$$

(definition of τ_n)

$$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f_{\tau_n}^k f^n h(x)$$

(properties of f_{τ_n})

$$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f^n h(x)$$

(induction hypothesis)

$$\equiv \tau_n(f)(x) \quad .$$

Now that equation (a) has been proved, let us consider

$$\tau_m(f_{\tau_n})(x) \equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f_{\tau_n}^m\ h(x)$$

$$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f_{\tau_n}\ h(x) \qquad \text{(by (a))}$$

$$\equiv \underline{\text{if}}\ p(x)\ \underline{\text{then}}\ x\ \underline{\text{else}}\ f_{\tau_n}^n\ h(x) \qquad \text{(by (a) again)}$$

$$\equiv \tau_n(f_{\tau_n})(x) \equiv f_{\tau_n}(x) \quad .$$

It follows by fixed-point induction that $f_{\tau_m} \sqsubseteq f_{\tau_n}$ and by symmetry $f_{\tau_n} \sqsubseteq f_{\tau_m}$. ☐

Example 2. Let us consider the two "squaring" programs

$$\tau:\ F(x,y,z) \Leftarrow \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ y\ \underline{\text{else}}\ F(x-1,y+z,z)$$

and

$$\sigma:\ G(x,y) \Leftarrow \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ y\ \underline{\text{else}}\ G(x-1,y+2x-1) \quad ,$$

over the natural numbers. We wish to show that $f_\tau(x,0,x) \equiv g_\sigma(x,0)$. Let $P(f,g)$ be $f(y,x(x-y),x) \equiv g(y,x^2-y^2)$. If we can prove $P(f_\tau,g_\sigma)$, the desired conclusion will follow by choosing x equal to y .

Base Proving $P(UU,UU)$ is straightforward.

Induction Assuming $P(f,g)$, consider

$$\tau(f)(y,x(x-y),x) \equiv \underline{\text{if}}\ y = 0\ \underline{\text{then}}\ x(x-0)\ \underline{\text{else}}\ f(y-1,x(x-y)+x,x)$$

(definition of τ)

$$\equiv \underline{\text{if}}\ y = 0\ \underline{\text{then}}\ x^2\ \underline{\text{else}}\ f(y-1,x(x-(y-1)),x)$$

$$\equiv \underline{\text{if}}\ y = 0\ \underline{\text{then}}\ x^2-0^2\ \underline{\text{else}}\ g(y-1,(x^2-y^2)+2y-1)$$

(induction hypothesis)

$$\equiv \sigma(G)(y,x^2-y^2) \quad .$$

☐

Example 3. (S. Ness) Let us consider the following two LISP functions

$$F(x) \Leftarrow \underline{\text{if}}\ \text{atom}(x)\ \underline{\text{then}}\ x.\text{NIL}\ \underline{\text{else}}\ F(\text{car}(x)) * F(\text{cdr}(x))$$

and

$$G(x,y) \Leftarrow \underline{\text{if}}\ \text{atom}(x)\ \underline{\text{then}}\ x.y\ \underline{\text{else}}\ G(\text{car}(x),G(\text{cdr}(x),y)) \quad ,$$

where $*$ represents the append function. We shall prove by computation induction that $G(x,y) \equiv F(x)*y$ (over the domain of lists).

Base The equality $UU \equiv UU*y$ is a consequence of the definition of $*$.

Induction If

$$A(x,y) \equiv (\underline{\text{if}}\ \text{atom}(x)\ \underline{\text{then}}\ x.\text{NIL}\ \underline{\text{else}}\ f(\text{car}(x)) * f(\text{cdr}(x))) * y \ ,$$

then

$$A(x,y) \equiv \underline{\text{if}}\ \text{atom}(x)\ \underline{\text{then}}\ (x.\text{NIL})*y\ \underline{\text{else}}\ (f(\text{car}(x)) * f(\text{cdr}(x))) * y$$

$$\equiv \underline{\text{if}}\ \text{atom}(x)\ \underline{\text{then}}\ x.y\ \underline{\text{else}}\ f(\text{car}(x))*(f(\text{cdr}(x))*y) \quad .$$

(LISP axioms)

The conclusion

$$A(x,y) \equiv \underline{\text{if}}\ \text{atom}(x)\ \underline{\text{then}}\ x.y\ \underline{\text{else}}\ g(\text{car}(x),g(\text{cdr}(x),y))$$

follows then by using the induction hypothesis twice. □

2. Modelling Some Proof Techniques Within LCF

Looking back at Chapter 3, we realize that Section 2.3 on termination of programs is the only place where we actually used functions which are not continuous. We therefore have to demonstrate how the technique of structural induction, as described for example, in Burstall [1] or Manna-Ness-Vuillemin [15] can be modelled within LCF.

Finally, a method which was not accounted for in Chapter 3, since its justification requires continuity, is that of Morris [23] and we shall study it in Section 2.2.

2.1 Structural Induction

Actually, the word structural induction covers two rather different techniques. The first one is a simple generalization of the induction principle on natural numbers, while the other one is a statement of Noetherian induction applied to arbitrary well-founded sets, which is the most general induction principle known to man.

Simple Structural Induction

(a) Mathematical Induction

The usual formulation of this principle for natural numbers is:

$$\underline{\text{from}} \quad p(0) \quad \text{and} \quad \forall x(p(x) \Rightarrow p(x+1))$$
$$\underline{\text{infer}} \quad \forall x p(x) \quad .$$

Let the predicate $n(x) \Leftarrow \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ TT\ \underline{\text{else}}\ n(x-1)$ characterize the natural numbers in our system. (We assume the usual axioms about 0 , 1 , $=$, $+$, $-$ as described in Newey [25].) Let $p(x)$ be any predicate which can be expressed as a term of the μ-calculus.

From the premises

$$p(x) \sqsubseteq TT \quad , \quad \underline{if}\ x = 0\ \underline{then}\ TT\ \underline{else}\ p(x-1) \sqsubseteq p(x)$$

we can infer by fixed-point induction that $n(x) \sqsubseteq p(x)$, i.e., that $p(x)$ holds for any natural number x .

In other words,

$$\underline{from}\quad p(0) \equiv TT\ \ \underline{and}\ \ p(x) \equiv TT \vdash p(x+1) \equiv TT$$

$$\underline{infer}\ \ n(x) \equiv TT \vdash p(x) \equiv TT \quad .$$

This method applies to any data-type which is recursively defined by a semi-computable predicate. For example, the domain Σ^* of words over some vocabulary Σ can be characterized by

$$word(x) \Leftarrow \underline{if}\ x = \Lambda\ \underline{then}\ TT\ \underline{else}\ word(t(x))$$

and the corresponding principle is:

$$\underline{from}\quad \underline{if}\ null(x)\ \underline{then}\ p(\Lambda)\ \underline{else}\ p(t(x)) \equiv TT \vdash p(h(x)\cdot t(x)) \equiv TT$$

$$\underline{infer}\ \ word(x) \equiv TT \vdash p(x) \equiv TT \quad .$$

(We are again assuming axioms about Λ , $=$, $\cdot$, h , t .)

Example 4. Let us consider two programs for computing the factorial function:

$$F(x) \Leftarrow \underline{if}\ x = 0\ \underline{then}\ 1\ \underline{else}\ x \times F(x-1)$$

$$G(x,y) \Leftarrow \underline{if}\ x = y\ \underline{then}\ 1\ \underline{else}\ (y+1) \times G(x,y+1) \quad .$$

In order to show that $G(x,0) \equiv F(x)$, we shall prove that $n(x-y) \sqsubseteq p(x,y)$ where $p(x,y)$ is $G(x,y) \times F(y) = F(x)$. Let r be defined as $r(x,y) \Leftarrow \underline{if}\ x = y\ \underline{then}\ TT\ \underline{else}\ r(x,y+1)$.

We first prove that $r(x,y) \equiv n(x-y)$. Then, since

$$p(x,y) \equiv \underline{\text{if}}\ x = y\ \underline{\text{then}}\ F(x) = F(y)\ \underline{\text{else}}\ (y+1)G(x,y+1).F(y) = F(x)$$

$$\equiv \underline{\text{if}}\ x = y\ \underline{\text{then}}\ TT\ \underline{\text{else}}\ p(x,y+1)$$

we can conclude that $r(x,y) \sqsubseteq p(x,y)$, i.e., $n(x-y) \sqsubseteq p(x,y)$. This last inequality is equivalent to $y \leq x \equiv TT \vdash p(x,y) \equiv TT$.

□

This technique required p to be a computable predicate; if P is an arbitrary well-formed-formula, a generalization (Milner [18]) yields:

$$\frac{Q \vdash P\{0/x\} \quad Q,P \vdash P\{(x+1)/x\}}{Q \vdash n(x) \Rightarrow P} \qquad (x \text{ not free in } Q)$$

where $q \Rightarrow s \sqsubseteq t$ means $\underline{\text{if}}\ q\ \underline{\text{then}}\ s\ \underline{\text{else}}\ UU \sqsubseteq \underline{\text{if}}\ q\ \underline{\text{then}}\ t\ \underline{\text{else}}\ UU$, and $q \Rightarrow w_1,w_2$ means $q \Rightarrow w_1$, $q \Rightarrow w_2$.

Example 5. Let

$$rev(x) \Leftarrow F(x,\Lambda)$$

$$F(x,y) \Leftarrow \underline{\text{if}}\ x = \Lambda\ \underline{\text{then}}\ y\ \underline{\text{else}}\ F(t(x),h(x)\cdot y) \quad .$$

In order to show that $rev(rev(x)) \equiv x$, one can prove that $word(x) \Rightarrow P$, where P is $rev(F(x,y)) \equiv F(y,x)$.

□

(b) Course of Values Induction

Another formulation of the induction principle over the natural numbers is the following:

$$\underline{\text{from}}\quad \forall x[\forall y[y < x \Rightarrow p(y)] \Rightarrow p(x)]$$

$$\underline{\text{infer}}\quad \forall x p(x) \quad .$$

Whenever p is computable, this course of value induction can also be modelled directly because the operation of bounded quantification is computable and can be defined as:

$$\forall \equiv \mu f.[\lambda x,p.\ \underline{if}\ x = 0\ \underline{then}\ TT\ \underline{else}\ \underline{if}\ p(x-1)\ \underline{then}\ f(x-1)\ \underline{else}\ UU]$$

According to this definition, $\forall(x,p)$ "means" $\forall y(y < x \Rightarrow p(y))$. We can define the partial predicate $m = \mu p.\lambda x[\forall(x,p)]$ and prove that $m \equiv n$ where $n \equiv \mu f.[\lambda x.\ \underline{if}\ x = 0\ \underline{then}\ TT\ \underline{else}\ f(x-1)]$ as follows.

(i) $\underline{m \sqsubseteq n}$.

$$\begin{aligned}\forall(x,n) &\equiv \underline{if}\ x = 0\ \underline{then}\ TT\ \underline{else}\ \underline{if}\ n(x-1)\ \underline{then}\ \forall(x-1,n)\ \underline{else}\ UU \\ &\sqsubseteq \underline{if}\ x = 0\ \underline{then}\ TT\ \underline{else}\ n(x-1) \\ &\qquad\qquad \text{(by cases using the fact that } \forall(x-1,n) \sqsubseteq TT\text{)} \\ &\equiv n(x)\ .\end{aligned}$$

Hence, $m \sqsubseteq n$ follows by fixed-point induction.

(ii) $\underline{n \sqsubseteq m}$.

Since $x = 0 \equiv FF \vdash m(x-1) \equiv \forall(x-1,m)$ by definition of m , we have $x = 0 \equiv FF \vdash (\underline{if}\ m(x-1)\ \underline{then}\ \forall(x-1,m)\ \underline{else}\ UU) \equiv m(x-1)$ (by cases again, using the fact that $m(x-1) \sqsubseteq TT$). It follows that

$$\begin{aligned}m(x) &\equiv \underline{if}\ x = 0\ \underline{then}\ TT\ \underline{else}\ \underline{if}\ m(x-1)\ \underline{then}\ \forall(x-1,m)\ \underline{else}\ UU \\ &\equiv \underline{if}\ x = 0\ \underline{then}\ TT\ \underline{else}\ m(x-1)\ .\end{aligned}$$

The conclusion $n \sqsubseteq m$ then follows by fixed-point induction again.

□

Having established the equivalence $n \equiv m$, we can justify the following rule of inference:

from $\forall(x,p) \equiv TT \ \vdash \ p(x) \equiv TT$

infer $n(x) \equiv TT \ \vdash \ p(x) \equiv TT$.

A similar rule can be derived for well-formed-formulas.

Example 6. Let us consider a modified version of McCarthy's 91-function:

$$F(x) \Leftarrow \underline{\text{if}}\ x < 0\ \underline{\text{then}}\ x+1\ \underline{\text{else}}\ F(F(x-2)) \quad .$$

In order to prove that $n(x) \equiv TT \vdash (F(x) = 0) \equiv TT$, let $p \equiv \lambda x.[F(x) = 0]$. The equalities $(F(0) = 0) \equiv TT$ and $(F(1) = 0) \equiv TT$ have to be checked first and then, assuming $\forall(x,p) \equiv TT$ and $x > 1 \equiv TT$, we prove $p(x)$:

$$\begin{aligned} p(x) \equiv (F(x) = 0) &\equiv (F(F(x-2)) = 0) && (x < 0 \equiv FF) \\ &\equiv (F(0) = 0) && (p(x-2) \equiv TT) \\ &\equiv TT && (\text{separate check}) \end{aligned}$$

□

Transfinite Induction

Let $\prec$ be a well-founded relation over the domain $\mathcal{D}$. We showed in Chapter 3 how to derive the following principle:

from $\forall x \in \mathcal{D}\{\forall y \in \mathcal{D}[y \prec x \Rightarrow p(y)] \Rightarrow p(x)\}$

infer $\forall x \in \mathcal{D}\{p(x)\}$.

The proof given precluded continuity and is therefore not applicable in the present context.

We shall describe a technique for deriving in LCF any instance of the above rule one may need in "practical" cases. Here, a "practical" well-founded relation is either one of the basic orderings described in the preceding section or an ordering constructed as a well-founded

collection of well-founded relations.[*] Since we already know how to handle the "base" case, all we need to model is the construction of complex orderings from simpler ones.

Let $\underset{1}{<}$ be a computable well-founded relation over the recursive domain $\mathcal{D}_1$, and, for any $x \in \mathcal{D}_1$, let $\underset{x}{<}$ be a well-founded relation over $\mathcal{D}_2(x)$. We then consider the domain $\mathcal{D} = \{(x,y) \mid x \in \mathcal{D}_1 , y \in \mathcal{D}_2(x)\}$ together with the ordering $<$ where $(x,y) < (x',y')$ is equivalent to $x \underset{1}{<} x'$ or $(x = x') \wedge (y \underset{x}{<} y')$. Assuming we already know that the rules

$$(1) \qquad \frac{Q, x' \underset{1}{<} x \Rightarrow P\{x'/x\} \ \vdash \ P}{Q \ \vdash \ \mathcal{D}_1(x) \Rightarrow P} \qquad (x \text{ and } x' \text{ free in } Q)$$

and

$$(2) \qquad \frac{Q, y' \underset{x}{<} y \Rightarrow P\{y'/y\} \ \vdash \ P}{Q \ \vdash \ \mathcal{D}_2(x,y) \Rightarrow P} \qquad (y \text{ and } y' \text{ free in } Q)$$

are valid, we want to justify the rule

$$(3) \qquad \frac{Q, (x',y') < (x,y) \Rightarrow P\{x'/x\}\{y'/y\} \ \vdash \ P}{Q \ \vdash \ \mathcal{D}(x,y) \Rightarrow P} \qquad (x, x', y \text{ and } y' \text{ free in } Q)$$

where $\mathcal{D}(x,y) \equiv \mathcal{D}_1(x) \wedge \mathcal{D}_2(x,y)$. Assuming rules (1) and (2) and the hypothesis of rule (3), we shall prove that $Q \vdash \mathcal{D}_1(x) \wedge \mathcal{D}_2(x,y) \Rightarrow P$ in two nested inductions, by distinguishing between the following cases:

[*] This is equivalent to multiplying the corresponding ordinals. The operation corresponding to ordinal exponentiation can be modelled just as well, although we could never find any practical application for it.

1) $\underline{x' \underset{1}{<} x \equiv TT}$.

The hypothesis of (3) is then $Q, x' \underset{1}{<} x \Rightarrow P\{x'/x\} \vdash P$; hence rule (1) implies that $Q \vdash \mathcal{B}_1(x) \Rightarrow P$ and, a-fortiori, $\mathcal{B}(x,y) \Rightarrow P$.

2) $\underline{x' \underset{1}{<} x \equiv FF}$.

Since $(x,x') < (y,y') \equiv TT$ is the only interesting case, one can assume that $x = x'$ and $y' \underset{x}{<} y$. The hypothesis of (3) then becomes $Q, y' \underset{x}{<} y \Rightarrow P\{y'/y\} \vdash P$ which, by rule (2), implies that $Q \vdash \mathcal{B}_2(x,y) \Rightarrow P$ and the conclusion $Q \vdash \mathcal{B}(x,y) \Rightarrow P$ then follows.

□

Example 7. Using the technique we just described, we shall prove that Ackermann's function

$$A(x,y) \Leftarrow \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ y{+}1\ \underline{\text{else}}$$
$$\underline{\text{if}}\ y = 0\ \underline{\text{then}}\ A(x{-}1,1)\ \underline{\text{else}}\ A(x{-}1,A(x,y{-}1))$$

is defined over the natural number.

Let P be $n(y) \sqsubseteq n(A(x,y))$, where $n \equiv \mu f.[\lambda x.\ \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ TT\ \underline{\text{else}}\ f(x{-}1)]$. We shall prove that $n(x) \vdash P$ which "means" that, whenever x and y are natural numbers, $A(x,y)$ must also be a natural number, is true.

The main proof is by induction on x .

Base: $\underline{x = 0}$. In this case, $P\{0/x\}$ is $n(y) \sqsubseteq n(y{+}1)$ which is always true, as a consequence of the axioms about 0 , 1 and + .

Induction. Assuming $P\{x{-}1/x\}$, that is $n(y) \sqsubseteq n(A(x{-}1,y))$ we must prove P , i.e., $n(y) \sqsubseteq n(A(x,y))$. Let us argue by cases on the predicate $y = 0$:

case $y = 0 \equiv TT$. Since in this case $A(x,y) \equiv A(x-1,1)$, it is sufficient to prove that

$$n(0) \sqsubseteq n(A(x-1,1)) \quad . \qquad (a)$$

We know by the induction hypothesis that $n(1) \sqsubseteq n(A(x-1,1))$ and equation (a) follows, since $n(0) \equiv n(1)$.

case $y = 0 \equiv FF$. Choosing $y = A(x,y-1)$ in the induction hypothesis $P\{x-1/x\}$ gives us:

$$n(A(x,y-1)) \sqsubseteq n(A(x-1,A(x,y-1))) \quad .$$

Since in this case $A(x,y) \equiv A(x-1,A(x,y-1))$ the last inequality implies that $n(A(x,y-1)) \sqsubseteq n(A(x,y))$. Hence, by a "nested" fixed-point induction applied to the predicate $q(y) \equiv n(A(x,y))$ we conclude that $n(y) \sqsubseteq n(A(x,y))$.

□

2.2 Truncation Induction

Recalling Kleene's first recursion theorem, we can characterize the least fixed-point of the program $F \Leftarrow \tau(F)$ as the least upper bound of the sequence of functions $f_0, f_1, \ldots, f_n, \ldots$ defined by $f_0 \equiv UU$ and $f_{n+1} \equiv \tau(f_n)$. The rule of truncation induction, as Morris [23] named it, can be formulated as

Rule TI

from $Q \vdash P\{f_n/f\}$ for any natural number n

infer $Q \vdash P\{f_\tau/f\}$.

Actually Morris [23] used the formulation

from $Q, \forall m(m < n \Rightarrow P\{f_m/f\}) \vdash P\{f_n/f\}$

infer $Q \vdash P\{f_n/f\}$ for all n

which is equivalent to ours since Section 2.1 of this chapter shows how to obtain the missing step, namely:

$$\underline{\text{from}}\quad Q, \forall m(m < n \Rightarrow P\{f_m/f\} \vdash P\{f_n/f\} \quad \text{for all } n$$

$$\underline{\text{infer}}\quad Q \vdash P\{f_n/f\} \quad \text{for all } n \,.$$

A first problem which arises with rule TI is that, since it requires knowledge about the integers in its formulation, it cannot even be *expressed* in pure LCF. (This should be regarded as an advantage of Scott's formulation of the rule.)

More dramatic is the fact that, even in an LCF with integers (where TI can then be expressed), there does not seem to be any way to justify it, despite the fact that it is clearly valid in any standard model. It is possible to get around this difficulty by slightly extending the logic. What is needed is a formal way to talk about limits. This can be achieved by embedding data-types into complete lattices, thus going back to the original definition of data-types in Scott [29]. This idea entails the following extensions to LCF:

(1) Introduce constant terms 00 (for overdefined) at each type. The corresponding axioms are $\vdash x \sqsubseteq 00$ and $\vdash 00 \sqsubseteq 00(x)$. In the case-rule, the case $P\{00/x\} \vdash Q$ should be added to the premise.

(2) If s and t are terms of type α , then $\underline{\text{sup}}(s,t)$ should also be a term of type α . It is axiomatized by $\vdash x \sqsubseteq \underline{\text{sup}}(x,y)$, $\vdash y \sqsubseteq \underline{\text{sup}}(x,y)$ and $x \sqsubseteq z, y \sqsubseteq z \vdash \underline{\text{sup}}(x,y) \sqsubseteq z$.

(3) We could introduce $\underline{\text{inf}}(x,y)$ in the same way, although we won't need it. Also, one should make up his mind as to what $\underline{\text{if}}$ 00 $\underline{\text{then}}$ x $\underline{\text{else}}$ y ought to mean. Two extreme possibilities are $\vdash \underline{\text{if}}\ 00\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y \equiv 00$ or $\vdash \underline{\text{if}}\ 00\ \underline{\text{then}}\ x\ \underline{\text{else}}\ y \equiv \underline{\text{sup}}(x,y)$.

In this extended logic (along with the natural numbers) we can then justify rule TI:

First of all, one needs to express the rule within the formal system, and we shall define $f_n \equiv \tau^n(UU)$ as $iter(\tau)(n)$ where

Definition 1.

$$iter \equiv \mu f.[\lambda\tau,n.\ \underline{if}\ n = 0\ \underline{then}\ UU\ \underline{else}\ \tau(f(n-1))]$$

Using this definition, it is easy to prove that

Lemma 1.

$$iter(\tau)(n) \sqsubseteq iter(\tau)(n+1)$$

and

Lemma 2.

$$iter(\tau)(n) \sqsubseteq f_\tau \quad .$$

We now wish to prove that $f_\tau \equiv \bigsqcup_{n \geq 0} \{f_n\}$ and, for this purpose, let

Definition 2.

$$\bigsqcup \equiv \mu f.[\lambda\beta,n.\ sup(\beta(n),f(\beta(n+1)))] \quad .$$

Using an induction on this formal definition of $\bigsqcup$, one can then prove that

Lemma 3.

$$\beta(n) \sqsubseteq g \quad \vdash \quad \bigsqcup(\beta,n) \sqsubseteq g$$

and

Lemma 4.

$$\beta(n) \sqsubseteq \beta(n+1) \quad \vdash \quad \gamma(\bigsqcup(\beta,n)) \equiv \bigsqcup(\lambda x.\gamma(\beta(x)),n) \quad .$$

Note that Lemma 4 is particularly interesting since it proves that any function γ which can be expressed within the logic must be continuous. Kleene's first recursion theorem may now be expressed as

$$f_\tau \equiv \sqcup(\text{iter}(\tau),n) \qquad (K)$$

and proved in two steps.

Firstly, combining Lemmas 2 and 3 yields

$$\underline{\sqcup(\text{iter}(\tau),n) \sqsubseteq f_\tau} \quad .$$

Then, the other half of the proof is a little bit more complicated.

$$\begin{aligned} \tau(\sqcup(\text{iter}(\tau),n)) &\equiv \sqcup(\lambda x.\tau(\text{iter}(\tau)(x)),n) && \text{(Lemmas 1 and 4)} \\ &\equiv \sqcup(\lambda x.\ \text{iter}(\tau)(x+1),n) && \text{(Definition 1)} \\ &\sqsubseteq \sqcup(\text{iter}(\tau),n) \quad . && \text{(Lemma 1)} \end{aligned}$$

The conclusion

$$\underline{f_\tau \sqsubseteq \sqcup(\text{iter}(\tau),n)}$$

follows by fixed-point induction.

We now have all the machinery required for justifying truncation induction. Assuming for simplicity that the well-formed-formula we want to use is of the form $\alpha(f) \sqsubseteq g$, we must prove that

$$\alpha(\text{iter}(\tau)(n)) \sqsubseteq g \ \vdash \ \alpha(f_\tau) \sqsubseteq g \quad .$$

Lemmas 1 and 4 tell us that

$$\sqcup(\lambda x.\alpha(\text{iter}(\tau)(x)),n) \equiv \alpha(\sqcup(\text{iter}(\tau),n)) \quad ,$$

and therefore

$$\alpha(\text{iter}(\tau)(n)) \sqsubseteq g \ \vdash \ \alpha(\sqcup(\text{iter}(\tau),n)) \sqsubseteq g \quad .$$

Since $f_\tau \equiv \sqcup(\text{iter}(\tau),n)$ by Kleene's theorem, this last implication reduces to

$$\alpha(\text{iter}(\tau)(n)) \sqsubseteq g \;\vdash\; \alpha(f_\tau) \sqsubseteq g$$

which is what we wanted to prove.

Applications

-- First of all, some equivalence proofs seem to be more natural (and may in fact require) using truncation induction.

For example, if two functionals s and t satisfy $s(UU) \equiv t(UU)$ and $st \equiv t^2 s$,*/ the natural truncation induction predicate would be $t^{2^n-1}(UU) \equiv s^n(UU)$, and therefore $\mu f.s(f) \equiv \mu f.t(f)$. If one uses the machinery we just developed, this informal proof can very easily be carried through within the extended logic. Actually, a more elegant proof (not using natural numbers) would be the following:

Define

$$M(g,f)(x) \Leftarrow \underline{\text{sup}}(f(x),M(g,f)(f(x)))$$

and

$$N(g,f)(x) \Leftarrow \underline{\text{sup}}(f(x),N(\lambda x.g(g(x)),f)(g(x))) \quad .$$

($M(s,\lambda f.f)(UU)$ represents $\bigsqcup_{n \geq 0} s^n(UU)$ and $N(t,\lambda f.f)(UU)$ represents $\bigsqcup_{n \geq 0} t^{2^n-1}(UU)$.) One can then prove that $f_s \equiv M(s,\lambda f.f)(UU)$ and $f_t \equiv N(t,\lambda f.f)(UU)$ and finally that

$$s(UU) \equiv t(UU)\, ,\, \lambda f.s(t(f)) \equiv \lambda f.t(t(s(f))) \;\vdash\; M(s,\lambda f.f)(UU) \equiv N(s,\lambda f.f)(UU) \quad .$$

---

*/ This example is due to J. W. deBakker. Robin Milner has a proof of it in pure LCF. The reader may find out for himself how tricky it is, and further away from the intuitive proof than the one presented here.

-- Similarly, let us consider the following version of the induction rule

<u>rule R6'</u>

$$\frac{Q \vdash h \sqsubseteq f_\tau, P\{h/f\} \quad Q, P \vdash P\{\tau(f)/f\}}{Q \vdash P\{f_\tau/f\}} \qquad (f \text{ not free in } Q)$$

where the base of computation induction is not taken at the undefined element UU but at any element $h \sqsubseteq f_\tau$.

Informally and assuming P to be $\alpha(f) \sqsubseteq \beta(f)$ for simplicity, the hypothesis of the rule implies that $\alpha(\tau^n(h)) \sqsubseteq \beta(\tau^n(h))$ for any n . On the other hand, $UU \sqsubseteq h \sqsubseteq f_\tau$ implies $\tau^n(UU) \sqsubseteq \tau^n(h) \sqsubseteq f_\tau$ and therefore $\bigsqcup_{n \geq 0} \{\tau^n(h)\} \equiv f_\tau$. The conclusion $\alpha(f_\tau) \sqsubseteq \beta(f_\tau)$ then follows easily from the continuity of α and monotonicity of β .

This argument can be carried through formally within the extended LCF. In particular, it applies to the following theorem

$$\vdash f_\tau \sqsubseteq f_\sigma$$

$$\frac{\tau(f) \sqsubseteq f \vdash \tau(\sigma(f)) \sqsubseteq \sigma(f)}{\vdash \tau(f_\sigma) \sqsubseteq f_\sigma}$$

which is provable in the extended logic; the author does not know how to prove it (and conjectures are not provable) in pure LCF.

Conclusion

In the actual state-of-the-art, Scott's approach to the semantics of programming languages seems to be the most promising one. The theoretical foundations are sound, and a natural step would now be to describe fully the semantics of a full-size programming language, along the lines of Scott-Strachey [32], Milner-Weyrauch [21], or Reynolds [27].

Another wide open and promising area seems to be that of semantics of operating-systems and parallel processes. Steps in this direction were taken by Kahn [11], Milner [20], and others.

Finally, the question of a "best" logic for expressing a theory of computation remains. As alternatives to LCF, the systems of Hitchcock-Park [8] and deBakker - deRoever [5] have some interesting features; in an unpublished work, Scott and Milner also considered the possibility of extending LCF to a "type-free" logic whose semantic domain is one of Scott's models of the λ-calculus.

In any case, more efforts should be put in studying the existing systems. In particular, LCF provides a nice framework for the area of schematology, where existing results can be expressed and sometimes simplified, and where new and interesting questions arise. (See deBakker [4] and Courcelles-Kahn-Vuillemin [3].)

References

[1] R. M. Burstall, "Proving Properties of Programs by Structural Induction," Computer Journal, Vol. 12, (1969), 41-48.

[2] J. M. Cadiou, "Recursive Definitions of Partial Functions and Their Computations," Ph.D. Thesis, Computer Science Department, Stanford University, (1972).

[3] B. Courcelles, G. Kahn, and J. Vuillemin, "Algorithmes d'Equivalence pour des Equations Récursives Simples," Rapport LABORIA, IRIA, 78-Rocquencourt, France, (1973).

[4] J. W. deBakker, "Recursive Procedures," Mathematical Centre Tracks 24, Amsterdam, (1971).

[5] J. W. deBakker and W. P. deRoever, "A Calculus for Recursive Program Schemes," Proceedings of IRIA Colloquium, North-Holland, (1972).

[6] J. W. deBakker and D. Scott, "A Theory of Programs," Unpublished memo, (1969).

[7] R. W. Floyd, "Assigning Meanings to Programs," Proceedings of a Symposia in Applied Mathematics, Vol. 19, American Mathematical Society, (1967), 19-32.

[8] P. Hitchcock and D. Park, "Induction Rules and Proofs of Termination," Proceedings of IRIA Colloquium, North-Holland, (1972).

[9] C. A. R. Hoare, "Procedures and Parameters: an Axiomatic Approach," Symposium on Semantics of Algorithmic Languages, Vol. 188, Springer-Verlag, (1971), 102-116.

[10] Y. I. Ianov, "The Logical Scheme of Algorithms," Problems of Cybernetics, Vol. 1, Pergamon Press, (1960), 82-140.

[11] G. Kahn, "A Preliminary Theory of Parallel Programs," Rapport LABORIA, IRIA, 78-Rocquencourt, France, (1973).

[12] W. Lonergan and P. King, "Design of the B5000 System," Datamation, Vol. 7, No. 5, (May 1961), 28-32.

[13] J. McCarthy, "A Basis for a Mathematical Theory of Computation," Computer Programming and Formal Systems, (Eds., P. Braffort and D. Hirshberg), North-Holland, (1963), 33-70.

[14] Z. Manna, "The Correctness of Programs," Journal of Computer and System Sciences, Vol. 3, No. 3, (1969), 119-127.

[15] Z. Manna, S. Ness, and J. Vuillemin, "Inductive Methods for Proving Properties of Programs," Proceedings ACM Conference, ACM, New York, (1972).

[16] Z. Manna and A. Pnueli, "Formalization of Properties of Functional Programs," J.ACM, Vol. 17, No. 3, (1970), 555-569.

[17] Z. Manna and J. Vuillemin, "Fixpoint Approach to the Theory of Computation," C.ACM, Vol. 15, No. 7, (1972), 528-536.

[18] R. Milner, "Implementation and Applications of Scott's Logic for Computable Functions," Proceedings ACM Conference, ACM, New York, (1972).

[19] R. Milner, "Models of LCF," AIM-186/CS-332, Computer Science Department, Stanford University, (1973).

[20] R. Milner, "An Approach to the Semantics of Parallel Programs," Edinburgh Tech. Memo, University of Edinburgh, (1973).

[21] R. Milner and R. Weyrauch, "Proving Compiler Correctness in a Mechanized Logic," Machine Intelligence 7, Edinburgh University Press, (1972).

[22] J. H. Morris, "Lambda-Calculus Models of Programming Languages," Report MAC-TR-57, Mass. Inst. of Technology, (1968).

[23] J. H. Morris, "Another Recursion Induction Principle," C.ACM, Vol. 14, No. 5, (1971), 351-354.

[24] P. Naur, "Proof of Algorithms by General Snapshots," BIT, Vol. 6, (1966), 310-316.

[25] M. Newey, Ph.D. Thesis, Computer Science Department, Stanford University, (to appear).

[26] D. Park, "Fixpoint Induction and Proofs of Program Properties," Machine Intelligence 5, Edinburgh University Press, (1969), 59-78.

[27] J. C. Reynolds, "Definitional Interpreters for Higher Order Programming Languages," Proceedings ACM Conference, ACM, New York, (1972).

[28] B. K. Rosen, "Tree-Manipulating Systems and Church-Rosser Theorems," J.ACM, Vol. 20, No. 1, (1973), 160-187.

[29] D. Scott, "Outline of a Mathematical Theory of Computation," Oxford Mono. PRG-2, Oxford University, (1970).

[30] D. Scott, Unpublished paper.

[31] D. Scott, "Continuous Lattices," Oxford Mono. PRG-7, Oxford University, (1972).

[32] D. Scott and C. Strachey, "Toward a Mathematical Semantics for Computer Languages," Oxford Mono. PRG-6, Oxford University, (1972).