Evaluation and denotation of pure LISP programs: a worked example in semantics

Michael J.C. Gordon

Ph.D. Thesis
University of Edinburgh
1973

ABSTRACT

A Scott/Strachey style denotational semantics intended to describe pure LISP is examined. I present evidence that it is an accurate rendering of the language described in chapter 1 of the LISP 1.5 Programmer's Manual, in particular I show that call-by-value and fluid variables are correctly handled. To do this I have:

(1) Written an operational 'semantics' of pure LISP and shown it equivalent to the denotational one

(2) Proved that, relative to the denotational semantics, the LISP functions apply,eval,...,etc. correctly compute meanings.

The proof techniques used are derived from the work of Wadsworth; roughly one first proves the results for a class of 'finite' programs and then extends them to all programs by a limiting argument. Conceptually these arguments are inductions on length of computation and to bring this out I've formulated a rule of inference which enables such operational reasoning to be applied to the denotational semantics.

CONTENTS

Abstract

Acknowledgements

Contents

1. Introduction
    - 1.1 Semantics: how this work fits in
    - 1.2 Overview
    - 1.3 Historical perspective
    - 1.4 Contents of this report and how to read it

2. The syntax and evaluation of pure LISP programs: a resume of the Manual
    - 2.1 Syntax
    - 2.2 Standard functions
    - 2.3 The translation of M-expressions into S-expressions
    - 2.4 The Manual Interpreter: evalquote,apply,eval,evcon,evlis

3. Denotational semantics of pure LISP
    - 3.1 Denotational intuition and its uses
    - 3.2 Some intuitions about LISP
    - 3.3 Introduction to the Scott and Strachey approach to semantics
    - 3.4 Scott's axioms: preliminary motivation
    - 3.5 Some definitions needed in order to state Scott's axioms
    - 3.6 Statement of Scott's axioms
    - 3.7 Some more definitions and notation
    - 3.8 An example: a naive denotational semantics of LISP
    - 3.9 What can be done about the wrong denotations assigned by the naive semantics?
    - 3.10 Simultaneous recursions: a defect of the language described in 3.8.2
    - 3.11 'Debugging' the semantics of 3.8
    - 3.12 How interpreters do recursion
    - 3.13 Syntax of pure LISP
    - 3.14 New improved semantics of pure LISP
    - 3.15/

3.15 Example calculation of denotations using the new improved semantics

3.16 Why I have assigned the same meaning to errors and non-termination

4. Is the semantics right?

4.1 Introduction

4.2 A formalization of some operational intuitions about LISP

4.3 Statement of Main Theorem connecting operational and denotational semantics

4.4 LISP-induction: an inference rule for applying the Main Theorem

4.5 Proof of half of the Main Theorem: an example of LISP-induction

4.6 The relation between μ and Label

4.7 A proposition about variables

4.8 A semantic analysis of the LISP eval function

5. Proof of the Main Theorem

5.1 Outline of the essential idea

5.2 Further specification of Env

5.3 Extended LISP (ELISP) and extended ALIST (EALIST)

5.4 The extension of -> to ELISP and EALIST

5.5 Approximants and the relation $<$

5.6 Lemma

5.7 Some terminology

5.8 Proof that all finite terms terminate

5.9 Terms which lead to an error denote $\perp$

5.10 Final step in the proof of the Main Theorem

6. Universal space U

6.1 Solving semi-domain equations

6.2 Specification of the universal space U used here

6.3 Existence of universal spaces

6.4 Properties of universal spaces

6.5 Uniqueness of universal space

6.6/

6.6 Representing things in universal space

6.7 Representing Env in universal space

6.8 BNF as recursion equations in universal space

7. Extensions of the semantics to more of LISP 1.5

7.1 Functional arguments (funargs)

7.2 Functional results

7.3 QUOTE'd arguments

7.4 Further extensions

8. Suggestions for future research

8.1 Analysis of more sophisticated semantic equations

8.2 Fluid variables at higher types: a calculus

8.3 The true relation between μ and Label

8.4 Axioms to disambiguate semi-domain equations

8.5 Rules of inference for LISP: LISP logic

8.6 Fixpoint treatment of simultaneous recursive definitions

8.7 When does the naive semantics work?

Appendix 1

References

1. INTRODUCTION

1.1 Semantics: how this work fits in

The approach to the semantics of programming languages developed by Scott and Strachey [22] has been remarkably successful. Complete descriptions of PAL [9], ALGOL 60 [14] and ALGOL 68 [9] have been given and it would now seem to be routine to write a formal definition of any language of this level of complexity. Milner's theory of processes [11] gives a very satisfactory explication of non-determinism and parallel processing and there is a good hope that it will considerably simplify the formal study of operating systems and similar complexes of interacting programs. Although the Scott/Strachey approach appears to give us an accurate way of expressing the intuitive meaning of programming languages it is very abstract and there is a large gap between the sophisticated mathematical devices it exploits and concrete computational mechanisms. Thus it is possible (though not likely, I hope) that some of these formal definitions might have properties which we would not expect e.g. they might assign meanings which differ subtly from what is intended. To reduce this danger it helps to investigate in detail the way abstract devices mimic concrete ones. This kind of investigation has a double use for as well as boosting our confidence in the abstract techniques it should also shed light on how to go about proving implementations of (abstractly defined) languages correct. The work described here is such an investigation.

I/

I have taken a simple <u>real</u> programming language - pure LISP* - and analysed in depth the relation between the possible abstract denotations of its programs and their mechanical evaluation. I have paid particular attention to the correct handling of call-by-value and fluid variables and to illustrate the dangers of a too naive approach to these features I describe a superficially plausible (but in fact wrong) semantics.

1.2 <u>Overview</u>

If you read the 'official' definition of pure LISP (chapter 1 of the LISP 1.5 Programmer's Manual) I think you will find that two distinct ways of thinking suggest themselves:

(1) LISP functions can be thought of as <u>denoting</u> mappings from S-expressions to S-expressions - so that e.g.

$$\lambda\,[[x];\ car[cdr[x]]]$$

denotes the composition of the cdr function with the car

---

* Pure LISP is the language described in chapter 1 of the LISP 1.5 Programmer's Manual. It consists of LISP minus all frills such as the Prog feature, arrays, list structure operators (rplaca), functional arguments, etc. A very brief summary (<u>not</u> intended as an introduction) is given in chapter 2 below.

function and so to work out the value of $\lambda[[x];\ car[cdr[x]]][(1\ 2)]$ one just applies this function to (1 2).

(2) LISP functions can be thought of as <u>rules</u> which specify how their arguments are to be manipulated to <u>compute</u> the result. In this way of thinking $\lambda[[x];\ car[cdr[x]]]$ is the rule which when given an argument <u>first</u> binds it to x in the environment <u>then</u> evaluates car[cdr[x]] - this consists in first evaluating x then taking the cdr and then the car of the result. This explanation does not invoke 'mathematical' notions such as application and composition - only sequences of finitistic operations are called for.

I shall call (1), (2) the "denotational" and "operational" conceptions of LISP respectively. In the manual both ways of thinking are alluded to; the informal descriptions are on the whole denotational whereas the formal definition expressed in the interpreter (i.e. the functions apply, eval,... etc.) is usually understood operationally.

In this report I describe some formal tools for translating operational intuitions into denotational terms and vice versa. More precisely what I do is:

(1) Give a Scott/Strachey style denotational semantics of pure LISP.

(2) Give, using a calculus, an operational 'semantics' of it.

(3)/

(3) Prove that the mapping denoted by any LISP function relative to the denotational semantics is correctly computed by the operations of the operational one.

Having done this I then exploit operational reasoning to infer facts about the denotational semantics. To assist in this I have formulated a rule of inference called "LISP-induction" - in operational terms this can be thought of as induction on the length of computations but it can be used to prove things about the denotational semantics - and associated with this there are no obvious computations to do induction on! Using LISP-induction I show:

(4) How recursive definitions (i.e. label expressions) can be understood in terms of minimal fixed points.

(5) That the mappings <u>denoted</u> by the functions apply,eval,...etc. constitute a correct (relative to the denotational semantics) implementation of LISP.

I chose LISP as a subject for this study because it is a widely used <u>real</u> language which is sufficiently simple to raise mathematically tractable problems. I chose a real language - rather than a theoretical one like the λ-calculus - because I wanted to be sure that the kind of problems which arise in practice would come to light. I also needed to have a language in which all the programs have a precise and unambiguous meaning for I needed to have a rigorous standard against/

against which to measure proposed denotational semantics.

1.3 Historical perspective

As far as I know not much work has been done connecting the kind of mathematical model described here with computation mechanisms. A fair amount has been done on the simulation of one machine by another machine or by a calculus (see e.g. [7]) but this is a rather different topic in which syntactic entities are not compared with conventional mathematical objects but only with other syntactic things.

The earliest relevant work seems to have been done by Kleene who showed how recursive definitions of numerical functions could be understood as defining the least fixed point of certain equations (derived naturally from the definitions). Kleene also showed how such recursive definitions could be used to effectively compute the application of the defined functions to their arguments. This work is described, from a computing point of view, in Morris's thesis [12]; Cadiou [2], de Bakker and de Roever [1] and others have extended it to deal with the kinds of recursive definitions, and associated evaluation rules, met in actual computing practice.

Although this work is extremely good and important the techniques it employs are of somewhat limited applicability and it was not until Wadsworth [25] came along that general methods of proving implementations of whole languages correct became available. Wadsworth shows how operational/

operational facts about the pure λ-calculus are reflected in the semantics. Although the λ-calculus is not a real programming language it turns out to be easy to apply, by analogy, Wadsworth's methods to real situations. Plotkin [16] has done this for ISWIM [5], a λ-calculus like language which uses a call-by-value evaluation strategy and has "basic" functions (δ-rules).

The importance of Wadsworth's methods is that they enable one to introduce a notion of 'progress' of computation into denotational semantics. This notion leads to ways of proving things by induction on the 'length' of computation - a kind of induction which is very intuitive but not easy to precisely formulate and validate relative to denotational semantics. Almost all realistic denotational semantics involve infinite-type spaces, even if (as in pure LISP) the object language is first order; Wadsworth's methods are applicable to such semantics and it is because of this that these methods are so much more powerful than the older ones.

The only other relevant research I know is that of Robert Milne of Oxford. This remarkable work (which I only heard of during the writing of this report), although rather more abstract and differently motivated from mine, promises to provide tools considerably more elegant and general than those described here. Unfortunately, I have not fully digested that work and so am unable to give a reliable description of it and its relation to what is done here. If you are interested I suggest you/

you (like me) await his forthcoming thesis [8].

1.4 Contents of this report and how to read it

I hope this report will be comprehensible to readers not familiar with Scott's approach to the theory of computation [20] and Scott and Strachey's approach to semantics [22]. To this end I have included some introductory material of a rather elementary nature in chapters 3 and 6; sophisticated readers are advised to skim through this at high speed. Chapter 2 is a very condensed summary of the contents of chapter 1 of the LISP 1.5 Programmer's Manual. I strongly advise readers not familiar with LISP to read chapter 1 of the Manual. Chapter 3 is devoted to describing how the denotational conception of LISP can be formalized. In chapter 4 I formalize the operational conception and also state my main theorem connecting these two conceptions and show how to exploit it. In chapter 5 I prove the main theorem - subject to the existence of a certain infinite-type space, (which is postulated as a model of alists) and in chapter 6 I develop enough tools to prove that this space exists. Chapter 7 contains denotational semantics of some extensions of pure LISP to show that I'm not yet at a dead end and in chapter 8 I suggest some (rather vague) topics for future research. Appendix 1 is of a technical nature, in it I compare the theory of semi-domains used here with the more usual theory of domains.

2. THE SYNTAX AND EVALUATION OF PURE LISP PROGRAMS: A RESUME OF THE MANUAL

In this chapter I state the syntax of pure LISP and give the definitions of the functions constituting the interpreter described in the Manual. This chapter is <u>not</u> meant to be an exposition of LISP-readers unfamiliar with the la guage are strongly advised to read chapter 1 of the Manual [6] before proceeding. I give the syntax and interpreter here for reference purposes and so that readers can check up that my formalizations are accurate (e.g. that I have not surreptitiously simplified things).

2.1 <u>Syntax</u>

2.1.1 <u>The Data Language</u>

```
<LETTER> ::= A | B | C | ... | Z
<number> ::= 0 | 1 | 2 | ... | 9
<atomic-symbol> ::= <LETTER><atom part>
<atom part> ::= <empty> | <LETTER><atom part> | <number><atom part>

<S-expression> ::= <atomic symbol> |
                   (<S-expression>.<S-expression>) |
                   (<S-expression>...<S-expression>)
```

2.1.2 <u>The Meta Language (M-expressions)</u>

```
<letter> ::= a | b | c | ... | z
<identifier> ::= <letter><id part>
<id part> ::= <empty> | <letter><id part> | <number><id part>
```

```
<form> ::= <constant> |
           <variable> |
           <function>[<argument>;...;<argument>] |
           [<form>→ <form>;...;<form>→ <form>]
<constant> ::= <S-expression>
<variable> ::= <identifier>
<argument> ::= <form>

<function> ::= <identifier> |
               λ[<var list>; <form>] |
               Label[<identifier>;<function>]
<var list> ::= [<variable>;...;<variable>]
```

2.2 Standard Functions

The following functions are needed to define the interpreter

2.2.1 caar,cdar,cadr,caddr,cadar

```
 caar[x]=car[car[x]]
 cdar[x]=cdr[car[x]]
 cadr[x]=car[cdr[x]]
caddr[x]=car[cdr[cdr[x]]]
cadar[x]=car[cdr[car[x]]]
```

2.2.2 null

$$\text{null}[x]=\begin{cases}\text{T} & \text{If } x=\text{NIL}\\ \text{F} & \text{otherwise}\end{cases}$$

2.2.3 equal

In the Manual equal is defined by:

```
equal[x;y]=[atom[x]→ [atom[y]→ eq[x;y];T→ F];
            equal[car[x];car[y]]→ equal[cdr[x];cdr[y]];
             T→ F]
```

It is claimed that equal, so defined, "is true if its two arguments are identical S-expressions and is false if they are different". Unfortunately this claim is false for, using the above definition, equal[(1);1] is undefined not F. I have taken the description quoted above (in English) as the intended meaning - this is formalized in 4.8.1.11 below.

2.2.4 pairlis

```
pairlis[x;y;a]=[null[x]→ a; T→ cons[cons[car[x];car[y]];
                               pairlis[cdr[x];cdr[y];a]]]
```

```
Example:  pairlis[(A B C);(U V W); ((D.X)(E.Y))]
              =((A.U)(B.V)(C.W)(D.X)(E.Y))
```

2.2.5 assoc

```
assoc[x;a]=[equal[caar[a];x]→ car[a];T→ assoc[x;cdr[a]]]
```

```
Example:  assoc[B;((A.(M N))(B.(CAR X))(C.(QUOTE M))(C.(CDR X)))]
          =(B.(CAR X))
```

2.3/

2.3 The Translation of M-expressions into S-expressions

The following rules define a method of translating functions written in the meta-language into S-expressions.

1. If a function is represented by its name, it is translated by changing all of the letters to upper case, making it an atomic symbol. Thus car is translated to CAR.

2. If the function uses the lambda notation, then the expression $\lambda[[x_1;\ldots;x_n];\ \epsilon]$ is translated into $(\mathrm{LAMBDA}(X_1\ldots X_n)\ \epsilon^*)$, where ϵ^* is the translation of ϵ.

3. If the function begins with Label, then the translation of Label[a; ϵ] is (LABEL a* ϵ^*).

Forms are translated as follows:

1. A variable, like a function name, is translated by using uppercase letters. Thus the translation of var1 is VAR1.

2. The obvious translation of letting a constant translate into itself will not work. Since the translation of x is X, the translation of X must be something different to avoid ambiguity. The solution is to quote it. Thus X is translated to (QUOTE X).

3. The form $\mathrm{fn}[\mathrm{arg}_1;\ldots;\mathrm{arg}_n]$ is translated into $(\mathrm{fn}^*\ \mathrm{arg}_1^*\ldots\mathrm{arg}_n^*)$.

4./

4. The conditional expression $[p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n]$ is translated into (COND($p_1^* e_1^*$)...($p_n^* e_n^*$)).

Examples

M-expressions	S-expressions
x	X
car	CAR
car[x]	(CAR X)
T	(QUOTE T)
ff[car[x]]	(FF((CAR X))
[atom[x]→ x;T→ ff[car[x]]]	(COND((ATOM X)X) ((QUOTE T)(FF(CAR X)))
Label[ff; λ[[x];[atom[x]→ x;T→ ff[car[x]]]]]	(LABEL FF(LAMBDA(X)(COND ((ATOM X)X) ((QUOTE T)(FF(CAR X))))))

2.4 The Manual Interpreter: evalquote,apply,eval,evcon,evlis

It says in the Manual that:

"The universal function evalquote that is about to be defined obeys the following identity. Let f be a function written as an M-expression, and let fn be its translation. (fn is an S-expression.) Let f be a function of n arguments and let args=(arg_1...arg_n),a list of the n S-expressions being used as arguments/

arguments. Then

evalquote[fn;args]=f[arg_1;...;arg_n]

if either side of the equation is defined at all."

I suspect that this statement was intended to connect up the operational and denotational conceptions of LISP i.e. the left hand side of the equation was intended to be understood operationally and the righ hand side denotationally. If this is so then 4.8 of this report ("A semantic analysis of the LISP eval function") is devoted to <u>proving</u> the above statement. Here now is the definition of evalquote:

```
evalquote[fn;x]=apply[fn;x;NIL]

apply[fn;x;a]=[atom[fn]→ [eq[fn;CAR]→ caar[x];
                          eq[fn;CDR]→ cdar[x];
                          eq[fn;CONS]→ cons[car[x];cadr[x]];
                          eq[fn;ATOM]→ atom[car[x]];
                          eq[fn;EQ]→ eq[car[x];cadr[x]];
                               T   → apply[eval[fn;a];x;a]];
               eq[car[fn];LAMBDA]→ eval[caddr[fn];pairlis[cadr[fn];x;a]];
               eq[car[fn];LABEL]→ apply[caddr[fn];x;cons[cons[cadr[fn];
                                                         caddr[fn]];a]]]
```

eval/

```
eval[e;a]=[atom[e]→ cdr[assoc[e;a]];
            atom[car[e]]→ [eq[car[e];QUOTE]→ cadr[e];
                           eq[car[e];COND]→ evcon[cdr[e];a];
                             T→ apply[car[e];evlis[cdr[e];a];a]];
                  T→ apply[car[e];evlis[cdr[e];a];a]]

evcon[c;a]=[eval[caar[c];a]→ eval[cadar[c];a];
                  T→ evcon[cdr[c];a]]

evlis[m;a]=[null[m]→ NIL;
                  T→ cons[eval[car[m];a];evlis[cdr[m];a]]]
```

3. DENOTATIONAL SEMANTICS OF PURE LISP

The main goal of this chapter is to describe a <u>denotational</u> semantics of pure LISP. I shall do this by showing how I evolved a satisfactory (though, at first sight, non-intuitive) semantics from an unsatisfactory (though intuitive) one.

First I shall give some idea of what I am trying to capture and why doing this is useful.

3.1 <u>Denotational intuition and its uses</u>

Naive intuitions derived from a mathematical upbringing are not always a good guide to the meaning of LISP functions. For example it is tempting to think of λ[[x];NIL] as denoting the constant function with value NIL and hence to deduce that for all forms e:

$$\lambda[[x];\mathrm{NIL}][e]=\mathrm{NIL}$$

but this is wrong for if e's evaluation does not terminate then nor does that of λ[[x];NIL][e] and so this does not evaluate to NIL. This is not the only way that 'mathematical' intuition can mislead: LISP's fluid variables also cause trouble: the problem arises when one passes functions containing free variables into contexts which bind (and hence 'capture') them. This is most simply illustrated using functional arguments but as I only wish to study first order pure LISP (i.e. what is described in chapter 1 of the Manual) I will not give such an example. I am going to give a form e and then describe/

describe two ways of working out its value; the first way (given in 3.1.1 below) will exploit reasoning dear to mathematically trained people, the second (given in 3.1.2) will use the LISP interpreter given in chapter 1 of the Manual. Here is the e. It is a bit complicated - do not try and apprehend it directly but go on and read 3.1.1 and 3.1.2.

```
e= λ[[y];
     Label[fn; λ[[x];
                 [y→ 1;
                  x→ 2;
                  T→ λ[[y];fn[y]][T]]]][F]][F]
```

If we let: $fn_1 = \lambda[[y];fn[y]]$

$e_1 = fn_1[T]$

$e_2 = [y\to 1;x\to 2;T\to e_1]$

$fn_2 = \lambda[[x];e_2]$

Then: $e = \lambda[[y];Label[fn;fn_2][F]][F]$

Here now are two chains of reasoning leading to values for e.

<u>3.1.1</u>

The value of e is the value of Label[fn;fn_2][F] when y=F, now Label[fn;fn_2] denotes the function defined recursively by:

fn/

$fn[x]=[y\rightarrow 1;x\rightarrow 2;T\rightarrow fn_1[T]]$

when y=F this reduces to

$fn[x]=[x\rightarrow 2;T\rightarrow fn_1[T]]$

and so $fn[F]=[F\rightarrow 2;T\rightarrow fn_1[T]]$

$=fn_1[T]$

$=\lambda[[y];fn[y]][T]$

$=fn[T]$

$=[T\rightarrow 2;\ldots]$

$=2$

hence $Label[fn;fn_2][F]=2$ when y=F and so e=2.

<u>3.1.2</u>

Let $fn_1^*,e_1^*,e_2^*,fn_2^*$ be the codings into S-expressions of $fn_1,e_1,e_2,$ fn_2 respectively, then the value of e is:

evalquote/

evalquote[(LAMBDA(Y)((LABEL FN fn_2^*)(QUOTE F)));(F)]

=apply[(LAMBDA(Y)((LABEL FN fn_2^*)(QUOTE F)));(F);NIL]

=eval[((LABEL FN fn_2^*)(QUOTE F));((Y.F))]

=apply[(LABEL FN fn_2^*);(F);((Y.F))]

=apply[fn_2^*;(F);((FN.fn_2^*)(Y.F))]

=eval[e_2^*;((X.F)(FN.fn_2^*)(Y.F))]

=eval[e_1^*;((X.F)(FN.fn_2^*)(Y.F))]

=apply[fn_1^*;(T);((X.F)(FN.fn_2^*)(Y.F))]

=eval[(FN Y);((Y.T)(X.F)(FN.fn_2^*)(Y.F))]

=apply[FN;(T);((Y.T)(X.F)(FN.fn_2^*)(Y.F))]

=apply[fn_2^*;(T);((Y.T)(X.F)(FN.fn_2^*)(Y.F))]

=eval[e_2^*;((X.T)(Y.T)(X.F)(FN.fn_2^*)(Y.F))]

=1

Thus unfortunately, these two chains of reasoning lead to different results; that described in 3.1.1 is short, lucid and wrong whilst that of 3.1.2 is tortuous, fails to exploit the intuitive meaning of LISP but is right. The intuitions which make 3.1.1 seem correct I shall refer to as "denotational", those used in 3.1.2 I shall call "operational". What the above shows is that denotational and operational thinking are not necessarily consistent with one another. It turns out that denotational thinking can be 'debugged' so that it becomes in harmony with the interpreter whilst still possessing its crisp and lucid quality, most of the rest of this thesis is devoted to doing this (and proving that it is done successfully).

Why/

Why should one be interested in the denotational intuitions about LISP? There are two (related) answers to this:

1) When it works, denotational intuition is a very powerful way of 'seeing' facts about LISP. For example it is much easier to apprehend directly the truth

$$\forall x.\ x=[\mathrm{atom}[x]\rightarrow x; T\rightarrow \mathrm{cons}[\mathrm{car}[x];\mathrm{cdr}[x]]]$$

than to follow the evaluation of

```
(COND((ATOM X)X)
     ((QUOTE T)(CONS(CAR X)(CDR X))))
```

on the interpreter. Thus when one is trying to formulate rules of inference for reasoning about LISP it is helpful to know which part of one's intuition to trust and which to reject.

2) When we read the definitions of the functions apply,eval,etc. we feel that in some sense they are 'right'. There is something against which we judge these functions and our intuition tells us that this thing is in harmony with them. It would be quite possible for apply,eval,etc. not to satisfy us (e.g. if there were a misprint in their definitions) and then we would reject them as 'wrong'. This thing, which is prior to the interpreter, is (I contend) the basis of the denotational intuition of LISP and it needs to be laid out in the open so that/

that any errors it might lead us to are brought to light.

3.2 Some intuitions about LISP

I want now to list three denotational intuitions, I1, I2, I3, I have about LISP. These intuitions certainly are not exhaustive but they are particularly important and it is not clear at first sight to what extent they are consistent with operational intuitions as expressed in the manual interpreter (i.e. apply,eval,evcon,evlis,etc.).

I1. There are two kinds of expressions in LISP, forms and functions. Some forms have a value which is then an S-expression, other forms are undefined and have no value (e.g. car[NIL], label[f;f][NIL]). Functions denote partial mappings from S-expressions to S-expressions; they may have one or more arguments.

I2. The value of a form $fn[e_1;...;e_n]$ is got by applying (in the mathematical sense) the mapping denoted by fn to the values of $e_1,...,e_n$.

I3. The denotation of a function $Label[f;\lambda[[x_1;...;x_n];e]]$ is a mapping f which satisfies the equation

$$f(x_1,...,x_n)=e$$

One can use I1-I3 in an extremely loose way to evaluate forms. For example/

example, to work out the value, v say, of

$$\text{Label}[f;\ \lambda[[x];[\text{atom}[x]\rightarrow x;\ T\rightarrow f[\text{car}[x]]]]][\text{cons}[1;\text{NIL}]]$$

we have	v=f[cons[1;NIL]]	by I3
	=f[(1)]	by I2
	=f[car[(1)]]	by I3
	=f[1]	by I2
	=1	by I3

By making I1-I3 precise one can convert informal 'proofs' of the above form into rigorous, formal proofs.

One reason why it is not obvious that I1-I3 are consistent with the interpreter in the LISP Manual is that mappings are usually infinite objects and mathematical application and equation-solving are not necessarily effective processes. Effective interpreters must manipulate finite representations of mappings and try to mimic application and equation-solving in finitistic terms. Because of this, if one takes the manual interpreter as a definition of LISP then I1-I2 are not immediately available for use. Since we want to exploit these intuitions in reasoning about LISP (e.g. see above or [3]) and since actual LISP systems are (loosely) based on the manual interpreter it is useful to attempt a reconciliation between the denotational and operational aspects.

To/

To do this I formalize both aspects (in 3.4, 4.2 respectively); to formalize the mathematical semantics I use techniques due to Scott and Strachey [22] - these are the only satisfactory techniques I have heard of.

3.3 Introduction to the Scott and Strachey approach to Semantics

The basic idea of this approach is to "extend BNF to semantics". The meaning of a language is described by a sequence of equations whose left hand sides consist of the various expressions in the language and whose right hand sides give the corresponding meanings. The original thing about the approach is the nature of these meanings and the assumptions made about the various sets in which they occur. These assumptions or axioms are rather subtle and at first sight can appear ad hoc and arbitrary. Considerable work has shown that in fact the axioms are consistent with intuition; indeed not only are they that but they capture in an amazingly concise and elegant way just enough of our intuition to allow useful results to be cleanly proved. To fully motivate and justify the axioms would take a lot of space and in any case it has been adequately done elsewhere (see e.g. [19], [20]). Instead I shall just list the assumptions and sketch the intuitions they axiomatize.

The kind of descriptions advocated by Scott and Strachey are often called "denotational" or "mathematical" semantics. Using such a description/

description commits one to regarding meanings as mathematical objects and the relation obtaining between an expression and its meaning as being a relation of denotation holding between the expression and the corresponding object. Because of I1 above this commitment is just what is wanted here, however, it is possible that in other languages one may feel that it is unnatural to regard programs as denoting anything and in that case a denotational semantics may be inappropriate.

3.4 Scott's Axioms: preliminary motivation

I said earlier (in I3) that I thought that the denotation, f, of Label[f; λ[[x_1;...;x_n];e]] should satisfy:

$$f(x_1,\dots,x_n)=e \qquad (1)$$

Now there could be many f's which satisfy this equation and if so how are we to choose among them? Well, given certain plausible assumptions, it turns out that there is always a unique <u>least</u> solution of (1) (where I mean "least" in the sense of the inclusion ordering, $\sqsubseteq$, on partial functions). Now there does not seem to be <u>anything more</u> in the meaning of Label over and above the requirement that (1) is satisfied hence f should not have any properties not forced on it by (1) and so the least solution would seem to be the one wanted. Although this argument is rather weak, there is considerable evidence (e.g. in [2]) that not only are least solutions of recursion equations the ones really intended but also analysing recursion in this way gives one enormous technical benefits/

benefits.

The point of the above is to show the usefulness, for semantics, of focussing on the order relation which exists naturally on the set of denotations of functions.

Consider now the set of denotations of forms; if a form is defined it should denote its value - an S-expression - but what if it has no value? Here, perhaps, is a case of an expression which cannot naturally be said to denote anything. It turns out to be very convenient (and not too unnatural I find) to introduce an 'undefined' object $\perp$ which undefined forms can denote. Any S-expression is more defined than the undefined object : this intuition can be expressed by introducing a relation $\sqsubseteq$ (read "is less defined than") such that $\perp \sqsubseteq$ A for all S-expressions A.

Thus on the set of denotations of forms and also on the set of denotations of functions there is a relation $\sqsubseteq$, and in both cases it expresses a notion of definedness. In fact on many sets of interest such a relation $\sqsubseteq$ is naturally definable; in particular this is so on those sets which are composed of intuitively computable members - the purpose of the relation being to partially axiomatize the computability property of the elements. The first assumption of Scott's theory is that all such 'effective' sets are ordered by a relation $\sqsubseteq$, and such $\sqsubseteq$'s are not arbitrary but satisfy certain axioms which I detail below (see/

(see Appendix 1 also). To state these axioms I first need to give some definitions.

3.5 Some Definitions needed in order to state Scott's axioms

3.5.1 Definition

A partially ordered set is a set D together with a binary relation such that for all $x,y,z \in D$:

(1) $x \sqsubseteq x$

(2) $x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$

(3) $x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x=y$.

Remark: I shall use the symbol $\sqsubseteq$ for all such partial orders, if context fails to specify which particular relation $\sqsubseteq$ denotes I shall use $\sqsubseteq_D$. I shall use $x \sqsupseteq y$ to mean $y \sqsubseteq x$.

3.5.2 Definition

An element $\perp$ of a partially ordered set D is a least or minimum element if for all $x \in D$. $\perp \sqsubseteq x$.

Remark: I shall use $\perp$ or (if context demands it) $\perp_D$ for such least elements. They are always unique if they exist.

3.5.3 Definition

If X is a subset of a partially ordered set D, then $z \in D$ is said to be/

be an <u>upper bound</u> of X if for all $x \in X . x \sqsubseteq z$. If for every other upper bound z' of X $z \sqsubseteq z'$ then z is said to be a <u>least upper bound</u> (lub).

Remark: It is easy to show that a set $X \subseteq D$ has at most one least upper bound which (if it exists) I shall denote by $\bigsqcup_D X$, $\bigsqcup X$ or $\bigsqcup_{x \in X} x$. I shall sometimes say that X has a least upper bound by saying that $\bigsqcup X$ exists or is defined.

3.5.4 <u>Definition</u>

A subset X of a partially ordered set D is said to be <u>directed</u> if it contains an upper bound for each of its finite subsets.

Remark: If X is directed then X contains an upper bound of the empty set and so is non-empty.

3.5.5 <u>Definition</u>

A <u>Semi-domain</u> is a partially ordered set D such that:

(1) D contains a least element

(2) Every directed subset of D has a least upper bound in D.

3.5.6 <u>Definition</u>

If D_1, D_2 are semi-domains and $f: D_1 \to D_2$ then f is <u>monotonic</u> if for all $x, y \in D$

$x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$

f/

f is strict if $f(\perp)=\perp$.

3.5.7 Definition

If D_1, D_2 are semi-domains and $f: D_1 \to D_2$ then f is continuous if for all directed $X \subseteq D$, $\bigsqcup\{f(x) | x \in X\}$ is defined and equals $f(\bigsqcup X)$. (i.e. $f(\bigsqcup X)=\bigsqcup f(X)$).

Remark: If f is continuous then f is monotonic (Proof: Consider directed sets of the form $\{x,y\}$ QED).

3.6 Statement of Scott's axioms

3.6.1 Axiom

Effective sets are semi-domains.

3.6.2 Axiom

Computable functions are continuous.

The notions of "effective" and "computable" are intuitive; using the axioms consists in modelling data types with semi-domains and requiring all admissible functions between data types to be continuous. I do not intend to justify these axioms, this is done in [19] and [20]. I showed above that the denotations of LISP forms and functions were ordered by a relation $\sqsubseteq$. It is easy to check that these sets, together with $\sqsubseteq$, form semi-domains. Axiom 3.6.2 serves to eliminate very/

very 'uncomputable' functions from the theory. These functions, if let in, would prevent certain theorems, which are true of all 'reasonably computable' functions, from being proved. In fact I am only really interested in computable functions, but the theorems needed are true of the wider class of continuous functions and this class is nicer to work with. The situation is well expressed in the following quote from Reynolds [19]:

> "The fact that Scott's assumptions are weaker than the usual notion of computability may be a considerable virtue. The generalization from computable to continuous functions is much like the generalization from algebraic to real numbers. In both cases one moves from a small but subtle set, determined by a certain kind of finite, implicit representation, to a larger but structurally simpler set which can be constructed by limiting processes."

The "limiting processes" mentioned at the end of the quotation concern further axioms, which Scott has proposed, which imply that any admissible function is a limit of computable ones. My purposes do not require these extra axioms which, consequently, I do not state. Before giving an example of a denotational semantics I need to describe some notation and give a few more definitions.

3.7 Some more definitions and notation

3.7.1/

3.7.1 Definition

If D_1, D_2 are semi-domains then D_1 is a sub semi-domain of D_2 if:

(1) $D_1 \subseteq D_2$

(2) $\perp_{D_1} = \perp_{D_2}$

(3) If $X \subseteq D_1$ is directed then $\bigsqcup_{D_1} X = \bigsqcup_{D_2} X$

Remark: Thus a sub semi-domain of D is a subset which contains $\perp$ and is closed under directed unions.

3.7.2 Definition

Semi-domains D_1, D_2 are isomorphic if there exist continuous functions $f: D_1 \to D_2$, $g: D_2 \to D_1$ such that:

(1) For all $x \in D_1$ $g(f(x))=x$

(2) For all $y \in D_2$ $f(g(y))=y$

Remark: If I_D (or I) denotes the identity function on D and $f \circ g$ denotes the function composition of g followed by f then (1), (2) of the above definition can be written as $g \circ f = I_{D_1}$ and $f \circ g = I_{D_2}$ respectively. $D_1 \cong D_2$ means D_1 and D_2 are isomorphic.

3.7.3 Definition

If D_1, D_2 are semi-domains let $[D_1 \to D_2] = \{f: D_1 \to D_2 \mid f \text{ is continuous}\}$, order $[D_1 \to D_2]$ by $f \sqsubseteq g \iff$ for all $x \in D_1$, $f(x) \sqsubseteq g(x)$.

Remark: It is easy to show that $[D_1 \to D_2]$ ordered by $\sqsubseteq$ is a semi-domain and/

and that if F is a directed subset of it then:

$$\forall x \in D_1 \,.\, (\bigsqcup F)(x) = \bigsqcup_{f \in F} f(x)$$

3.7.4 Definition

If D_1, D_2 are semi-domains then the product of D_1 and D_2, $D_1 \times D_2$ is the set. $\{(x,y) | x \in D_1 \text{ and } y \in D_2\}$ ordered by:

$$(x,y) \sqsubseteq (x',y') \iff x \sqsubseteq x' \text{ and } y \sqsubseteq y'$$

Remark: It is easy to show that $D_1 \times D_2$ is a semi-domain, that $\bigsqcup_i (x_i, y_i) = (\bigsqcup_i x_i, \bigsqcup_i y_i)$ and that the projection and pairing functions are continuous.

I shall denote the product $D \times D \ldots \times D$, of D with itself $n>0$ times, by D^n, $D^0 = \{\perp\}$.

3.7.5 Definition

If D_1, D_2 are semi-domains then the coalesced product $D_1 \otimes D_2$ is the set $\{\langle x,y\rangle | x \in D_1 \text{ and } y \in D_2\}$ ordered by:

$$\langle x,y\rangle \sqsubseteq \langle x',y'\rangle \iff x \text{ or } y \text{ is } \perp \text{ or } (x \sqsubseteq x' \text{ and } y \sqsubseteq y')$$

Remark: I do not use the coalesced product until 6.6.20 but I have included it here because its definition belongs with the others in this section. Note that $D_1 \otimes D_2$ is got from $D_1 \times D_2$ by identifying with $\perp$ all (x,y) such that $x = \perp$ or $y = \perp$.

3.7.6/

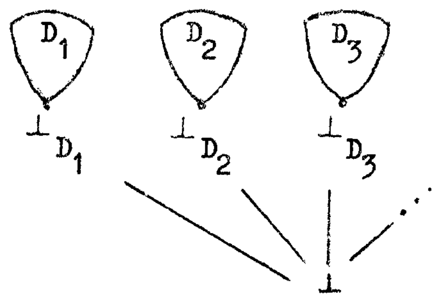
3.7.6 Definition

Let $D_1, D_2, \dots, D_n$ $(1 \leq n \leq \infty)$ be a sequence of semi-domains. Then the separated sum $\sum_{i=1}^{n} D_i$, is defined by:

$$\sum_{i=1}^{n} = \{(i,x_i) \mid x_i \in D_i \text{ and } 1 \leq i \leq n\} \,\dot{\cup}\, \{\perp\} \quad (\dot{\cup} \text{ is disjoint union})$$

$\sum_{i=1}^{n}$ is ordered by: $\perp \sqsubseteq (i,x_i)$ and $(i,x_i) \sqsubseteq (j,x_j) \Leftrightarrow i=j$ and $x_i \sqsubseteq x_j$.

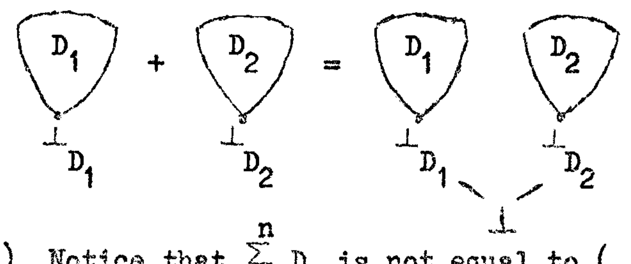
Remark: $\sum_{i=1}^{n} D_i$ can be pictured by:



3.7.7 Definition

$D_1 + D_2 = \sum_{i=1}^{2} D_i$.

Remarks: (1) In pictures we have:



(2) Notice that $\sum_{i=1}^{n} D_i$ <u>is not</u> equal to $(\dots((D_1+D_2)+D_3),\dots)$

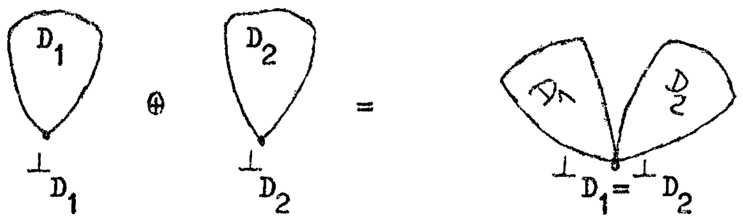
3.7.8/

3.7.8 <u>Definition</u>

If D_1, D_2 are semi-domains then the <u>coalesced sum</u> $D_1 \oplus D_2$ is the set $D_1 \dot{\cup} D_2$ ordered by:

$$x \sqsubseteq y \text{ if } x = \perp_{D_1} \text{ or } x = \perp_{D_2} \text{ or } (x,y \in D_1 \text{ and } x \sqsubseteq y)$$
$$\text{or } (x,y \in D_2 \text{ and } x \sqsubseteq y)$$

Remark: $D_1 \oplus D_2$ can be pictured as:



$D_1 \oplus D_2$ is got from $D_1 + D_2$ by identifying $\perp$, $\perp_{D_1}$, $\perp_{D_2}$.

Associated with the sum $\sum_{i=1}^{n} D_i$ are some injection and projection functions. Notations for these are given in the next definition.

3.7.9 <u>Definition</u>

Let $D_1, \ldots, D_n$ $(1 \leq n \leq \infty)$ be semi-domains then:

(1) For $x_j \in D_j$ $(1 \leq j \leq n)$ let $(x_j \text{ in } \sum_{i=1}^{n} D_i) = (j, x_j) \in \sum_{i=1}^{n} D_i$

(2) For $x \in \sum_{i=1}^{n} D_i$ and $1 \leq j \leq n$ let $x | D_j = \begin{cases} x_j & \text{if } x = (j, x_j) \\ \perp & \text{otherwise} \end{cases}$

Remarks: (1) The functions $x_j \mapsto (x_j \text{ in } \sum_{i=1}^{n} D_i): D_j \to \sum_{i=1}^{n} D_i$ and $x \mapsto x_j | D_j : \sum_{i=1}^{n} D_i \to D_j$ are continuous.

(2)/

(2) If $x_j \in D_j$ then $(x_j \text{ in } \sum_{i=1}^{n} D_i) \mid D_j = x_j$

Suppose $x=f(x)$ is a recursion equation. I shall denote its least solution by $Y(f)$: the rigorous version of this remark now follows.

3.7.10 Definition

If D is a semi-domain and $f \in [D\to D]$, define $Y(f)$ by:

$$Y(f) = \bigsqcup\{f^n(\perp) \mid n \geq 0\} = \bigsqcup_{n=0}^{\infty} f^n(\perp)$$

Remark: Here $f^n = f \circ f \circ \ldots \circ f$ n-times. If D is not clear from context I shall use Y_D. This definition is valid since if $f \in [D\to D]$ then f is monotonic and $\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \ldots \sqsubseteq f^n(\perp) \sqsubseteq \ldots$ so $\{f^n(\perp) \mid n \geq 0\}$ is directed.

The following proposition shows that Y, as defined above, does indeed extract least solutions of $x=f(x)$.

3.7.11 Proposition

(1) $Y \in [[D\to D]\to D]$ (i.e. Y is continuous)

(2) $f(Y(f)) = Y(f)$ (i.e. $Y(f)$ is a fixed-point of f)

(3) for all $x \in D$, $f(x)=x$ implies $Y(f) \sqsubseteq x$ (i.e. $Y(f)$ is the least fixed-point of f)

Proof

(1) I need to show that if $F \subseteq [D\to D]$ is directed then $\bigsqcup_{f \in F} Y(f) = Y(\bigsqcup F)$.

Now/

Now $Y(\sqcup F)=\bigsqcup_{n=0}^{\infty}(\sqcup F)^n(\perp)$

and $\bigsqcup_{f\in F}Y(f)=\bigsqcup_{f\in F}\ \bigsqcup_{n=0}^{\infty}f^n(\perp)=\bigsqcup_{n=0}^{\infty}\ \bigsqcup_{f\in F}f^n(\perp)$

so it is enough to show that for each $n>0$ $(\sqcup F)^n(\perp)=\bigsqcup_{f\in F}f^n(\perp)$.

But $(\sqcup F)^n(\perp)=((\bigsqcup_{f_1\in F}f_1)\circ\ldots\circ(\bigsqcup_{f_n\in F}f_n))(\perp)$

$$=\bigsqcup_{f_1\in F}\ldots\bigsqcup_{f_n\in F}f_1\circ\ldots\circ f_n(\perp)$$

and $\bigsqcup_{f\in F}f^n(\perp)=\bigsqcup_{f\in F}f\circ\ldots\circ f(\perp)$.

Now clearly $\bigsqcup_{f\in F}f\circ\ldots\circ f(\perp)\sqsubseteq\bigsqcup_{f_1\in F}\ldots\bigsqcup_{f_n\in F}f_1\circ\ldots\circ f_n(\perp)$ and since F is directed if $f_1,\ldots,f_n\in F$ then there exists $f\in F$ such that $f_1\sqsubseteq f,\ldots,f_n\sqsubseteq f$ and hence $\bigsqcup_{f_1\in F}\ldots\bigsqcup_{f_n\in F}f_1\circ\ldots\circ f_n(\perp)\sqsubseteq\bigsqcup_{f\in F}f\circ\ldots\circ f(\perp)$ also.

(2) $f(Y(f))=f(\bigsqcup_{n=0}^{\infty}f^n(\perp))=\bigsqcup_{n=0}^{\infty}f(f^n(\perp))=\bigsqcup_{n=1}^{\infty}f^n(\perp)=\bigsqcup_{n=0}^{\infty}f^n(\perp)$ (as $f^0(\perp)=\perp$).

(3) Suppose $x=f(x)$ then $\perp\sqsubseteq x$ so $f^n(\perp)\sqsubseteq f^n(x)=f(f(\ldots f(x)\ldots))=x$ hence $\bigsqcup_{n=0}^{\infty}f^n(\perp)\sqsubseteq x$ i.e. $Y(f)\sqsubseteq x$.

Q.E.D.

When I explain recursive definitions in terms of solving equations, Y will be used to get the solution.

3.7.12 <u>Definition</u>

A semi-domain of the form:

$x_1/$

$$\begin{tikzcd} x_1 \arrow[drr, dash] & x_2 \arrow[dr, dash] & x_3 \arrow[d, dash] & \cdot \arrow[dl, dash] \\ & & \bot \end{tikzcd}$$

is called <u>flat</u>. D is flat if and only if $x,y \in D$ and $x \sqsubseteq y \Rightarrow x=\bot$ or $x=y$. If S is a set then flat(S) is the semi-domain obtained by adjoining $\bot$ to S and imposing the ordering $\bot \sqsubseteq x$ for all $x \in S$. More precisely $\text{flat}(S)=S \,\dot{\cup}\, \{\bot\}$, $\forall x,y \in \text{flat}(S).x \sqsubseteq y \Leftrightarrow x=\bot$ or $x=y$.

Remark: The set of denotations of LISP forms is flat(<S-expression>).

3.7.13 <u>Lambda notation</u>

If D_1,D_2 are semi-domains and $E(t_1,\dots,t_n)$ is an expression which takes values in D_2 when variables $t_1,\dots,t_n$ range over D_1. Then

$\lambda t_1\dots t_n: D_1. E(t_1,\dots,t_n)$ denotes the function $f: D_1^n \to D_2$ such that $f(x_1,\dots,x_n)=E(x_1,\dots,x_n)$.

Remarks: (1) This is a notation of <u>my</u> metalanguage (<u>not</u> of the "LISP metalanguage" i.e. M-expressions) it should be distinguished from LISP functions of the form $\lambda[[x_1;\dots;x_n];e]$. The semantics to be presented will connect up these two uses of λ and describe the LISP metalanguage (M-expressions) in terms of my metalanguage.

(2) I shall sometimes omit the type indication "$:D_1$" if it is clear from context.

(3)/

(3) I shall sometimes use $(t_1,\ldots,t_n) \mapsto E(t_1,\ldots,t_n)$ as an infix version of $\lambda t_1 \ldots t_n .\ E(t_1,\ldots,t_n)$.

3.7.14 Syntactic notation

I want to use the kind of BNF notation used by Scott and Strachey. I think that for my purposes it is rather more lucid and technically convenient than standard BNF. In this notation the definition of the syntactic class <function> of the LISP 1.5 Manual (see p. 9) could be phrased:

Meta variables

e	ranges	over	<form>
fn	"	"	<function>
x	"	"	<identifier>
vl	"	"	<var list>

Syntax equations

fn ::= x | λ[vl;e] | Label[x;fn]

vl ::= $[x_0;\ldots;x_n]$

Thus such a definition consists of two parts:

(1) A set of meta variables (e,fn,x,vl, in this case) together with the syntactic classes they range over (<form>,<function>, <identifier>/

<identifier>,<var list> here).

(2) A set of BNF-like equations specifying the structure of the syntactic classes being defined (<function>, <var list>) in terms of assumed known classes (<form>, <identifier>). Subscripts on the meta variables are used to distinguish different occurrences of the same meta variable in a single expression. Thus if n ranges over integers then $e ::= n \mid n_0.e.n_1$ defines e to range over all strings of integers but $e ::= n \mid n.e.n$ defines e to range just over palindromes.

3.7.15 Standard syntactic classes

Let <identifier>, <S-expression> be the syntactic classes defined in the LISP 1.5 Manual. Thus <identifier> consists of strings of numerals and lower case letters in which the first character is a letter and

<S-expression> ::= <atomic symbol> | (<atomic symbol>.<atomic symbol>)

(where <atomic symbol> consists of strings of numerals and upper case letters in which the first character is a letter).

I shall use three meta variables x,f,z to range over <identifier>: x will be used in contexts where the identifier is a form, f where it is a function and z where it could be either.

I/

I shall use the meta variable A to range over <S-expression>.

3.7.16 Standard functions on S-expressions

In order to explain semantically expressions such as car[cdr[NIL]] it is necessary for the standard functions car, cdr, cons, atom, eq to be defined on S=flat(<S-expression>) rather than just on <S-expression>, then car[cdr[NIL]] denotes car($\perp$)=$\perp$. I am going to use the same names for the standard LISP functions and their meanings (as I did in the last sentence), no confusion should result because in LISP square brackets [and] are used whereas in my meta language I shall use round brackets (and). Thus car[cdr[NIL]] is a LISP form which means car(cdr(NIL)). Note that the so called LISP meta language of the manual, - i.e. the language of LISP M-expressions, is my <u>object language</u>, when I use "meta language" I mean <u>my</u> meta language not M-expressions.

Now define: car: $S\to S$

cdr: $S\to S$

cons: $S^2\to S$

atom: $S\to S$

eq: $S^2\to S$

by/

by

$$\text{car}(t)=\begin{cases}\perp & \text{if } t=\perp \text{ or } t \text{ is an atomic symbol}\\ A_1 & \text{if } t=(A_1.A_2)\end{cases}$$

$$\text{cdr}(t)=\begin{cases}\perp & \text{if } t=\perp \text{ or } t \text{ is an atomic symbol}\\ A_2 & \text{if } t=(A_1.A_2)\end{cases}$$

$$\text{cons}(t_1,t_2)=\begin{cases}\perp & \text{if } t_1=\perp \text{ or } t_2=\perp\\ (t_1.t_2) & \text{otherwise}\end{cases}$$

$$\text{atom}(t)=\begin{cases}\perp & \text{if } t=\perp\\ T & \text{if } t \text{ is an atomic symbol}\\ F & \text{if } t \text{ is composite (i.e. of the form } (A_1.A_2))\end{cases}$$

$$\text{eq}(t_1,t_2)=\begin{cases}\perp & \text{if } t_1=\perp \text{ or } t_2=\perp \text{ or } t_1 \text{ is composite or } t_2 \text{ is composite}\\ T & \text{if } t_1,t_2 \text{ are atomic symbols and } t_1=t_2\\ F & \text{if } t_1,t_2 \text{ are atomic symbols and } t_1\neq t_2\end{cases}$$

3.7.17 Conditional expressions

If $t_{11},t_{12},\ldots,t_{n1},t_{n2} \in S$, where S=flat(<S-expression>), then $(t_{11}\rightarrow t_{12},\ldots,t_{n1}\rightarrow t_{n2}) \in S$ is defined by induction on n as follows:

n=1:
$$(t_{11}\rightarrow t_{12})=\begin{cases}t_{12} & \text{if } t_{11}=T\\ \perp & \text{otherwise}\end{cases}$$

n>1:
$$(t_{11}\rightarrow t_{12},t_{21}\rightarrow t_{22},\ldots,t_{n1}\rightarrow t_{n2})=\begin{cases}t_{12} & \text{if } t_{11}=T\\ (t_{21}\rightarrow t_{22},\ldots,t_{n1}\rightarrow t_{n2}) & \text{if } t_{11}=F\\ \perp & \text{otherwise}\end{cases}$$

Thus/

Thus $(t_{11}\to t_{12},\ldots,t_{n1}\to t_{n2})=t_{m2} \Leftrightarrow ((1\leq i<m) \Rightarrow t_{i1}=F$ and $t_{m1}=T)$

Also let

$$(t_{11}\to t_{12},\ldots,t_{n1}\to t_{n2},t_{n3})=(t_{11}\to t_{12},\ldots,t_{n1}\to t_{n2},T\to t_{n3})$$

so in particular (with n=1)

$$(t_{11}\to t_{12},t_{13})=(t_{11}\to t_{12},T\to t_{13})=\text{if } t_{11} \text{ then } t_{12} \text{ else } t_{13}.$$

3.8 <u>An example: a naive denotational semantics of LISP</u>

3.8.1 <u>The semi-domains of denotations and environments</u>

Recall intuition I1 of 3.2; formalizing this we see that a form denotes a member of S=flat(<S-expression>) and a function (being computable and hence continuous) denotes a member of $[S^n\to S]$ (where n is the number of arguments it takes). Since we want to handle functions of arbitrary numbers of arguments we need to take $FUN=\sum_{n=1}^{\infty}[S^n\to S]$ as the semi-domain of denotations of LISP functions. Thus a LISP expression, whether it be a form or a function, denotes a member of D where

$$D=S+FUN$$
$$S=\text{flat}(\langle\text{S-expression}\rangle)$$
$$FUN=\sum_{n=1}^{\infty}[S^n\to S]$$

What I have just said is not quite right: what, for example does the form f[x] denote? We cannot answer this until we know an environment which/

which tells us what the identifiers f and x denote. Such an environment is a map r: <identifier>->D, it is desirable that it is computable so we should like it to be continuous, this is made possible with minimal ad hocery by letting

Id=flat(<identifier>)

and then requiring $r \in [\text{Id}\text{->}D]$. Thus the semi-domain, Envr, of environments is given by:

Envr=[Id->D]

Now I can patch up the above remarks about what forms and functions denote to take into account of free variables. The denotation of a form or function is a function of the environment thus the correct semi-domains of denotations are [Envr->S], [Envr->FUN] for <form>, <function> respectively where

D=S+FUN

S=flat(<S-expression>)

$$\text{FUN}=\sum_{i=1}^{\infty}[S^n\text{->}S]$$

Here is some notation which I need later on.

(1): Given an $r \in \text{Env}$, $x \in \text{Id}$, $t \in D$ let (t/x)r be the result of 'updating'r so that t is assigned to x. More precisely

$$(t/x)r=\lambda x'.\ (\text{if } x=\perp \text{ or } x'=\perp \text{ then } \perp \text{ else if } x=x' \text{ then } t \text{ else } r(x'))$$

(2):/

(2): If $t \in S$ then $(t/x)r$ is an abbreviation for $((t \text{ in } D)/x)r$ and if $F \in FUN$ then $(F/f)r$ is an abbreviation for $((F \text{ in } D)/f)r$.

(3): Suppose $t_1,\ldots,t_n \in S$ and $F \in FUN$ then let

$$F(t_1,\ldots,t_n)=F \mid [S^n\text{->}S]\ (t_1,\ldots,t_n)$$

Then $F(t_1,\ldots,t_n)$ is what you would expect when it makes sense (i.e. when F takes n arguments) and $\perp$ otherwise.

3.8.2 Provisional syntax of pure LISP

The syntactic definition of pure LISP I am going to use for the time being is given below in the notation of 3.7.14. In 3.13 I amend this syntax slightly.

Meta variables

A	ranges	over	<S-expression>
x,f	"	"	<identifier>
e	"	"	<form>
fn	"	"	<function>

Syntax equations

$$e ::= A \mid x \mid fn[e_1;\ldots;e_n] \mid [e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}]$$

$$fn ::= car \mid cdr \mid cons \mid atom \mid eq \mid f \mid \lambda[[x_1;\ldots;x_n];e] \mid Label[f;fn]$$

Remark:/

Remark: I hope it is clear that this definition amounts to the same thing as the one in the Manual.

3.8.3 The inadequate semantics

I am now going to define the denotations $V[\![e]\!](r)$, $W[\![fn]\!](r)$ of forms e and functions fn relative to an environment r. The 'emphatic' brackets $[\![$, $]\!]$ are just an aid to the eye, they always enclose expressions from the object language. The semantic functions V,W which map expressions to their denotations are of type V: <form>->[ENVR->S]

W: <function>->[ENVR->FUN]

respectively. They are defined by structural induction with the following semantic equations which I give and then explain.

Semantics (First attempt)

Denotations: D=S+FUN

S=flat(<S-expression>)

$FUN=\sum_{n=1}^{\infty}[S^n\text{->}S]$

Environments: Envr=[Id->D]

Semantic functions: V: <form>->[Envr->S]

Semantic/

Semantic equations: $W: \langle function\rangle \to [Envr \to FUN]$

(S1) $V[\![A]\!](r)=A$

(S2) $V[\![x]\!](r)=r(x) \mid S$

(S3) $V[\![fn[e_1;\dots;e_n]]\!](r)=W[\![fn]\!](r)(V[\![e_1]\!](r),\dots,V[\![e_n]\!](r))$

(S4) $V[\![[e_{11}\to e_{12};\dots;e_{n1}\to e_{n2}]]\!](r)=(V[\![e_{11}]\!](r)\to V[\![e_{12}]\!](r),\dots, V[\![e_{n1}]\!](r)\to V[\![e_{n2}]\!](r))$

(S5) $W[\![car]\!](r)=(car \text{ in } FUN)$

$W[\![cdr]\!](r)=(cdr \text{ in } FUN)$

$W[\![cons]\!](r)=(cons \text{ in } FUN)$

$W[\![atom]\!](r)=(atom \text{ in } FUN)$

$W[\![eq]\!](r)=(eq \text{ in } FUN)$

(S6) $W[\![f]\!](r)=r(f) \mid FUN$

(S7) $W[\![\lambda[[x_1;\dots;x_n];e]]\!](r)=\lambda t_1\dots t_n{:}S.V[\![e]\!]((t_1/x_1)\dots(t_n/x_n)r)$

(S8) $W[\![Label[f;fn]]\!](r)=Y(\lambda F{:}FUN.W[\![fn]\!]((F/f)r))$

Explanation

(S1) says that S-expressions denote themselves in all environments

(S2) says that, relative to an environment r, a form variable x denotes the S-expression it is 'bound to' in r. If x isn't bound to an S-expression but to a member of FUN then x denotes $\perp$ in r. This last remark corresponds to the "$\mid S$" in (S2).

(S3) makes precise intuition I2 of 3.2. Notice that by the convention described at the end of 3.8.1 (S3) is really:

V/

$$V[\![fn[e_1;\ldots;e_n]]\!](r)=W[\![fn]\!](r)\mid[S^n\text{->}S](V[\![e_1]\!](r),\ldots,V[\![e_n]\!](r))$$

(S4) just says conditional forms denote conditionals.

(S5) says that in all environments the standard functions get the appropriate meanings.

(S6) is analogous to (S2): a function variable f denotes, in r, r(f) if this is a function and $\perp$ otherwise.

(S7) connects the λ notations of the object and meta languages. It says that in r $\lambda[[x_1;\ldots;x_n];e]$ denotes the function f: $S^n\text{->}S$ which maps $(t_1,\ldots,t_n)$ to the value of e in an environment in which x_i is 'bound' to t_i. Note that by the convention described at the end of 3.8.1 (S7) is really:

$$W[\![\lambda[[x_1;\ldots;x_n];e]]\!](r)=\lambda t_1\ldots t_n;S.V[\![e]\!](((t_1 \text{ in } D)/x_1)\ldots((t_n \text{ in } D)/x_n)r)$$

(S8) makes precise intuition I3 of 3.2 and also the remarks about least solutions of recursion equations made in 3.4. Label[f;fn] denotes in r the least solution of the equation

$$F=W[\![fn]\!]((F/f)r)$$

If fn= $\lambda[[x_1;\ldots;x_n];e]$ then this can be written as:

$$F(t_1,\ldots,t_n)=V[\![e]\!]((t_1/x_1)\ldots(t_n/x_n)(F/f)r)$$

which is a precise semantic analogue of $f[x_1;\ldots;x_n]=e$. Note that the conventions at the end of 3.8.1 mean that (S8) is really:

$$W[\![Label[f;fn]]\!](r)=Y(\lambda F:FUN.W[\![fn]\!](((F \text{ in } D)/f)r)).$$

3.8.4/

3.8.4 Some example calculations of denotations

I shall use some of the examples below to illustrate certain inadequacies of this semantics.

Let wfn=Label[f;f]

we =Label[f;f][NIL]

Example 1 let e=cons[1;NIL;NIL]

$$\begin{aligned} \text{Then } V[\![e]\!](r) &= W[\![\text{cons}]\!](r)(V[\![1]\!](r), V[\![\text{NIL}]\!](r), V[\![\text{NIL}]\!](r)) \\ &= (\text{cons in FUN}) \mid [S^3\text{->}S](1,\text{NIL},\text{NIL}) \\ &= \perp \quad ((\text{cons in FUN}) \mid [S^3\text{->}S] \neq \perp \Leftrightarrow n=2) \end{aligned}$$

Example 2 let fn=wfn=Label[f;f]

$$\begin{aligned} \text{Then } W[\![fn]\!](r) &= Y(\lambda F.[\![f]\!]((F/f)r)) \\ &= Y(\lambda F.F) \\ &= \bigsqcup_{n=0}^{\infty}(\lambda F.F)^n(\perp) \\ &= \perp \end{aligned}$$

Example 3 let e=we=wfn[NIL]

Then $V[\![e]\!](r) = \perp(\text{NIL}) = \perp$

The following examples come from the discussion in 3.1. "fn" is used as an identifier "fn_2", "fn_3" as names for M-expressions.

Example 4 let e=λ[[x];NIL][we]

Then $V[\![e]\!](r) = (\lambda t.\text{NIL})(\perp) = \text{NIL}$

Example/

<u>Example 5</u> let $fn_2 = \lambda[[x];[y \to 1; x \to 2; T \to \lambda[[y];fn[y]][T]]]$

(fn$_2$ was previously defined in 3.1)

Then
$$\begin{aligned} W[\![fn_2]\!](r) &= \lambda t.(r(y) \mid S \to 1,\\ &\qquad t \to 2,\\ &\qquad T \to (\lambda t'.r(fn) \mid FUN(t'))(T))\\ &= \lambda t.(r(y) \mid S \to 1,\\ &\qquad t \to 2,\\ &\qquad T \to r(fn) \mid FUN(T)). \end{aligned}$$

<u>Example 6</u> let $fn_3 = \text{Label}[fn;fn_2]$ and $r(y) \mid S = F$ (fn$_2$ as above)

Then
$$\begin{aligned} W[\![fn_3]\!](r) &= Y(\lambda F.W[\![fn_2]\!]((F/fn)r))\\ &= W[\![fn_2]\!]((W[\![fn_3]\!](r)/fn)r) \quad (\text{as } Y(F)=F(Y(F)))\\ &= \lambda t.(F \to 1, t \to 2, T \to W[\![fn_3]\!](r)(T))\\ &= \lambda t.(t \to 2, T \to W[\![fn_3]\!](r)(T))\\ &= \lambda t.(t \to 2, T \to (F \to 1, T \to 2, \ldots))\\ &= \lambda t.(t \to 2, T \to 2). \end{aligned}$$

<u>Example 7</u> let $e = \lambda[[y];fn_3[F]][F]$ (fn$_3$ as above)

(This is the e defined in 3.1)

Then
$$\begin{aligned} V[\![e]\!](r) &= (\lambda t.W[\![fn_3[F]]\!]((t/y)r))(F)\\ &= W[\![fn_3]\!]((F/y)r)(F)\\ &= (F \to 2, T \to 2) \qquad (\text{by example 6})\\ &= 2. \end{aligned}$$

Examples 5,6,7 show that this semantics formalizes that intuition used in/

used in 3.1.1. Examples 1,4,7 show different ways in which the semantics diverges from the behaviour of LISP as it runs on actual machines. I shall explain the differences in detail:

Example 1: e=cons[1;NIL;NIL]

Then apply[CONS;(1 NIL NIL);a]=cons[1;NIL]
so in real LISP e evaluates to (1).

Example 4: e=λ[[x]; NIL][Label[f;f][NIL]]
The evaluation of this on an actual LISP system never terminates since arguments are evaluated <u>before</u> being bound on the alist and the evaluation of Label[f;f][NIL] does not terminate. The non-termination of e is not obviously forced by the manual interpreter for we have the calculation:

```
  eval[((LAMBDA(X)(QUOTE NIL))((LABEL F F)NIL));a]
      =apply[(LAMBDA(X)(QUOTE NIL)); evlis[(((LABEL F F)NIL));a];a]
      =eval[(QUOTE NIL); pairlis[(X); evlis[(((LABEL F F)NIL));a];a]]
      =NIL
```

Thus the manual interpreter is not a good specification of actual interpreters, it is for this reason that I do not use apply,eval,...etc. as a formalization of my operational intuitions about LISP, to find out what I ~~do~~ used see 4.2.

Example/

Example 7: $e = \lambda[[y]; fn_3[F]][F]$, fn_3 as in example 6.
The explanation of how e evaluates in practice was given in 3.1.2.

3.9 What can be done about the wrong denotations assigned by the naive semantics?

In 3.8 I formalized a denotational conception of pure LISP, unfortunately as the examples of 3.8.4 show, this formalization does not correspond to reality (as defined by actual LISP systems). There are three possible moves I could now make:

Move 1: Monsterbarring

I could claim that examples such as ex1, ex4, ex7 of 3.8.4 are pathological cases, that one is only interested in the semantics of sensible programs and if "monsters" such as these creep in it is due to sloppy syntax or some such thing. This move is a use of the "method of Monsterbarring" described in [4].

Move 2: The implementers got it wrong

I could claim that my semantics formalizes the intuitive description of LISP given in the Manual and so implementations which are not in accord with it (e.g. the manual interpreter) are just plain wrong.

Move 3: My mathematical intuitions are wrong

I/

I could admit that my semantics does not describe real LISP (though it may describe a beautiful 'platonic' LISP). That, like it or not, LISP programs behave as they do and if a semantics is to be at all useful it must help with deductions about the real thing and not the objects of theoreticians' fantasies.

To decide which move to make I can apply the thought of chairman Mao Tse Tung:

> "If a man wants to succeed in his work, that is, to achieve the anticipated results, he must bring his ideas into correspondence with the laws of the objective external world; if they do not correspond, he will fail in his practice. After he fails, he draws his lessons, corrects his ideas to make them correspond to the laws of the external world, and can thus turn failure into success; this is what is meant by "failure is the mother of success" and "a fall into the pit, a gain in your wit"."
>
> Mao Tse-Tung
> "On Practice" (July 1937)
> Selected Works, Vol.1,pp.296-97.

Thus I must bring my semantics "into correspondence with the laws of the objective external world" and so move 3 is the one I must make. Beside Mao Tse Tung thought there are other arguments against moves 1 and 2. Against move 1 we have the unpleasant fact that although "monsters"/

"monsters" may not arise by <u>design</u> they may well arise by accident and so we need our semantics to tell us what they do so that we can find out that it is wrong. Against move 2 I think we must accept that even if implementers were <u>initially</u> wrong, we have <u>now</u> evolved to a state where LISP is understood as corresponding to what they have implemented and so describing LISP should be describing <u>that</u>. To "turn failure into success" I do not need to completely abandon denotational intuition, I just need to 'debug' it by isolating those aspects that are misleading and patching them up. In doing this care must be taken to preserve as much as possible of the naive intuition's useful features.

What are these useful features of our intuition? Well, I believe they include I1-I3 of 3.2. Which, to briefly recap, are:

I1. Forms denote S-expressions if they are defined. Functions denote mappings (of various arities) from S-expressions to S-expressions.

I2. $fn[e_1;\ldots;e_n]$ denotes the result of applying the mapping denoted by fn to the denotations of $e_1,\ldots,e_n$.

I3. $Label[f;\lambda[[x_1;\ldots;x_n];e]]$ denotes the (least) solution of the equation:

$$f(x_1,\ldots,x_n)=e$$

Before/

Before going on to show how to construct a correct semantics which exploits I1-I3 I should like to describe yet another deficiency of the incorrect one in 3.8.

3.10 Simultaneous recursions: a defect of the language described in 3.8.2

Suppose we wanted to work out the relation between meanings assigned by (S1)-(S8) of 3.8.3, and meanings computed by apply,eval...etc. where the meanings of these are worked out from the semantic equations: we would be in trouble because these functions are defined by a simultaneous recursion, but Label only allows us to construct expressions corresponding to recursion on a single variable. To overcome this one could introduce a sort of generalized Label expression of the form:

$$\text{Label}[[f_1;\ldots;f_n];[fn_1;\ldots;fn_n]]$$

which would allow simultaneous recursions to be expressed (S8) could then be extended to:

$$W[\![\text{Label}[[f_1;\ldots;f_n]:[fn_1;\ldots;fn_n]]]\!](r)$$
$$=Y(\lambda F_1\ldots F_n.\text{FUN}.(W[\![fn_1]\!]((F_1/f_1)\ldots(F_n/f_n)r),\ldots,W[\![fn_n]\!]((F_1/f_1)\ldots(F_n/f_n)r)))$$

The Y here is of type $[\text{FUN}^n\text{->}\text{FUN}^n]\text{->}\text{FUN}^n$ and $\text{Label}[[f_1;\ldots;f_n];[fn_1;\ldots;fn_1]]$ denotes an n-tuple of functions whose components could be got at with projection functions. Notice that example 1 of 3.8.4 shows that V and eval compute different values, thus even without extending LISP as above we can see that V and eval are not the same. Example 4 of 3.8.4 shows that eval, when interpreted via (S1)/

(S1)-(S8), does not correspond to actual interpreters. Thus (S1)-(S8), apply,eval...etc. (interpreted via (S1-(S8)) and real implementations provide three different accounts of pure LISP.

3.11 'Debugging' the semantics of 3.8

There are three features of real LISP which are not reflected in the semantics above, these are:

(1) In real LISP one can give a function too many arguments and, as long as they are all defined, no harm is done.

(2) In real LISP if a form e_i has no value then neither does $fn[e_1;...;e_i;...;e_n]$ for any function fn.

(3) In real LISP variables are fluid - that is they are looked up in the environment when they are evaluated (not when the (outermost) function in which they are free is evaluated).

The way in which the above semantics fails to reflect these is illustrated in examples 1,4,7 of 3.8.4 respectively. Although it is clear that (S1)-(S8) fail in the three ways above it is not so obvious that they _only_ fail in these ways. This is in fact the case as is shown by the main theorem below.

I shall now describe how to modify (S1)-(S8) to incorporate (1)-(3) above.

(1):/

(1):

Here the problem is that in real LISP functions are variadic - it is more correct to think of them as mapping strings of S-expressions to S-expressions than of being of any fixed arity. To incorporate this, FUN must be changed from $\sum_{n=1}^{\infty}[S^n\text{->}S]$ to $[S^*\text{->}S]$ where S*- the semi-domain of <u>finite</u> strings over S- is defined in the definition below.

3.11.1 <u>Definition</u>

If D is a semi-domain define the semi-domain D* by:

$$D^*=\sum_{n=0}^{\infty}D^n$$

Remark: The D^0 of this sum provides an 'empty' string $\perp_{D^0}=\perp|D^0$ which I may denote by (). If $(x_1,\ldots,x_n)$, $(y_1,\ldots,y_m)\in D^*$. Then $(x_1,\ldots,x_n)\sqsubseteq(y_1,\ldots,y_m) \Leftrightarrow n=m$ and $\forall i.x_i\sqsubseteq y_i$.

If f: D*->D' (where D' is some semi-domain) and if $x_1,\ldots,x_n \in D$ then $f(x_1,\ldots,x_n)$ is to be interpreted as $f(((x_1,\ldots,x_n)$ in $D^*))$.
The standard functions, car,cdr,cons,atom,eq can be extended to S* in a natural way (see below).

(2):

To fix this case we need to ensure that whenever an argument denotes $\perp$ the result of applying a function to it denotes $\perp$ also. This is already the case for car,cdr,cons,atom,eq so we need only consider functions/

functions of the form $\lambda[[x_1;\ldots;x_n];e]$ and Label[f;fn]. It turns out that fixing λ-expressions automatically fixes Label-expressions (see corollary 4.3.4) so we just need to arrange that the semantic equation for $\lambda[[x_1;\ldots;x_n];e]$ always assigns it a strict function. Now recall that, because of (1) above, $\lambda[[x_1;\ldots;x_n];e]$ is going to denote a member of [S*->S] (given an environment) thus we also need a new λ-notation (in our meta language) to enable us to talk about functions in [S*->S], this is provided by the following definition which also provides a tool for writing a correct semantic equation for λ-expressions and also for extending the standard functions.

3.11.2 <u>Definition</u>

If D_1, D_2 are semi-domains and $E(t_1,\ldots,t_n)$ is an expression which takes values in D_2 when variables $t_1,\ldots,t_n$ range over D_1. Then

$\underline{\lambda}t_1\ldots t_n: D_1.E(t_1,\ldots,t_n)$ denotes the function $f: D_1^* \to D_2$

such that

$$f(t) = \begin{cases} E(t_1,\ldots,t_n) & \text{if } t=(t_1,\ldots,t_n,\ldots,t_m) \text{ where } m \geq n \\ & \text{and } x_i \neq \bot \text{ for } 1 \leq i \leq m \\ \bot & \text{otherwise} \end{cases}$$

Remark: I shall omit the type indication ":D_1" if it is clear from context. Thus $(\underline{\lambda}t_1\ldots t_n.E(t_1,\ldots,t_n))(x_1,\ldots,x_m)$ is $\bot$ if $m<n$ or $x_i = \bot$ for some $1 \leq i \leq m$ otherwise it is $E(x_1,\ldots,x_n)$.

Now/

Now (S7) can be modified to:

$$W[\![\lambda[[x_1;\ldots;x_n];e]]\!](r)\underline{\triangle}\,\lambda t_1\ldots t_n.V[\![e]\!]((t_1/x_1)\ldots(t_n/x_n)r)$$

In fact another modification will be forced on us by the steps needed to cope with (3). I describe these steps below.

(3):

This fluid property of LISP varibles is the hardest property of all to handle consistently with I1-I3. To see what to do consider again Ex7 of 3.8.4:

$e=\lambda[[y];fn_3[F]][F]$ where fn_3 is fn defined recursively by:

```
fn[x]=[y->1;
       x->2;
       T->λ[[y];fn[y]][T]]
```

so the value of e is the value of fn[F] when y=F.

What goes wrong in my semantics is that when evaluating fn[F] the free y gets looked up in the environment before the equation is 'solved' (i.e. Y is applied) and this is too early since at that time the binding of y to T by the evaluation of $\lambda[[y];fn[y]][T]$ has not yet occurred. We need to fix things so that Y can be applied <u>before</u> free variables are bound and if after that any variables are still left free their binding can <u>then</u> be done. This sounds rather obscure I know - if/

if you are getting confused skip to the "summing up" below. To continue: in order to be able to plug in an environment after applying Y we need Y's application to yield something of type [Envr->FUN]. Now intuitively (i.e. by I3 of 3.2) to produce the denotation of Label[f;fn], in an environment r, Y should be applied to

$\lambda F{:}FUN.W[\![fn]\!]((F/f)r)$

However, as we have seen we want to put r in after applying Y. To do this I abstract out r, apply Y and then apply the resulting abstraction to r to put it back in. To see what I mean consider the expression:

(A): $Y(\lambda F{:}FUN.\lambda r'{:}\ Envr.W[\![fn]\!]((F/f)r'))(r)$

In this we have just what we want. Y is applied yielding something of type [Envr->FUN] which is then applied to r. There is a snag though; $\lambda F{:}FUN.\lambda r'{:}\ Envr.W[\![fn]\!]((F/f)r')$ is of type FUN->[Envr->FUN] and applying Y to this does not make sense. Now just suppose Envr was of type Id->[Envr->FUN] then we would be O.K. for we would have the expression $\lambda F{:}\ [Envr\text{->}FUN].\lambda r'{:}\ Envr.W[\![fn]\!]((F/f)r')$ which has type [Envr->FUN]->[Envr->FUN] and so Y could be applied to it to yield the required thing of type [Envr->FUN]. The only snag of this move is that it requires that Envr be of type [Id->[Envr->FUN] i.e. that Envr satisfy Envr=[Id->[Envr->FUN]]. Mathematically, as long as we work in Scott's framework and interpret "=" as "is isomorphic" this presents no/

no problems (though in every other framework I know it would ring the death knell!); but more importantly is it consistent with our intuitions about environments? Fortunately it is; in LISP environments are the logical counterparts of alists so we should be able to see what the logical type of environments is by looking to see what alists ought to denote. Consider the alist

```
((F.(LAMBDA(X)(F Y))))
```

Intuitively it binds identifiers so it should be of type [Id->?] - what is "?"? Well, in the example above, it is the type of the denotation of $\lambda[[x];\ f[y]]$ and this, <u>in the absence of knowledge of its (activation) environment</u>, denotes $W[\![\lambda[[x];\ f[y]]]\!] \in$ [Envr->FUN], thus ?=[Envr->FUN] as desired. In 3.8.1 I took ?=FUN (well D actually - but FUN '$\subseteq$' D) and that decision was the cause of my troubles; it was based on too shallow an analysis of the 'meaning' of alists (I was misled by 'λ-calculus intuition'). Thus Envr=[Id->?]=[Id->[Envr->FUN]] - to cope smoothly with form as well as function variables it is necessary to slightly generalize this equation to

Envr=[Id->[Envr->D]]

<u>Summing up</u>

I have shown that to handle fluid variables a new semi-domain of environments is called for. I shall call this Env (to distinguish it from [Id->D] which I shall continue to call Envr) then:

Env/

Env=[Id->[Env->D]]

There are two reasons given above why Env has the right type:

(i) To enable minimal-fixed-point-extraction to be done before variable binding.

(ii) Because fluid-variable-intuition demands that alists denote things of type [Id->[Env->D]].

Taking Env=[Id->[Env->D]] as the semi-domain of environments necessitates changing (S2) and (S6) - the semantic equations for variables - recall that these were:

(S2) $V[\![x]\!](r)=r(x)\mid S$

(S6) $W[\![f]\!](r)=r(f)\mid FUN$

Now if $r \in$ Env then $r(x),r(f) \in$ [Env->D] so $r(x)\mid S$, $r(f)\mid FUN$ do not make sense (and even if they did $V[\![x]\!](r)$,$W[\![f]\!](r)$ would be of the wrong type). We need the expressions on the right hand sides of (S2) and (S6) to be of the form $r(x)(r')\mid S$, $r(f)(r')\mid FUN$ respectively where r' is the environment at the <u>evaluation-time</u> (or <u>activation-time</u>) of x and f - but that is just r so the correct equations are:

(S2) $V[\![x]\!](r)=r(x)(r)\mid S$

(S6) $W[\![f]\!](r)=r(f)(r)\mid FUN$

Now also taking Env=[Id->[Env->D]] renders the equation

(S7)/

(S7) $W[\![\lambda[[x_1;\ldots;x_n];e]]\!](r)=\underline{\lambda}t_1\ldots t_n\text{: }S.V[\![e]\!]((t_1/x_1)\ldots(t_n/x_n)r)$

meaningless since for $(t_1/x_1)\ldots(t_n/x_n)r$ to make sense (if $r \in$ Env) $t_1,\ldots,t_n$ must be of type [Env->D] not of type S. To fix this is easy; clearly the meaning of constants is environment-independent so we can identify $t \in S$ with the constant function $\lambda r.(t \text{ in } D) \in$ [Env->D]. The following definition and conventions make this precise:

3.11.3 <u>Definition</u>

If $r \in$ Env=[Id->[Env->D]], $v \in$ [Env->D] and $z \in$ Id then
$(v/z)r=\lambda z'$: Id. (if $z=\perp$ or $z'=\perp$ then $\perp$ elseif $z=z'$ then v else $r(z')$).

Remark: $(v/z)r \in$ Env. The expression $(v/z)r$ is continuous in v,z,r.

Recall that D=S+FUN.

3.11.4 <u>Conventions</u>

Suppose $z \in$ Id, $r \in$ Env=[Id->[Env->D]] then:

(1) if $t \in S$ then (t in [Env->D])=λr. (t in D)
and $(t/z)r=((t$ in [Env->D]$)/z)r$

(2) if $v \in$ [Env->FUN] then (v in [Env->D])=$\lambda r.(v(r)$ in D)
and $(v/z)r=((v$ in [Env->D]$)/z)r$.

Remark: Notice that if $r \in$ Env, $t \in S$, $v \in$ [Env->FUN] then
((t in [Env->D])(r) | S)=((t in D) | S)=t
((v in [Env->D])(r) | FUN)=(($v(r)$ in D) | FUN)=$v(r)$.

3.12/

3.12 How interpreters do recursion

The purpose of this section is to motivate my addition of expressions of the form μ[f;fn] to LISP.

If you examine how apply treats Label expressions you will find that:

apply[(LABEL F fn*);x;a]=apply[fn*;x;((F.fn*).a)]

I can mimic this 'semantically' with the equation:

(A) $W'[\![\mathrm{Label}[f;fn]]\!](r)=W'[\![fn]\!]((W'[\![fn]\!]/f)r)$

Now this equation does not explain recursion in terms of solving equations, i.e. in terms of Y, so it fails to reflect intuition I3. However, it certainly does not defy intuition;indeed, to some extent, it formalizes the operational intuition we bring to bear when understanding the definition of apply. An interesting (and, as it turns out, important) question is: do the two analyses of recursion give the same results? The answer is a qualified "yes" but I shall not elaborate the qualifications here - they are dealt with in 4.6. To make the study of this question smooth I shall adjoin to LISP a new type of expression, μ[f;fn], then μ[f;fn] will get a fixed point analysis and Label[f;fn] one as in (A) above. I have allocated meanings to μ and Label this way round firstly because for a lot of technical work later I want to use analysis (A) and secondly because, by convention, the symbol μ seems to adhere rather strongly to fixpointing/

fixpointing operations.

The fact that I need to use (A) for certain purposes shows that by applying Mao Tse Tung thought and resisting monsterbarring (move 1 of 3.9) and my theoretician's arrogance (move 2) I have not only entered reality but also given myself a powerful tool - a tool which is not only useful for dealing with essentially fluid variables (monsters) but also for simpler cases which could be handled correctly by the semantics of 3.8. Notice that (A), for its formation, requires Env to be of type [Id->[Env->D]], thus we could not even formulate the question of whether (A) is understandable in terms of Y if Env=[Id->D] were taken. (A) is a more interpreter-like modelling of recursion and studying it sheds light on the 'semantics' of interpreting - in particular on the use of fluid variables for implementing recursion.

3.13 Syntax of pure LISP

The syntax I am going to use from now on is (in the notation of 3.7.14):

Syntax

Meta variables

A	ranges	over	<S-expression>
x,f	"	"	<identifier>
e	"	"	<form>
F/			

F ranges over <standard function>

fn " " <function>

Syntax equations

$$e ::= A \mid x \mid fn[e_1;\ldots;e_n] \mid [e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}]$$

$$F ::= car \mid cdr \mid cons \mid atom \mid eq$$

$$fn ::= F \mid f \mid \lambda[[x_1;\ldots;x_n;e] \mid Label[f;fn] \mid \mu[f;fn]$$

3.14 New improved semantics of pure LISP

The semantics below should be read in conjunction with the notes that follow it. I make use of Definition 3.11.3 and convention 3.11.4.

Semantics

Denotations: D=S+FUN

S=flat(<S-expression>)

FUN=[S*->S]

Environments: Env=[Id->[Env->D]] (see note 1)

Semantic functions: $e \mapsto [\![e]\!]$: <form>->[Env->S]

$fn \mapsto [\![fn]\!]$: <function>->[Env->FUN] (see note 2)

Semantic/

Semantic equations:

(S1) $[\![A]\!](r)=A$

(S2) $[\![x]\!](r)=r(x)(r) \mid S$

(S3) $[\![fn[e_1;\ldots;e_n]]\!](r)=[\![fn]\!](r)([\![e_1]\!](r),\ldots,[\![e_n]\!](r))$

(S4) $[\![[e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}]]\!](r)=([\![e_{11}]\!](r)\rightarrow [\![e_{12}]\!](r),\ldots,[\![e_{n1}]\!](r)\rightarrow [\![e_{n2}]\!](r))$

(S5)
$$\left.\begin{array}{l} [\![car]\!](r)= \underline{\lambda} t.car(t) \\ [\![cdr]\!](r)= \underline{\lambda} t.cdr(t) \\ [\![cons]\!](r)= \underline{\lambda} t_1 t_2.cons(t_1,t_2) \\ [\![atom]\!](r)= \underline{\lambda} t.atom(t) \\ [\![eq]\!](r)= \underline{\lambda} t_1 t_2.eq(t_1,t_2) \end{array}\right\}\text{(see note 3)}$$

(S6) $[\![f]\!](r)=r(f)(r) \mid FUN$

(S7) $[\![\lambda[[x_1;\ldots;x_n];e]]\!](r)= \underline{\lambda} t_1\ldots t_n: S[\![e]\!]((t_1/x_1)\ldots(t_n/x_n)r)$

(S8) $[\![Label[f;fn]]\!](r)=[\![fn]\!](([\![fn]\!]/f)r)$

(S9) $[\![\mu[f;fn]]\!](r)=Y(\lambda v: [Env\to FUN].\lambda r': Env.[\![fn]\!]((v/f)r'))$

(see note 4)

Notes

Note 1: To state the semantics I just need to know the type of Env i.e. that if $r,r' \in Env$ and $z \in Id$ then $r(z)(r')$ makes sense and is in D. Solutions of the equation Env=[Id->[Env->D]] are semi-domains, Env, such that Env and [Id->[Env->D]] are isomorphic. Thus if Env is such a solution and f: Env->[Id->[Env->D]], g: [Id->[Env->D]]->Env are the isomorphisms then $r(z)(r')$ is 'really'/

'really' $f(r)(z)(r')$ and λz: Id.$r(z)$ is identified with $g(\lambda z$: Id.$r(z))$. Because of these identifications I write Env=[Id->[Env->D]] rather than Env≅[Id->[Env->D]]. The equation for Env does not, a priori, fully specify it up to isomorphism, thus there may be many non-isomorphic solutions each yielding a distinct semantics. In section 6.7. I construct the 'obvious' minimal solution to Env=Id->[Env->D] which is the one I intend, reasons and more details of this choice are given in 5.2.

Note 2: I have not named the semantic functions explicitly (alternatively: I have used invisible symbols for them!), unless otherwise indicated I shall reserve V,W for those functions defined by (S1)-(S8) of 3.8.3.

Note 3: I shall use car,cdr,cons,atom,eq as names for $\lambda t.\mathrm{car}(t)$, $\lambda t.\mathrm{cdr}(t)$, $\lambda t_1 t_2.\mathrm{cons}(t_1,t_2)$, $\lambda t.\mathrm{atom}(t)$, $\lambda t_1 t_2.\mathrm{eq}(t_1,t_2)$ respectively. Thus cons(1,NIL,NIL) makes sense and means cons(1,NIL).

Note 4: Using the fixedpoint property of Y one can derive from (S9) the equation of which $[\![\mu[f;fn]]\!]$ is a least solution:

[/

$$\begin{aligned}[\![\mu[f;fn]]\!] &= Y(\lambda v.\lambda r'.[\![fn]\!]((v/f)r')) \\ &= (\lambda v.\lambda r'.[\![fn]\!]((v/f)r'))(Y(\lambda v.\lambda r'.[\![fn]\!]((v/f)r'))) \\ &\qquad\qquad (\text{as } Y(F)=F(Y(F))) \\ &= (\lambda v.\lambda r'.[\![fn]\!]((v/f)r'))([\![\mu[f;fn]]\!]) \\ &= \lambda r'.[\![fn]\!](([\![\mu[f;fn]]\!]/f)r')\end{aligned}$$

hence (S10): $[\![\mu[f;fn]]\!](r)=[\![fn]\!](([\![\mu[f;fn]]\!]/f)r)$

This last equation (S10) looks like (S8) except that $([\![fn]\!]/f)$ is replaced by $([\![\mu[f;fn]]\!]/f)$. Note that $[\![Label[f;fn]]\!]$ is not obviously a fixedpoint of (S10). I don't know in fact whether it is or not (see 8.3). If $fn=\lambda[[x_1;\ldots;x_n];e]$ then from (S10) we have:

$$[\![\mu[f,fn]]\!](r)(A_1,\ldots,A_n)=[\![e]\!]((A_1/x_1)\ldots(A_n/x_n)([\![\mu[f;fn]]\!]/f)r)$$

This shows that μ is analysed semantically in a way harmonious with intuition I3 of 3.2.

3.15 Example calculation of denotations using the new improved semantics

I now apply the new improved semantics to examples 1-7 of 3.8.4 to show that they get the rignt meaning. I shall use whichever of μ or Label is convenient - the fact that sometimes it is one and sometimes the other shows that both μ and Label are useful.

Example/

Example 1 e=cons[1;NIL;NIL]

$[\![e]\!](r)=(\underline{\lambda}t_1t_2.\mathrm{cons}(t_1,t_2))(1,\mathrm{NIL},\mathrm{NIL})$

$=\mathrm{cons}(1,\mathrm{NIL})$

$=(1)$

Example 2 fn= μ[f;f]

$[\![fn]\!]=Y(\lambda v.\lambda r'.[\![f]\!]((v/f)r'))$

$=Y(\lambda v.\lambda r'.v((v/f)r'))$

$=\bigsqcup_{n=0}^{\infty}(\lambda v.\lambda r'.((v/f)r'))^n(\perp)$

$=\perp$ (as for all n, $(\lambda v.\lambda r'.v((v/f)r'))^n(\perp)=\perp$)

Example 3 e= μ[f;f][NIL]

$[\![e]\!](r)=\perp(r)(\mathrm{NIL})=\perp$

Example 4 e= λ[[x];NIL][μ[f;f][NIL]]

$[\![e]\!](r)=[\![\lambda[[x];\mathrm{NIL}]]\!](r)(\perp)$

$=(\underline{\lambda}t.\mathrm{NIL})(\perp)$

$=\perp$ (by definition of $\underline{\lambda}$)

Example 5 $fn_2=\lambda$[[x];[y→ 1;x→ 2;T→λ[[y];fn[y]][T]]

(fn$_2$ was previously defined in 3.1, fn ∊ <identifier>)

$[\![fn_2]\!](r)=\underline{\lambda}t.[\![[y\rightarrow 1;x\rightarrow 2;T\rightarrow\lambda[[y];fn[y]][T]]]\!](r')$

where $r'=(t/x)r=((\lambda r.(t \text{ in } [Env\rightarrow D]))/x)r$

(by S7 and convention 3.11.4)

=/

$$= \underline{\lambda} t.(r(y)(r') \mid S \rightarrow 1,$$
$$t \rightarrow 2,$$
$$T \rightarrow (\underline{\lambda} t'.[\![fn[y]]\!]((t'/y)r'))(T))$$

$$= \underline{\lambda} t.(r(y)(r') \mid S \rightarrow 1,$$
$$t \rightarrow 2,$$
$$T \rightarrow [\![fn[y]]\!]((T/y)r'))$$

$$= \underline{\lambda} t.(r(y)(r') \mid S \rightarrow 1,$$
$$t \rightarrow 2,$$
$$T \rightarrow r(fn)(r'')(T))$$
$$\text{where } r''=(T/y)(t/x)r$$

<u>Example 6</u> fn_3=Label[$fn;fn_2$] (fn_2 as above)

$$[\![fn_3]\!](r)=[\![fn_2]\!](([\![fn_2]\!]/fn)r)$$
$$= \underline{\lambda} t.(r(y)(r') \mid S \rightarrow 1,$$
$$t \rightarrow 2,$$
$$T \rightarrow [\![fn_2]\!](r'')(T))$$
$$\text{where } r''=(T/y)r'$$
$$r'=(t/x)([\![fn_2]\!]/fn)r$$

$$= \underline{\lambda} t.(r(y)(r') \mid S \rightarrow 1,$$
$$t \rightarrow 2,$$
$$T \rightarrow 1\ )$$
(by example 5 $[\![fn_2]\!](r'')(T)=1$)

<u>Example/</u>

<u>Example 7</u> $e=\lambda[[y];fn_3[F]][F]$

$$[\![e]\!](r)=(\underline{\lambda}t.[\![fn_3]\!]((t/y)r)(F))(F)$$

$$=[\![fn_3]\!]((F/y)r)(F)$$

$$=1 \qquad \text{(by example 6')}$$

Thus the examples which caused trouble for the semantics of 3.8 now get the right denotations. Also, as these examples illustrate, it is not much more difficult to work out denotations with the new semantics, than it was with the old one, this reflects the fact that properties I1-I3 are retained. I have not yet shown that if Label (rather than μ) is always used then these examples still get the right denotations, this is because to work out things like $[\![$Label[f;f]$]\!]$ I need to know more about Env than its type. Examination of the detailed properties of Env is made in 5.2.

3.16 <u>Why I have assigned the same meaning to errors and non-termination</u>

The semantics just given has the property that forms whose evaluation does not terminate (e.g. μ[f;f][NIL]) get the same denotation (viz. $\perp$) as forms whose evaluations lead to an error (e.g. cons[NIL]). One might feel that errors and unending computations should be distinguished semantically, but if one felt that one might then feel that different kinds of errors should be distinguished or that non-termination of a subcomputation should be distinguished from non-termination of the main computation. One has to draw the line somewhere and I have chosen to draw/

draw it between well-behaved programs on the one hand and ill-behaved ones (whether the bad behaviour be due to non-termination or error) on the other. Thus my semantics is not intended to be defined on badly behaved programs so they all get lumped together and assigned $\perp$ as their denotation.

4. IS THE SEMANTICS RIGHT?

4.1 Introduction

I got the definition of $[\![e]\!],[\![fn]\!]$ by 'debugging' the definitions of $V[\![e]\!],W[\![fn]\!]$ on examples, I now turn to the question of whether this debugging has been completely successful. What I would like to do is to take a well known LISP implementation and prove that it computes the same values as my semantic equations. Unfortunately that task is beyond me and I am going to have to perform a less reliable analysis. What I have tried to do is to formalize the essential algorithm embodied in many actual interpreters and to compare that with my semantics. In order to make things mathematically tractable for me I have had to express this "essential algorithm" in a rather abstract form and this makes the gap between real algorithms and mine rather wide. There is another, related, test I shall apply to my semantics and that is to see if the functions denoted by the M-expressions defining apply,eval,...etc. compute the same values as my semantics i.e. whether

$$\text{(A)} \quad [\![apply[fn^*;(A_1\ldots A_n);a]]\!](r_0)=[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)$$

$$\text{(B)} \quad [\![eval[e^*;a]]\!](r_0)=[\![e]\!]([\![a]\!])$$

where fn*,e* are the S-expression codings of fn and e, $[\![a]\!]$ is the environment naturally associated with a and r_0 contains the definitions of apply,eval...etc. The precise formulation and answering of this question/

question is done in 4.8, however, I feel that this latter test is less reliable than the former because although (A) and (B) might be true both sides of the equations could be wrong but the errors cancel out (e.g. if both sides of (A) and (B) were always $\perp$). People who have implemented apply,eval....etc. have probably understood the M-expression definitions of these functions with their operational intuition rather than their denotational one. Thus to get the spirit of the manual interpreter it is better to formalize directly what is intended rather than to try and get a precise meaning via a possibly wrong semantics of LISP.

In addition to its role in comparing my semantic equations with reality, the abstract 'implementation' I am about to describe is also a very useful aid to their mathematical analysis. This is because it formalizes our (i.e. my!) operational intuitions about LISP and so makes that available for use in generating rigorous proofs. Without such an aid many operationally obvious results are rather hard to prove. In 4.4 I describe a type of argument which I call "LISP-induction" and which, <u>intuitively</u>, is induction on the length of computation. Now length of computation is an <u>operational</u> notion and starting from semantic equations it is not clear what it corresponds to formally. Solving this problem is one of the useful roles of my abstract implementation - which I now get down to describing.

4.2 <u>A formalization of some operational intuitions about LISP</u>

I/

I am going to describe a calculus whose conversion rules are intended to formalize the evaluation process described in the LISP 1.5 Manual. At the end of 3.8.4, as well as in 4.1, I said why I did not find the definitions of apply, eval,...etc. an adequate formalization.

To form the terms of this calculus I need a notation for denoting environments. I could use ordinary alists (as in the Manual) but this is messy because it requires all M-expressions to be coded into S-expressions. Instead I have invented a little language called ALIST.

Syntax of ALIST

Meta variables:	a	ranges	over	<alist>	(see note 1 below)
	A	"	"	<S-expression>	
	z	"	"	<identifier>	
	fn	"	"	<function>	

Syntax equation: a ::= NIL | (A/z)a|(fn/z)a (see note 2 below)

Semantics of ALIST

Denotations: Env

Semantic function: $a \mapsto [\![a]\!]$: <alist>->Env

Semantic/

Semantic equations:

(AS1) $[\![\mathrm{NIL}]\!] = \perp$ (see note 3 below)

(AS2) $[\![(A/z)a]\!] = (A/z)[\![a]\!]$

(AS3) $[\![(fn/z)a]\!] = ([\![fn]\!]/z)[\![a]\!]$

Notes

Note 1: <S-expression>, <identifier> and <function> are as defined in 3.7.15 and 3.13.

Note 2: I shall abbreviate $(E_1/z_1),\ldots,(E_n/z_n)\mathrm{NIL}$ by $(E_1/z_1)\ldots(E_n/z_n)$ so for example (fn/f)(A/x) means (fn/f)(A/x)NIL. I may use x,f as well as z to range over <identifier>.

Note 3: I use conventions 3.11.4 in these semantic equations.

4.2.1 Definition

If a ϵ <alist>, z' ϵ <identifier> then a(z') is defined by structural induction on a as follows:

1. $\mathrm{NIL}(z') = \perp$
2. ((A/z)a)(z')=if z=z' then A else a(z')
3. ((fn/z)a)(z')=if z=z' then fn else a(z')

Remark: Thus $a(z) \in \{\perp\} \cup$ <S-expression> $\cup$ <function>. Let $[\![a(z)]\!]$ be $\perp_{[Env\to D]}$, (A in [Env->D]),($[\![fn]\!]$ in [Env->S]) according as a (z) is $\perp$, A, fn so that $[\![a]\!](z)=[\![a(z)]\!]$ (a quick structural/

structural induction on a proves this).

Now let the meta variable p range over the set <term> where this is defined by:

$$p ::= A \mid (e,a)$$

I am going to define a binary relation -> on <term> . p->p' (read "p immediately reduces to p'") is intended to mean that, using the algorithm implicit in the manual interpreter, p can be converted to p'. If p's are thought of as states of an abstract machine then p->p' means that p' is the state immediately following p in any computation. Before giving the formal definition here are some examples of such computations.

```
(1):  (car[cdr[(1 2)]],NIL)
      (car[(2)],NIL)
      2.

(2):  (λ[[x];x][(1 2)],NIL)
      (x,((1 2)/x)NIL)
      ((1 2),((1 2)/x)NIL)
      (1 2)

(3):  (λ[[x];car[cdr[x]]][(1 2)],NIL)
      ( car[cdr[x]],((1 2)/x)NIL)
      (car[(2)],((1 2)/x)NIL)
      2.
```

The/

The individual steps in such computations will satisfy -> and *> will denote the reflexive, transitive closure of ->. ->, *> are defined by structural induction below; I shall first give the definition then immediately follow it with a description of the notation in which it is written.

4.2.2 Definition

Define ->, *> by:

P1. $(A,a)\rightarrow A$

P2. $$\frac{a(x)=A}{(x,a)\rightarrow A}$$ (See note 1 below)

P3. $$\frac{F(A_1,\dots,A_n)=A}{(F[A_1;\dots;A_n],a)\rightarrow A}$$

P4. $$\frac{(\forall i.(e_i,a)\overset{*}{\rightarrow}A_i)\text{ and }(\exists i.e_i\neq A_i)}{(fn[e_1;\dots;e_n],a)\rightarrow(fn[A_1;\dots;A_n],a)}$$ (See note 2 below)

P5. $$\frac{(e_{m1},a)\overset{*}{\rightarrow}T\text{ and }\forall i<m.(e_{i1},a)\overset{*}{\rightarrow}F}{([e_{11}\rightarrow e_{12};\dots;e_{n1}\rightarrow e_{n2}],a)\rightarrow(e_{m2},a)}$$

P6. $$\frac{a(f)=fn}{(f[A_1;\dots;A_n],a)\rightarrow(fn[A_1;\dots;A_n],a)}$$ (See note 3 below)

P7. $$\frac{m\leq n}{(\lambda[[x_1;\dots;x_m];e][A_1;\dots;A_n],a)\rightarrow(e,(A_1/x_1)\dots(A_m/x_m)a)}$$

P8. $(\mathrm{Label}[f;fn][A_1;\dots;A_n],a)\rightarrow(fn[A_1;\dots;A_n],(fn/f)a)$

P9. $(\mu[f;fn][A_1;\dots;A_n],a)\rightarrow(fn[A_1;\dots;A_n],(\mu[f;fn]/f)a)$

P10. $$\frac{1\leq n,\text{þ}_1\rightarrow\text{þ}_2\rightarrow\text{þ}_3,\dots,\text{þ}_{n-1}\rightarrow\text{þ}_n}{\text{þ}_1\overset{*}{\rightarrow}\text{þ}_n}$$ (See note 4 below)

This definition should be largely self explanatory but in case it is not here is a brief explanation. Each clause P1-P10 is a schema. The meta variables in them range over the sets previously assigned to them viz:

A/

A	ranges	over	<S-expression>	(as in 3.7.15)
a	"	"	<alist>	(as above)
x,f	"	"	<identifier>	(as in 3.7.15)
F	"	"	<standard function>={car,cdr,cons,atom,eq}	
e	"	"	<form>	(as in 3.13)
fn	"	"	<function>	(as in 3.13)
p	"	"	<term>	(as above)

A schema of the form p->p' means that any instance of it is a pair for which -> holds.

A schema of the form

$$\frac{\underline{\text{conditions}}}{\text{p->p'}}$$

is a kind of rule of inference. It means that any instance of p->p' which satisfies the conditions is a pair for which -> holds.

Each step in the example computations (1), (2), (3) above satisfy -> e.g.

(car[cdr[(1 2)]],NIL)->(car[(2)],NIL)->2.

Remark: -> is deterministic in the sense that p->p' and p->p" => p'=p".

Note 1: The reason I use P2 rather than (x,a)->a(x) is that if a(x)= ⊥ or a(x)=fn then (x,a)-> ⊥ or (x,a)->fn and I do not want this.

Note/

Note 2: The reason I have the condition "$(\exists i. e_i \neq A_i)$" in P4 is to exclude unending computations of the form

$(fn[A_1;\ldots;A_n],a) \to (fn[A_1;\ldots;A_n],a) \to \ldots$

and also to make $\to$ deterministic e.g. I do not want:

$(Label[f;fn][A_1;\ldots;A_n],a) \to (fn[A_1;\ldots;A_n],(fn/f)a)$ and

$(Label[f;fn][A_1;\ldots;A_n],a) \to (Label[f;fn][A_1;\ldots;A_n],a)$.

Note 3: $(f[A_1;\ldots;A_n],a) \to (a(f)[A_1;\ldots;A_n],a)$ will not do for P6 because of the possibility that $a(f)=\perp$ or $a(f)=A$ (c.f. Note 1).

Note 4: Taking n=1 in P10 yields $p \overset{*}{\to} p$ for all p.

I think that if you look at the interpreter in the Manual you will see that the definition of $\to$, $\overset{*}{\to}$ embody the essential idea there. If you utterly disagree with this do not give up yet. I shall show soon that $\to$ is a powerful technical device for studying $[\![\ldots]\!]$ and is needed to investigate the connection between μ and Label and the relation between $[\![\ldots]\!]$ and apply, eval...etc. Think of $\to$ as an abstract tool if you feel I have 'puffed up' its intuitive significance.

4.3 Statement of Main Theorem connecting operational and denotational semantics

Here are four questions concerning the agreement of $\to$, $\overset{*}{\to}$ and $[\![\ ]\!]$.

Question 1: If $(e,a)\overset{*}{\to}A$ then does $[\![e]\!]([\![a]\!])=A$?

Question 2: If for no A: $(e,a)\overset{*}{\to}A$ then does $[\![e]\!]([\![a]\!])=\perp$?

Question 3/

Question 3: If $[\![e]\!]([\![a]\!])=A$ then does $(e,a)\overset{*}{>}A$?

Question 4: If $[\![e]\!]([\![a]\!])=\perp$ then is there no A such that $(e,a)\overset{*}{>}A$?

The answer to all those is "yes". Using Wadsworth's beautiful techniques [25] the process of answering them is straightforward. In chapter 5 I do this but first I shall formulate a theorem containing the answers and then draw some consequences from it.

4.3.1 Definition (extension of $[\![\ldots]\!]$ to terms)

For $p \in$ <term> $[\![p]\!] \in S$ is defined by:

1. $[\![A]\!]=A$
2. $[\![(e,a)]\!]=[\![e]\!]([\![a]\!])$

4.3.2 Main Theorem

$p\overset{*}{>}A \Leftrightarrow [\![p]\!]=A$

Proof

The proof of the Main Theorem is the goal of chapter 5.

4.3.3 Corollary

The answers to questions 1-4 are "yes".

Proof

Question 1 and question 3 follow directly from the Main Theorem. Question 2 yields true because if for no A: $(e,a)\overset{*}{>}A$ and if $[\![e]\!]([\![a]\!])\neq\perp$ then/

then as S=flat(<S-expression>)$[\![e]\!]([\![a]\!])$=A for some A $\in$ <S-expression>, but then by the Main Theorem $(e,a)\overset{*}{\Rightarrow}A$ contradicting the assumption. Question 4 yields true because if $(e,a)\overset{*}{\Rightarrow}A$ then $[\![e]\!]([\![a]\!])=A\neq\perp$

Q.E.D.

I shall now illustrate how the calculus enables operational reasoning to be applied to the denotational semantics.

Consider the evaluation of expressions of the form $fn[e_1;\ldots;e_n]$: First the evaluations of the e_i's are attempted so if one of these fails to terminate (or leads to an error) then so does that of $fn[e_1;\ldots;e_n]$. Now in the semantics a form's evaluation failing to terminate (or leading to an error) is modelled by the form denoting $\perp$ so from the above remark we would expect that:

e_i denotes $\perp$ => $fn[e_1;\ldots;e_n]$ denotes $\perp$.

This reasoning was operational but using Corollary 4.3.3 it can be justified: if $[\![e_i]\!]([\![a]\!])=\perp$ then, as the answer to question 4 is "yes", there is no A_i such that $(e_i,a)\overset{*}{\Rightarrow}A_i$, but then by P4 there is no A such that $(fn[e_1;\ldots;e_n],a)\overset{*}{\Rightarrow}A$ and so as the answer to question 2 is "yes", $[\![fn[e_1;\ldots;e_n]]\!]([\![a]\!])=\perp$.

The next corollary sums up what I have just said.

4.3.4 <u>Corollary</u>

If/

If $t_1,\ldots,t_n \in S$ and for some m $t_m = \perp$ $(1 \leq m \leq n)$ then for any fn $\in$ <function>, a $\in$ <alist>:

$$[\![fn]\!]([\![a]\!])(t_1,\ldots,t_n) = \perp .$$

Proof

Let $e_1,\ldots,e_n \in$ <form> be such that $[\![e_i]\!]([\![a]\!]) = t_i$ e.g.

$$e_i = \begin{cases} t_i & \text{if } t_i \neq \perp \\ \mu[f;f][NIL] & \text{if } t_i = \perp \end{cases}$$

(this works by example 3 of 3.15 - $e_i = t_i$ does not work as $t_m = \perp \notin$ <form>)

Then $[\![fn]\!]([\![a]\!])(t_1,\ldots,t_n) = [\![fn]\!]([\![a]\!])([\![e_1]\!]([\![a]\!]),\ldots,[\![e_n]\!]([\![a]\!]))$

$= [\![fn[e_1;\ldots;e_n]]\!]([\![a]\!])$ (by the semantic equations)

$= \perp$ (by the reasoning in the remarks before this corollary)

Q.E.D.

4.4 LISP-induction: an inference rule for applying the Main Theorem

Many consequences of the Main Theorem which I want to deduce are statements of the form:

$$\forall p \,.\, p \overset{*}{>} A \Rightarrow R(p,A)$$

where/

where R is some relation* e.g. $R(p,A) \Leftrightarrow [\![p]\!]=A$.

Such statements can usually be proved by induction on the size of the computation from p to A. These proofs all have the same basic structure and to show what this is I define a relation $<^*$, where informally:

$p' <^* p \Leftrightarrow p'$ has to be evaluated in the course of evaluating p (I shall give a precise definition of $<^*$ in 4.4.1 below). Then, in a sense, the set $\{p' \mid p' <^* p\}$ is the computation induced by p and so should be finite if that computation terminates e.g. if $p \overset{*}{>} A$. Thus to prove $\forall p.(p \overset{*}{>} A \Rightarrow R(p,A))$ it suffices to prove it for p's such that $\{p' \mid p' <^* p\}$ is finite and so it suffices to prove BASE and INDUCTION where:

BASE: p minimal for $<^* \Rightarrow (p \overset{*}{>} A \Rightarrow R(p,A))$

INDUCTION: $(\forall p' <^* p.(p' \overset{*}{>} A' \Rightarrow R(p',A'))) \Rightarrow (p \overset{*}{>} A \Rightarrow R(p,A))$

In fact INDUCTION => BASE (take p minimal in INDUCTION) but I shall

---

* Notice that since $p \overset{*}{>} A$, $p \overset{*}{>} A' \Rightarrow A=A'$ a statement of the form $(p \overset{*}{>} A \Rightarrow R(p,A))$ is equivalent to

$((\exists A.p \overset{*}{>} A) \Rightarrow \underline{P}(p))$ where

$\underline{P}(p) \Leftrightarrow (\exists A.p \overset{*}{>} A$ and $R(p,A))$. I use the former form rather than the latter because it enables proofs to be laid out slightly more neatly.

shortly refine BASE and INDUCTION and then this will not be so. Before describing this I need to give a precise definition of $<^*$. I first define a relation $<$ of which $<^*$ is the transitive closure.

4.4.1 Definition

Define $<$ and $<^*$, binary relations on <term> by:

$p' < p \iff$ (1) $p \rightarrow p'$

<u>or</u> (2) $p=(fn[e_1;\ldots;e_n],a)$
and $p'=(e_i,a)$ some $1\leq i\leq n$

<u>or</u> (3) $p=([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}],a)$,
$p' \in \{(e_{11},a),\ldots,(e_{m1},a)\}$ some $1\leq m\leq n$
and $(e_{m1},a)\overset{*}{\Rightarrow}T$ and $\forall\, i<m.(e_{i1},a)\overset{*}{\Rightarrow}F$

$p' <^* p \iff p'=p_1<p_2<\ldots<p_n=p$ for some $p_1,\ldots,p_n$ $(n>1)$

Remark: I only need the relation $<^*$, $<$ was just used to aid in defining $<^*$. I shall want to use the symbol $<$ later with a completely different meaning. $p \;{}^*\!\!> p'$ means $p' <^* p$. Note that $<^*$ is <u>not</u> reflexive. $p' \leq^* p$ means $p' <^* p$ or $p'=p$.

A little thought will show (I hope) that $<^*$ corresponds to its intuitive meaning given earlier viz:

$p' <^* p \iff p'$ has to be evaluated in the course of evaluating p

Call/

Call p <u>well-founded</u> if $\{p' \mid p' \leq^* p\}$ is finite, then lemma 4.4.3 below can be interpreted as showing that if p*>A then A can be effectively computed from p (i.e. the computation of A from p is finite i.e. it terminates) this observation is important as it shows that -> does constitute a useable interpreter for pure LISP - it could be implemented on a real computer (though, of course, for practical purposes it would not be sensible to do so).

Before proving lemma 4.4.3 I need to clarify exactly what the "size" of a computation p*>A really is. To do this close scrutiny of definition 4.2.2 is called for.

As is usual with induction definitions it is the <u>least</u> relation which satisfies the conditions that is wanted. In view of clausesP4 and P5 of definition 4.2.2, exactly what this least solution is may not be immediately obvious. Because of this I shall now exhibit more explicitly what -> and *> are. I am going to define relations $\underset{n}{\rightarrow}$, $\underset{n}{\overset{*}{>}}$ inductively for each $n \geq 0$; $p \underset{n}{\rightarrow} p'$ will mean p->p' is deducible using at most n 'recursive calls' of clauses P4, P5. $\underset{n}{\overset{*}{>}}$ is the reflexive, transitive closure of $\underset{n}{\rightarrow}$. Then

$$p \rightarrow p' \iff \exists n. p \underset{n}{\rightarrow} p' \text{ and } p \overset{*}{>} p' \iff \exists n. p \underset{n}{\overset{*}{>}} p'$$

and the size of a computation from p to A can be measured by the pair (m,n) where

p/

$p=p_1 \underset{n}{\rightarrow} p_2 \underset{n}{\rightarrow} \dots \underset{n}{\rightarrow} p_m = A$

Arguments by induction on the size of computation are messy double inductions on (m,n), one of the functions of LISP-induction is to disguise this messiness. Showing LISP-induction valid (i.e. proving lemma 4.4.3) is messy but applying it is not - fortunately validity only has to be proved once and so by formulating LISP-induction we can factor out the messiness.

4.4.2 <u>Definition</u>

Define $\underset{n}{\rightarrow}$, $\underset{n}{\overset{*}{\rightarrow}}$ for $n \geq 0$ by induction on n as follows:

$p \underset{0}{\rightarrow} p' \iff p \rightarrow p'$ follows from P1, P2, P3, P6, P7, P8, P9 of definition 4.2.2.

$p \underset{0}{\overset{*}{\rightarrow}} p' \iff p=p_1 \underset{0}{\rightarrow} p_2 \underset{0}{\rightarrow} \dots \underset{0}{\rightarrow} p_m = p'$ for some $p_1, p_2, \dots, p_m$ $(m \geq 1)$

$p \underset{n+1}{\rightarrow} p' \iff$ (1) $p \underset{n}{\rightarrow} p'$

<u>or</u> (2) $p=(fn[e_1;\dots;e_n],a)$
$p'=(fn[A_1;\dots;A_n],a)$
and $\forall i.(e_i,a) \underset{n}{\overset{*}{\rightarrow}} A_i$

<u>or</u> (3) $p=([e_{11} \rightarrow e_{12};\dots;e_{s1} \rightarrow e_{s2}],a)$
$p'=(e_{m2},a)$ $(1 \leq m \leq s)$
and $(e_{m1},a) \underset{n}{\overset{*}{\rightarrow}} T$ and $\forall i<m.(e_{i1},a) \underset{n}{\overset{*}{\rightarrow}} F$.

$p \underset{n+1}{\overset{*}{\rightarrow}} p' \iff p=p_1 \underset{n+1}{\rightarrow} p_2 \underset{n+1}{\rightarrow} \dots \underset{n+1}{\rightarrow} p_m = p'$ for some $p_1, p_2, \dots, p_m$ $(m \geq 1)$

Then/

Then $p \rightarrow p' \iff \exists n. p \underset{n}{\rightarrow} p'$

$p \overset{*}{\rightarrow} p' \iff \exists n. p \underset{n}{\overset{*}{\rightarrow}} p'$

Clearly $\rightarrow$, $\overset{*}{\rightarrow}$ so defined are the least relations satisfying P1-P10. Although you might feel that explicitly introducing $\underset{n}{\rightarrow}$, $\underset{n}{\overset{*}{\rightarrow}}$ is unnecessary verbosity, I found that until I did so I often got confused. The size of a computation $p_1 \underset{n}{\rightarrow} p_2 \underset{n}{\rightarrow} \dots \underset{n}{\rightarrow} p_m$ depends on both n and m so arguments by induction on the size of computation (e.g. the proof of 4.4.3 below) are difficult to formulate if n is not in sight.

4.4.3 <u>Lemma</u>

$p \overset{*}{\rightarrow} A \Rightarrow p$ well-founded

<u>Proof</u>

The lemma is trivial if $p=A$ so assume $p \neq A$.

If $p \overset{*}{\rightarrow} A$ then $p \underset{n}{\overset{*}{\rightarrow}} A$ for some n. I shall do induction on n.

n=0: Then:

$p=p_1 \underset{o}{\rightarrow} p_2 \underset{o}{\rightarrow} \dots \underset{o}{\rightarrow} p_m = A$ and $\{p' \mid p' \leq^* p\} = \{p_1, \dots, p_m\}$

n>0: Assume true for (n-1). For some m>0:

$p=p_1 \underset{n}{\rightarrow} p_2 \underset{n}{\rightarrow} \dots \underset{n}{\rightarrow} p_m = A$.

Call p_i <u>special</u> if it is of the form $(fn[e_1;\dots;e_n],a)$ where $e_i \neq A_i$ for some i <u>or</u> of the form $([e_{11} \rightarrow e_{12};\dots;e_{s1} \rightarrow e_{s2}],a)$. I shall do induction/

induction on the number of special p_i's. If there are none then $p_o \overset{*}{>} A$ and so by induction p is well-founded. Otherwise let p_j be the first special term (i.e. p_j is special and $\forall i<j.p_i$ is not special).

case 1: $p_j=(fn[e_1;\ldots;e_n],a)$. $\forall i.(e_i,a) \overset{*}{>}_{n-1} A_i$

$p_{j+1}=(fn[A_1;\ldots;A_n],a)$ (since $\exists i.e_i \neq A_i$)

Then:

$\{p' \mid p' \leq^* p\}=\{p_i \mid 1\leq i\leq j \} \cup \bigcup_{i=}^{n}\{p' \mid p' \leq^* (e_i,a)\} \cup \{p' \mid p' \leq^* p_{j+1}\}$

which is finite since $\{p_i \mid 1\leq i\leq j \}$ is finite, each (e_i,a) is well-founded by induction on n and p_{j+1} is well-founded by induction on the number of special terms.

case 2: $p_j=([e_{11}\rightarrow e_{12};\ldots;e_{s1}\rightarrow e_{s2}].a)$

$p_{j+1}=(e_{m2},a)$

and $(e_{m1},a) \overset{*}{>}_{n-1}$ T and $\forall i<m.(e_{i1},a) \overset{*}{>}_{n-1}$ F

Then $\{p' \mid p' \leq^* p\}=\{p_i \mid 1\leq i\leq j \} \cup \bigcup_{i=1}^{n}\{p' \mid p' \leq^* e_{i1}\} \cup \{p' \mid p' \leq^* p_{j+1}\}$

which is finite by induction on n and the number of special terms.

Q.E.D.

Here now is the statement of LISP-induction.

4.4.4 Inference rule: LISP-induction

To infer $(\forall p.p \overset{*}{>} A \Rightarrow R(p,A))$ prove BASE and INDUCTION below:

BASE/

BASE: $\forall$ A.R(A,A)

INDUCTION: $\forall$ e,a.(($\forall$p' <* (e,a).p'*>A' => R(p',A')) and (e,a)*>A) => R((e,a),A)

Remark: The <u>LISP-induction hypothesis (LIH)</u> of such arguments is

($\forall$p' <* (e,a).p'*>A' => R(p',A'))

The validity of LISP-induction follows directly from lemma 4.4.3.as that lemma implies:

($\forall$p.p*>A => R(p,A)) <=> ($\forall$ well-founded p.p*>A => R(p,A))

and BASE and INDUCTION are equivalent to this latter statement.
In doing the INDUCTION step of a proof by LISP-induction one has to consider the various cases of p=(e,a) for which p*>A, the next lemma lists these cases.

4.4.5 <u>Lemma</u>

Suppose p=(e,a)*>A then one of the nine cases below obtains (in what follows p *> p'*>p" means p *> p' and p'*>p").

1. p=(A,a)

2. p=(x,a) and a(x)=A

3. p=(F[A_1;...;A_n],a) and F(A_1,...,A_n)=A

4./

4. $p=(fn[e_1;\dots;e_n],a)$ and there are $A_1,\dots,A_n$ such that:
$\forall i.p\overset{*}{>}(e_i,a)\overset{*}{>}A_i$, $\exists m.e_m\neq A_m$ and $p \overset{*}{>} (fn[A_1;\dots;A_n],a)\overset{*}{>}A$

5. $p=([e_{11}\rightarrow e_{12};\dots;e_{n1}\rightarrow e_{n2}],a)$ and there is an m ($1\leq m\leq n$) such that:
$\forall i<m.p \overset{*}{>} (e_i,a)\rightarrow F, p \overset{*}{>} (e_{m1},a)\overset{*}{>}T$ and $p \overset{*}{>} (e_{m2},a)\overset{*}{>}A$

6. $p=(f[A_1;\dots;A_n],a), a(f)=fn$ and $p \overset{*}{>} (fn[A_1;\dots;A_n],a)\overset{*}{>}A$

7. $p=(\lambda[[x_1;\dots;x_m];e][A_1;\dots;A_n],a), m\leq n$ and $p \overset{*}{>} (e,(A_1/x_1)\dots(A_m/x_m)a)\overset{*}{>}A$

8. $p=(Label[f;fn][A_1;\dots;A_n],a)$ and $p \overset{*}{>} (fn[A_1;\dots;A_n],(fn/f)a)\overset{*}{>}A$

8. $p=(\mu[f;fn][A_1;\dots;A_n],a)$ and $p \overset{*}{>} (fn[A_1;\dots;A_n],(\mu[f;fn]/f)a)\overset{*}{>}A$

<u>Proof</u>

Straight from definitions of -> (4.2.2) and <* (4.4.1).

Q.E.D.

The rest of this chapter is concerned with using LISP-induction to get interesting results.

4.5 <u>Proof of half of the Main Theorem: an example of LISP-induction</u>

4.5.1 <u>Lemma</u> (Soundness of ->)

$p\overset{*}{>}A \Rightarrow [p]=A$

<u>Proof</u>

If/

If $R(p,A) \iff [\![p]\!]=A$ then what has to be proved is:

$p \overset{*}{\Rightarrow} A \Rightarrow R(p,A)$

I show this by LISP-induction.

BASE:

$R(A,A) \iff [\![A]\!]=A$ which is true.

INDUCTION:

Assume LIH (viz. $(\forall p' <^* (e,a).p' \overset{*}{\Rightarrow} A' \Rightarrow [\![p']\!]=A')$) and $p=(e,a)\overset{*}{\Rightarrow}A$

I show $[\![(e,a)]\!]=[\![e]\!]([\![a]\!])=A$ by cases on p (see 4.4.5).

1. $p=(A,a)$

   Clearly $[\![p]\!]=A$

2. $p=(x,a)$ and $a(x)=A$

   $[\![p]\!]=[\![x]\!]([\![a]\!])$ (Definition of $[\![p]\!]$ - 4.3.1)

   $=[\![a]\!](x)([\![a]\!]) \mid S$ (by semantic equations - (S2) of 3.14)

   $=A$ (by the remark after definition 4.2.1)

3. $p=(F[A_1;\ldots;A_n],a)$ and $F(A_1,\ldots,A_n)=A$

   $[\![p]\!]=F(A_1,\ldots,A_n)=A$

4. $p=(fn[e_1;\ldots;e_n],a)$ and $\forall i.(e_i,a)\overset{*}{\Rightarrow}A_i$ and $\exists m.e_m \neq A_m$

   Then by LIH $R((e_i,a),A_i)$ i.e. $[\![e_i]\!]([\![a]\!])=A_i$

   and $R((fn[A_1;\ldots;A_n],a),A)$ i.e. $[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)=A$

   So $[\![p]\!]=[\![fn]\!]([\![a]\!])([\![e_1]\!]([\![a]\!]),\ldots,[\![e_n]\!]([\![a]\!]))$

   $=[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)$

   $=A$

5./

5. $p=([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}],a)$ and $(e_{m1},a)\overset{*}{\Rightarrow}T$ and $\forall\, i<m.(e_{i1},a)\overset{*}{\Rightarrow}F$

Then by LIH $R((e_{m1},a),T)$, $\forall i<m.R((e_{i1},a),F)$ and $R((e_{m2},a),A)$

i.e. $[\![e_{m1}]\!]([\![a]\!])=T$, $\forall i<m\ [\![e_{i1}]\!]([\![a]\!])=F$ and $[\![e_{m2}]\!]([\![a]\!])=A$

so $[\![p]\!]=([\![e_{11}]\!]([\![a]\!])\rightarrow [\![e_{12}]\!]([\![a]\!]),\ldots,[\![e_{n1}]\!]([\![a]\!])\rightarrow [\![e_{n2}]\!]([\![a]\!]))$

$=(F\rightarrow [\![e_{12}]\!]([\![a]\!]),\ldots,T\rightarrow A,\ldots,[\![e_{n1}]\!]([\![a]\!])\rightarrow [\![e_{n2}]\!]([\![a]\!]))$

$=A.$

6. $p=(f[A_1;\ldots;A_n],a)$ and $a(f)=fn.$

Then by LIH $R((fn[A_1;\ldots;A_n],a),A)$ i.e. $[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)=A$

so $[\![p]\!]=[\![f]\!]([\![a]\!])(A_1,\ldots,A_n)$

$=[\![a]\!](f)([\![a]\!])\mid FUN(A_1,\ldots,A_n)$

$=[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)$

$=A$

7. $p=(\lambda[[x_1;\ldots;x_m];e][A_1;\ldots;A_n],a)$ and $m\leq n$

Then by LIH $R((e,(A_1/x_1)\ldots(A_m/x_m)a),A)$ i.e. $[\![e]\!]([\![(A_1/x_1)\ldots(A_m/x_m)a]\!])=A$

so $[\![p]\!]=(\lambda t_1\ldots t_m.[\![e]\!]((t_1/x_1)\ldots(t_m/x_m)[\![a]\!]))(A_1,\ldots,A_n)$

$=[\![e]\!]((A_1/x_1)\ldots(A_m/x_m)[\![a]\!])$

$=[\![e]\!]([\![(A_1/x_1)\ldots(A_m/x_m)a]\!])$

$=A$

8./

8. $p=(\mathrm{Label}[f;fn][A_1;\ldots;A_n],a)$

Then by LIH $R((fn[A_1;\ldots;A_n],(fn/f)a),A)$ i.e. $[\![fn]\!]([\![(fn/f)a]\!])(A_1,\ldots,A_n)=A$

so $[\![p]\!]=[\![\mathrm{Label}[f;fn]]\!]([\![a]\!])(A_1,\ldots,A_n)$

$=[\![fn]\!](([\![fn]\!]/f)[\![a]\!])(A_1,\ldots,A_n)$

$=[\![fn]\!]([\![(fn/f)a]\!])(A_1,\ldots,A_n)$

$=A$

9. $p=(\mu[f;fn][A_1;\ldots;A_n],a)$

Then by LIH $R((fn[A_1;\ldots;A_n],(\mu[f;fn]/f)a),A)$ i.e. $[\![fn]\!]([\![(\mu[f;fn]/f)a]\!])$

$(A_1,\ldots,A_n)=A$

so $[\![p]\!]=[\![\mu[f;fn]]\!]([\![a]\!])(A_1,\ldots,A_n)$

$=[\![fn]\!](([\![\mu[f;fn]]\!]/f)[\![a]\!])(A_1,\ldots,A_n)$

$=[\![fn]\!]([\![(\mu[f;fn]/f)a]\!])(A_1,\ldots,A_n)$

$=A.$

Q.E.D.

4.6 <u>The relation between μ and Label</u>

It would be very nice if for all f,fn: $[\![\mu[f;fn]]\!]=[\![\mathrm{Label}[f;fn]]\!]$, unfortunately this is not so. Here is a counterexample:

First given $f,g \in Id$ let me construct an $r \in Env$ such that for all $r' \in Env$:

1. $r(f)=[\![car]\!]$
2. $r(g)(r')=r'(f)(r)$

This/

This is easily achieved by letting $r = Y(\lambda r''.(\llbracket car \rrbracket/f)((\lambda r'.r'(f)(r''))/g)\perp)$

i.e. $r = (\llbracket car \rrbracket/f)((\lambda r'.r'(f)(r))/g)\perp$

Then I claim that: $\llbracket Label[f;g] \rrbracket(r) = \llbracket car \rrbracket(r) \neq \perp = \llbracket \mu[f;g] \rrbracket(r)$

The proof of this is as follows:

$$\begin{aligned}
\llbracket Label[f;g] \rrbracket(r) &= \llbracket g \rrbracket((\llbracket g \rrbracket/f)r) \\
&= r(g)((\llbracket g \rrbracket/f)r) \mid FUN \\
&= ((\llbracket g \rrbracket/f)r)(f)(r) \mid FUN && \text{(by 2. above)} \\
&= \llbracket g \rrbracket(r) \\
&= r(g)(r) \mid FUN \\
&= r(f)(r) \mid FUN && \text{(by 2. above)} \\
&= \llbracket car \rrbracket(r) && \text{(by 1. above)}
\end{aligned}$$

$$\begin{aligned}
\llbracket \mu[f;g] \rrbracket(r) &= Y(F)(r) \text{ where } F = \lambda v.\lambda r'.\llbracket g \rrbracket((v/f)r') \\
&= \bigsqcup_{n=0}^{\infty} F^n(\perp)(r)
\end{aligned}$$

Now $F^0(\perp)(r) = \perp(r) = \perp$ Suppose that $n>0$ and $F^n(\perp)(r) = \perp$ then:

$$\begin{aligned}
F^{n+1}(\perp)(r) &= \llbracket g \rrbracket((F^n(\perp)/f)r) \\
&= r(g)((F^n(\perp)/f)r) \mid FUN \\
&= ((F^n(\perp)/f)r)(f)(r) \mid FUN && \text{(by 2. above)} \\
&= F^n(\perp)(r) \mid FUN \\
&= \perp \mid FUN = \perp && \text{(by assumption)}
\end{aligned}$$

So by induction on n: for all $n \geq 0$ $F^n(\perp)(r) = \perp$ hence

[/

$$[\![\mu[f;g]]\!](r)=\bigsqcup_{n=0}^{\infty}F^{n}(\perp)(r)=\perp$$

Thus the claim above is justified. Despite this counterexample it is the case that for all a ϵ <alist> and all f, fn:

$$[\![\text{Label}[f;fn]]\!]([\![a]\!])=[\![\mu[f;fn]]\!]([\![a]\!])$$

and this is good enough because the only environments which arise in practice are those which correspond to alists, i.e. ones of the form $[\![a]\!]$. To prove this result I shall use LISP-induction, but first let me show you how to intuit it. By the Main Theorem and corollary 4.3.4 all that needs to be shown is:

$$(\text{Label}[f;fn][A_1;\ldots;A_n],a)\overset{*}{\rightarrow}A \iff (\mu[f;fn][A_1;\ldots;A_n],a)\overset{*}{\rightarrow}A$$

Now if you look at clauses P6, P8, P9 of the definition of -> you will see that to any computation of the form:

$$(\text{Label}[f;fn][A_1;\ldots;A_n],a)\rightarrow P_1\rightarrow P_2\rightarrow\ldots\rightarrow P_n\rightarrow A$$

there corresponds one of the form:

$$(\mu[f;fn][A_1;\ldots;A_n],a)\rightarrow P_1'\rightarrow P_2'\rightarrow\ldots\rightarrow P_n'\rightarrow A$$

and vice versa; where (approximately) P_i' is got from P_i by replacing some Labels by μ's and adding some extra μ's to the alist. This is an operational intuition and that is why LISP-induction (and the Main Theorem) is needed to convert it into a formal proof. The next two/

two definitions are to enable me to say precisely what the correspondence between the above two computations is.

4.6.1 <u>Definition</u>

If e, e' are forms (and fn, fn' functions) then write e≑e' (and fn≑fn') if and only if e' can be got from e (fn' got from fn) by changing zero or more Labels to μ's and zero or more μ's to Labels.

Examples: 1. Label[f;fn]≑μ[f;fn]

2. μ[g;Label[f;fn]][A]≑μ[g; μ[f;fn]][A]

I shall prove that if e≑e' then ⟦e⟧(⟦a⟧)=⟦e'⟧(⟦a⟧) for all a; however this statement is not strong enough for the induction to be carried through. The next definition enables me to formulate a strong enough induction hypothesis.

4.6.2 <u>Definition</u>

If a, a' $\in$ <alist> then write a≑a' <=> for all z:

(1) a(z)=A <=> a'(z)=A

(2) If a(z)=fn then <u>either</u> a'(z)=fn' where fn≑fn'

<u>or</u> a'(z)=μ[z;fn'] where fn≑fn'

(3)/

(3) If $a'(z)=fn'$ then <u>either</u> $a(z)=fn$ where $fn \doteq fn'$

<u>or</u> $a(z)=\mu[z;fn]$ where $fn \doteq fn'$

If $p, p' \in$ <term> then $p \doteq p' \Leftrightarrow p=A=p'$ or $p=(e,a)$

and $p'=(e',a')$

and $e \doteq e'$ and $a \doteq a'$

Remark: $\doteq$ is symmetric.

4.6.3 <u>Lemma</u>

$p \doteq p'$ and $p \overset{*}{\Rightarrow} A \Rightarrow p' \overset{*}{\Rightarrow} A$

<u>Proof</u>

Let $R(p,A) \Leftrightarrow (\forall p'. p \doteq p' \Rightarrow p' \overset{*}{\Rightarrow} A)$ then I need to show that $p \overset{*}{\Rightarrow} A \Rightarrow R(p,A)$

I prove this by LISP-induction.

BASE:

$R(A,A) \Leftrightarrow \forall p' \doteq A.\ p' \overset{*}{\Rightarrow} A$ which is true since $p' \doteq A \Rightarrow p'=A$ and $A \overset{*}{\Rightarrow} A$.

INDUCTION:

Assume LIH and $p=(e,a) \overset{*}{\Rightarrow} A$. I show $R(p,A)$ by cases on p (see lemma 4.4.5). In what follows assume $a \doteq a'$, $fn \doteq fn'$, $e_i \doteq e_i'$, $e_{ij} \doteq e_{ij}'$ etc.

1. $p=(A,a)$

$p \doteq p' \Rightarrow p'=(A,a') \Rightarrow p' \rightarrow A \Rightarrow p' \overset{*}{\Rightarrow} A$

2./

2. $p=(x,a)$ and $a(x)=A$

   $p \doteqdot p' \Rightarrow p'=(x,a)$ and $a'(x)=A \Rightarrow p' \to A \Rightarrow p' \overset{*}{\Rightarrow} A$

3. $p=(F[A_1;\ldots;A_n],a)$ and $f(A_1,\ldots,A_n)=A$

   $p \doteqdot p' \Rightarrow p'=(F[A_1;\ldots;A_n],a') \Rightarrow p' \to A \Rightarrow p' \overset{*}{\Rightarrow} A$

4. $p=(fn[e_1;\ldots;e_n],a)$ and $\forall i.(e_i,a) \overset{*}{\Rightarrow} A_i$ and $\exists m.e_m \neq A_m$

   Then $p \doteqdot p' \Rightarrow p'=(fn'[e_1';\ldots;e_n'],a')$. Now $p \to (fn[A_1;\ldots;A_n],a)$ and by LIH $\forall i.R((e_i,a),A_i)$ and $R((fn[A_1;\ldots;A_n],a),A)$ hence

   $p' \to (fn'[A_1;\ldots;A_n],a') \overset{*}{\Rightarrow} A$

5. $p=([e_{11} \to e_{12};\ldots;e_{n1} \to e_{n2}],a)$ and $(e_{m1},a) \overset{*}{\Rightarrow} T$ and $\forall i<m.(e_{i1},a) \overset{*}{\Rightarrow} F$

   Then $p \doteqdot p' \Rightarrow p'=([e_{11}' \to e_{12}';\ldots;e_{n1}' \to e_{n2}'],a')$

   Now $p \to (e_{m2},a)$ and by LIH $R((e_{m1},a),T)$, $\forall i<m.R((e_{i1},a),F)$

   and $R((e_{m2},a),A)$ so

   $p' \to (e_{m2}',a') \overset{*}{\Rightarrow} A$

6. $p=(f[A_1;\ldots;A_n],a)$ and $a(f)=fn$

   Then $p \to (fn[A_1;\ldots;A_n],a)$ so by LIH $R((fn[A_1;\ldots;A_n],a),A)$

   Let $p \doteqdot p'$, there are two cases to consider:

   6.1 $p'=(f[A_1;\ldots;A_n],a')$ and $a'(f)=fn'$ where $fn \doteqdot fn'$

   Then $p' \to (fn'[A_1;\ldots;A_n],a') \overset{*}{\Rightarrow} A$ (by LIH)

   6.2 $p'=(f[A_1;\ldots;A_n],a')$ and $a'(f)=\mu[f;fn']$ where $fn \doteqdot fn'$

   Then $p' \to (\mu[f;fn'][A_1;\ldots;A_n],a') \to (fn'[A_1;\ldots;A_n],(\mu[f;fn']/f)a')$

   $\overset{*}{\Rightarrow} A$ (by LIH and definition 4.6.2)

7./

7. $p=(\lambda[[x_1;\dots;x_m];e][A_1;\dots;A_n],a)$ and $m\leq n$

   Then $p\Rightarrow p' \Rightarrow p'=(\lambda[[x_1;\dots;x_m];e'][A_1;\dots;A_n],a')$

   Now $p\rightarrow(e,(A_1/x_1)\dots(A_m/x_m)a)$ so by LIH $R((e,(A_1/x_1)\dots(A_m/x_m)a),A)$

   hence $p'\rightarrow(e',(A_1/x_1)\dots(A_m/x_m)a')\overset{*}{\Rightarrow}A$

8. $p=(\text{Label}[f;fn][A_1;\dots;A_n],a)$

   Then $p\rightarrow(fn[A_1;\dots;A_n],(fn/f)a)$ so by LIH $R((fn[A_1;\dots;A_n],(fn/f)a),A)$

   Let $p\Rightarrow p'$ then there are two cases to consider:

   8.1 $p'=(\text{Label}[f;fn'][A_1;\dots;A_n],a')$

   Then $p'\rightarrow(fn'[A_1;\dots;A_n],(fn'/f)a')\overset{*}{\Rightarrow}A$ (by LIH)

   8.2 $p'=(\mu[f;fn'][A_1;\dots;A_n],a')$

   Then $p'\rightarrow(fn'[A_1;\dots;A_n],(\mu[f;fn']/f)a')\overset{*}{\Rightarrow}A$ (by LIH)

9. $p=(\mu[f;fn][A_1;\dots;A_n],a)$

   Then $p\rightarrow(fn[A_1;\dots;A_n],(\mu[f;fn]/f)a)$ so by LIH $R((fn[A_1;\dots;A_n],$
   $(\mu[f;fn]/f)a),A)$

   Let $p\Rightarrow p'$ then there are two cases to consider:

   9.1 $p'=(\mu[f;fn'][A_1;\dots;A_n],a')$

   Then $p'\rightarrow(fn'[A_1;\dots;A_n],(\mu[f;fn']/f)a')\overset{*}{\Rightarrow}A$ (by LIH)

   9.2 $p'=(\text{Label}[f;fn'][A_1;\dots;A_n],a')$

   Then $p'\rightarrow(fn'[A_1;\dots;A_n],(fn'/f)a')\overset{*}{\Rightarrow}A$ (by LIH)

Q.E.D.

4.6.4 <u>Theorem</u>

$p\Rightarrow p' \Rightarrow [\![p]\!]=[\![p']\!]$

<u>Proof/</u>

Proof

By lemma 4.6.3 and the obvious symmetry of $\doteqdot$

$$p \doteqdot p' \Rightarrow (p \overset{*}{\Rightarrow} A \Leftrightarrow p' \overset{*}{\Rightarrow} A)$$

hence by the Main Theorem

$$p \doteqdot p' \Rightarrow ([\![p]\!]=A \Leftrightarrow [\![p']\!]=A)$$

so as either $[\![p]\!]=\perp$ or $[\![p]\!]=A$ for some A (and similarly for $[\![p']\!]$) it follows that:

$$p \doteqdot p' \Rightarrow [\![p]\!]=[\![p']\!]$$

Q.E.D.

This shows that if you take any term p=(e,a) and randomly change some Labels to μ's and μ's to Labels you do not change the meaning of the term. In particular since $(\mathrm{Label}[f;fn][A_1;\ldots;A_n],a) \doteqdot (\mu[f;fn][A_1;\ldots;A_n],a)$ the theorem (and corollary 4.3.4) imply that:

$$[\![\mathrm{Label}[f;fn]]\!]([\![a]\!])=[\![\mu[f;fn]]\!]([\![a]\!]).$$

4.7 A proposition about variables

The result proved in this section is needed later on. It is not of much intrinsic interest.

One/

One might at first sight expect that if r, r' agree on the free variables in fn then $[\![fn]\!](r)=[\![fn]\!](r')$; this is not the case for let $r=[\![(g/f)(car/g)]\!]$, $r'=[\![(g/f)(cdr/g)]\!]$ then r and r' agree on f but $[\![f]\!](r)=car\neq cdr=[\![f]\!](r')$. The proposition below is the result of debugging this intuition.

4.7.1 Definition

Let vs(e), vs(fn) be the sets of variables (identifiers) not bound by λ's, Labels or μ's in e, fn respectively.

Example: $vs(\lambda[[x];y][z])=\{y,z\}$

Given a form e the next definition gives sufficient conditions on a set $Z \subseteq$ <identifier> so that if a, a' agree on Z then $[\![e]\!]([\![a]\!])=[\![e]\!]([\![a']\!])$. This is proved in the lemma below.

4.7.2 Definition

If $Z \subseteq$ <identifier> and p, p' $\in$ <term> then $p \underset{Z}{=} p'$ $\Leftrightarrow$

either p=p'=A

or p=(e,a), p'=(e,a') and (1), (2), (3) where

(1) $vs(e) \subseteq Z$

(2) $\forall z \in Z.\ a(z)=a'(z)$

(3) $\forall z \in Z.\ vs(a(z)) \subseteq Z$.

4.7.3/

4.7.3 <u>Lemma</u>

If p, $p' \in$ <term> and there exists $Z \subseteq$ <identifier> such that $p \equiv_Z p'$ then

$$p \overset{*}{\rightarrow} A \Rightarrow p' \overset{*}{\rightarrow} A$$

<u>Proof</u>

Let $R(p,A) \Leftrightarrow \forall p'((\exists Z. p \equiv_Z p') \Rightarrow p' \overset{*}{\rightarrow} A)$

Then I need to show $p \overset{*}{\rightarrow} A \Rightarrow R(p,A)$ which I can prove by LISP-induction.

BASE:

$R(A,A) \Leftrightarrow A \overset{*}{\rightarrow} A$ which is true.

INDUCTION:

Assume LIH and $p=(e,a) \overset{*}{\rightarrow} A$ I prove $R(p,A)$ by cases on p (see lemma 4.4.5).

1. $p=(A,a)$

   Then $p \equiv_Z p' \Rightarrow p'=(A,a') \rightarrow A$

2. $p=(x,a)$ and $a(x)=A$

   Then $x \in Z$ and so $p \equiv_Z p' \Rightarrow p'=(x,a')$ where $a'(x)=A$ (by definition of $\equiv_Z$)

   $\Rightarrow p' \rightarrow A$

3. $p=(F[A_1;\ldots;A_n],a)$ and $F(A_1,\ldots,A_n)=A$

   Then $p \equiv_Z p' \Rightarrow p'=(F[A_1;\ldots;A_n],a') \Rightarrow p' \rightarrow A$

4./

4. $p=(fn[e_1;\ldots;e_n],a)$ and $\forall i.(e_i,a)\overset{*}{=}>A_i$ and $\exists m.e_m\not\overset{*}{=}>A_m$

   By LIH $\forall i.R((e_i,a),A_i)$ and $R((fn[A_1;\ldots;A_n],a),A)$ so

   $p\overline{\overline{=}}_Z p' \Rightarrow p'=(fn[e_1;\ldots;e_n],a')\text{->}(fn[A_1;\ldots;A_n],a')\overset{*}{=}>A$

   since $vs(fn),vs(e_i) \subseteq vs(fn[e_1;\ldots;e_n]) \subseteq Z$

5. $p=([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}],a)$ and $(e_{m1},a)\overset{*}{=}>T$ and $\forall i<m.(e_{i1},a)\overset{*}{=}>F$

   By LIH $R((e_{m1},a),T)$, $\forall i<m.R((e_{i1},a),F)$ and $R((e_{m2},a),A)$ so

   $p\overline{\overline{=}}_Z p' \Rightarrow p'=([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}],a')\text{->}(e_{m2},a')\overset{*}{=}>A$

   since $vs(e_{ij}) \subseteq vs([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}]) \subseteq Z$

6. $p=(f[A_1;\ldots;A_n],a)$ and $a(f)=fn$

   By LIH $R((fn[A_1;\ldots;A_n],a),A)$ so

   $p=p' \Rightarrow p'=(f[A_1;\ldots;A_n],a')\text{->}(fn[A_1;\ldots;A_n],a')\overset{*}{=}>A$

   since $f \in Z$ so $a'(f)=a(f)=fn$ and $vs(fn)=vs(a(f)) \subseteq Z$

7. $p=(\lambda[[x_1;\ldots;x_m];e][A_1;\ldots;A_n],a)$ and $\underline{m\leq n}$

   By LIH $R((e,(A_1/x_1)\ldots(A_m/x_m)a),A)$ so

   $p\overline{\overline{=}}_Z p' \Rightarrow p'=(\lambda[[x_1;\ldots;x_m];e][A_1;\ldots;A_n],a')\text{->}(e,(A_1/x_1)\ldots(A_m/x_m)a')\overset{*}{=}>A$

   since $(e,(A_1/x_1)\ldots(A_m/x_m)a)\overline{\overline{=}}_{Z'}(e,(A_1/x_1)\ldots(A_m/x_m)a')$

   where $Z'=Z \cup \{x_1,\ldots,x_m\}$

8. $p=(Label[f;fn][A_1;\ldots;A_n],a)$

   By LIH $R((fn[A_1;\ldots;A_n],(fn/f)a),A)$ so

   $p\overline{\overline{=}}_Z p' \Rightarrow p'=(Label[f;fn][A_1;\ldots;A_n],a')\text{->}(fn[A_1;\ldots;A_n],(fn/f)a')\overset{*}{=}>A$

   since $(fn[A_1;\ldots;A_n],(fn/f)a)\overline{\overline{=}}_{Z'}(fn[A_1;\ldots;A_n],(fn/f)a')$

   where $Z'=Z \cup \{f\}$

9./

9. $p=(\mu[f;fn][A_1;\ldots;A_n],a)$

By LIH $R((fn[A_1;\ldots;A_n],(\mu[f;fn]/f)a),A)$ so

$p\underset{Z}{=}p' \Rightarrow p'=(\mu[f;fn][A_1;\ldots;A_n],a')\rightarrow(fn[A_1;\ldots;A_n],(\mu[f;fn]/f)a')\overset{*}{>}A$

since $(fn[A_1;\ldots;A_n],(\mu[f;fn]/f)a)\underset{Z'}{=}(fn[A_1;\ldots;A_n],(\mu[f;fn]/f)a')$

where $Z'=Z\cup\{f\}$

Q.E.D.

4.7.4 <u>Corollary</u>

If p, $p' \in$ <term> and there exists $Z \subseteq$ <identifier> such that $p\underset{Z}{=}p'$ then $p\overset{*}{>}A \Leftrightarrow p'\overset{*}{>}A$.

<u>Proof</u>

By lemma 4.7.3 and symmetry of $\underset{Z}{=}$.

Q.E.D.

4.7.5 <u>Proposition</u>

Let $fn \in$ <function>, $a,a' \in$ <alist> then if there exists $Z \subseteq$ <identifier> such that:

$vs(fn) \subseteq Z$ (vs is defined in 4.7.1)

<u>and</u> $\forall z \in Z.a(z)=a'(z)$

<u>and</u> $\forall z \in Z.vs(a(z)) \subseteq Z$

Then $[\![fn]\!]([\![a]\!])=[\![fn]\!]([\![a']\!])$.

<u>Proof</u>/

Proof

By corollary 4.7.4 for all $A_1,\ldots,A_n \in$ <S-expression>:

$$(fn[A_1;\ldots;A_n],a)\overset{*}{\Rightarrow}A \iff (fn[A_1;\ldots;A_n],a')\overset{*}{\Rightarrow}A$$

hence $[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)=[\![fn]\!]([\![a']\!])(A_1,\ldots,A_n)$
and so (by corollary 4.3.4) $[\![fn]\!]([\![a]\!])=[\![fn]\!]([\![a']\!])$.

Q.E.D.

4.8 A semantic analysis of the LISP eval function

In this section I shall examine the relation holding between values computed by the manual interpreter and denotations assigned by my semantics. I am going to interpret the M-expressions defining apply, eval,...etc. via the semantic equations in 3.14 - as mentioned earlier this denotational interpretation is not necessarily the same as the one implementers have taken. That one I have tried to capture in $\Rightarrow$.

In my discussion of $v[\![e]\!]$,$w[\![fn]\!]$ in 3.10 I pointed out the problem of accommodating simultaneous recursions as used to define apply,eval...etc. That problem raises its head here too, but using my improved environments I can get over it in the same way that actual implementations do. I shall define a special alist int (for "interpreter") which contains the definitions of apply,eval,evcon,evlis and other 'system functions/

functions', then the denotations of apply,eval...etc. will be ⟦apply⟧(⟦int⟧),⟦eval⟧(⟦int⟧)...etc. Here is the definition of int; this should be compared with the definition of the interpreter given in the Manual.

4.8.1 Definition

```
Let int=(applyfn/apply)(evalfn/eval)(evconfn/evcon)(evlisfn/evlis)
        (caarfn/caar)(cdarfn/cdar)(cadrfn/cadr)(caddrfn/caddr)(cadarfn/cadar)
        (nullfn/null)(equalfn/equal)(pairlisfn/pairlis)(assocfn/assoc)NIL
```

where

4.8.1.1

```
applyfn=λ[[fn;x;a];
          [atom[fn]→ [eq[fn;CAR]→ caar[x];
                      eq[fn;CDR]→ cdar[x];
                      eq[fn;CONS]→ cons[car[x];cadr[x]];
                      eq[fn;ATOM]→ atom[car[x]];
                      eq[fn;EQ]→ eq[car[x];cadr[x]];
                           T   → apply[eval[fn;a];x;a]];
          eq[car[fn];LAMBDA]→ eval[caddr[fn];pairlis[cadr[fn];x;a]];
          eq[car[fn];LABEL]→ apply[caddr[fn];x;cons[cons[cadr[fn];
                                                   caddr[fn]];a]]]]
```

4.8.1.2/

4.8.1.2

```
    evalfn=λ[[e;a];
              [atom[e]→ cdr[assoc[e;a]];
               atom[car[e]]→ [eq[car[e];QUOTE]→ cadr[e];
                              eq[car[e];COND]→ evcon[cdr[e];a];
                                     T        → apply[car[e];evlis[cdr[e];a];a]];
               T→ apply[car[e];evlis[cdr[e];a];a]]]
```

4.8.1.3

```
evconfn= λ[[c;a];
           [eval[caar[c];a]→ eval[cadar[c];a];
                   T       → evcon[cdr[c];a]]]
```

4.8.1.4

```
evlisfn= λ[[m;a];
           [null[m]→ NIL;
               T    → cons[eval[car[m];a];evlis[cdr[m];a]]]]
```

4.8.1.5

```
caarfn= λ[[x];car[car[x]]]
```

4.8.1.6

```
cdarfn= λ[[x];cdr[car[x]]]
```

4.8.1.7/

4.8.1.7

```
cadrfn= λ[[x];car[cdr[x]]]
```

4.8.1.8

```
caddrfn= λ[[x];car[cdr[cdr[x]]]]
```

4.8.1.9

```
cadarfn= λ[[x];car[cdr[car[x]]]]
```

4.8.1.10

```
nullfn= λ[[x];[atom[x]→ eq[x;NIL];T→ F]]
```

4.8.1.11

```
equalfn= λ[[x;y];[atom[x]→ [atom[y]→ eq[x;y];T→ F];
                  atom[y]→ [atom[x]→ eq[x;y];T→ F];
                  equal[car[x];car[y]]→ equal[cdr[x];cdr[y]];
                     T→ F]]
```

Remark: The definition of equal given in the Manual does not agree with the informal description of it if x=(1), y=1 (see 2.2.3).

4.8.1.12/

<u>4.8.1.12</u>

```
pairlisfn= λ[[x;y;a];[null[x]→ a;
                       T     → cons[cons[car[x];car[y]];
                                   pairlis[cdr[x];cdr[y];a]]]]
```

<u>4.8.1.13</u>

```
assocfn= λ[[x;a];[equal[caar[a];x]→ car[a];T→ assoc[x;cdr[a]]]]
```

The functions apply,eval,evcon,evlis,caar,cdar,cadr,caddr,cadar,null, equal,pairlis,assoc are all in FUN=[S*->S] and are defined below, context will distinguish these from the identifiers they are bound to on ⟦int⟧.

4.8.2 <u>Definition</u>

```
  apply=⟦apply⟧(⟦int⟧)=⟦applyfn⟧(⟦int⟧)
   eval=⟦eval⟧(⟦int⟧) =⟦evalfn⟧(⟦int⟧)
  evcon=⟦evcon⟧(⟦int⟧)=⟦evconfn⟧(⟦int⟧)
  evlis=⟦evlis⟧(⟦int⟧)=⟦evlisfn⟧(⟦int⟧)
   caar=⟦caar⟧(⟦int⟧) =⟦caarfn⟧(⟦int⟧)
   cdar=⟦cdar⟧(⟦int⟧) =⟦cdarfn⟧(⟦int⟧)
   cadr=⟦cadr⟧(⟦int⟧) =⟦cadrfn⟧(⟦int⟧)
  caddr=⟦caddr⟧(⟦int⟧)=⟦caddrfn⟧(⟦int⟧)
  cadar=⟦cadar⟧(⟦int⟧)=⟦cadarfn⟧(⟦int⟧)
   null=⟦null⟧(⟦int⟧) =⟦nullfn⟧(⟦int⟧)
  equal=⟦equal⟧(⟦int⟧)=⟦equalfn⟧(⟦int⟧)
pairlis=⟦pairlis⟧(⟦int⟧)=⟦pairlisfn⟧(⟦int⟧)
  assoc=⟦assoc⟧(⟦int⟧)=⟦assocfn⟧(⟦int⟧)
```

In order to formulate precisely what it means for the semantic equations and manual interpreter to be in harmony I shall extend the translation of M-expressions into S-expressions given in the Manual to include alists. For the rest of this section (4.8) I want to restrict the range of the meta variables e, fn, a, p to exclude terms containing μ's. To achieve this here is a 'local' definition of these variables:

$$e ::= A \mid x \mid fn[e_1;\ldots;e_n] \mid [e_{11} \rightarrow e_{12};\ldots;e_{n1} \rightarrow e_{n2}]$$

$$fn ::= F \mid f \mid \lambda[[x_1;\ldots;x_m];e] \mid Label[f;fn]$$

$$a ::= NIL \mid (A/x)a \mid (fn/f)a$$

$$p ::= A \mid (e,a)$$

4.8.3 Definition

The translations e*,fn*,a* of e,fn,a are defined by structural induction by:

e*:

$A^* = (QUOTE\ A)$

$x^* = X$

$fn[e_1;\ldots;e_n]^* = (fn^*\ e_1^*\ldots e_n^*)$

$[e_{11} \rightarrow e_{12};\ldots;e_{n1} \rightarrow e_{n2}]^* = (COND\ (e_{11}^*\ e_{12}^*)\ldots(e_{n1}^*\ e_{n2}^*))$

fn*:/

<u>fn*</u>:

car*=CAR

cdr*=CDR

cons*=CONS

atom*=ATOM

eq*=EQ

$\lambda[[x_1;\ldots;x_m];e]$*=(LAMBDA ($x_1^*\ldots x_m^*$) e*)

Label[f;fn]*=(LABEL f* fn*)

<u>a*</u>:

NIL*=NIL

((A/z)a)*=((z*.A).a*)

((fn/z)a)*=((z*.fn*).a*)

The following questions can now be asked:

1. Does apply(fn*,($A_1\ldots A_n$),a*)=⟦fn⟧(⟦a⟧)($A_1,\ldots,A_n$) ?
2. Does eval(e*,a)=⟦e⟧(⟦a⟧) ?

Unfortunately the answer to both these is "no", however they only fail in a rather trivial way which is illustrated by the following example:

example: let e=z, a=(fn/z)NIL

Then eval(e*,a*)=eval(Z,((Z.fn*)))

=fn*

but/

$$\begin{aligned} \text{but } [\![e]\!]([\![a]\!]) &= [\![z]\!]([\![a]\!]) \\ &= [\![a]\!](z)[\![a]\!] \mid S \\ &= [\![fn]\!]([\![a]\!]) \mid S \\ &= \perp \end{aligned}$$

To bar such monsters one just needs to separate the x's from the f's i.e. to say that if an identifier is used to name a function then it cannot be used as a form variable in the same program. Given this the two questions above get affirmative answers. To make this precise here is a definition.

4.8.4 Definition

Call (e,a) nice if

$$(\text{formvs}(e) \cap \text{formvs}(a)) \cup (\text{funvs}(e) \cap \text{funvs}(a)) = \emptyset$$

where: $\text{formvs}(e)=\{z \mid z \text{ is a form variable in } e\}$

$\text{formvs}(a)=\{z \mid a(z) \in \langle\text{form}\rangle\}$

$\text{funvs}(e)=\{z \mid z \text{ is a function variable in } e\}$

$\text{funvs}(a)=\{z \mid a(z) \in \langle\text{function}\rangle\}$

4 8.5 Theorem

(1) For all $fn, e, a, A_1, \ldots, A_n$

$$\text{apply}(fn^*, (A_1 \ldots A_n), a^*) \sqsupseteq [\![fn]\!]([\![a]\!])(A_1, \ldots, A_n)$$

$$\text{eval}(e^*, a^*) \sqsupseteq [\![e]\!]([\![a]\!]).$$

(2)/

(2) For all $fn,e,a,A_1,\ldots,A_n$ such that (e,a) and $(fn[A_1;\ldots;A_n],a)$ are <u>nice</u>:

$$apply(fn^*,(A_1\ldots A_n),a^*)=[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)$$
$$eval(e^*,a^*)=[\![e]\!]([\![a]\!])$$

<u>Proof</u>

The theorem is the conjunction of propositions 4.8.10 and 4.8.13 below.

Q.E.D.

The next lemma lists some elementary properties of 'system functions'. First a definition.

4.8.6 <u>Definition</u>

Define def, list $\in [S^*\text{->}S]$ by:

1. $def(\ )=T,\ def(t_1,\ldots,t_n)=\begin{cases}\perp & \text{if } \exists i.t_i=\perp \\ T & \text{otherwise}\end{cases}$

2. $list(\ )=NIL,\ list(t_1,\ldots,t_n)=cons(t_1,cons(t_2,\ldots,cons(t_n,NIL)\ldots))$

Remark: $def(list(A_1,\ldots,A_n))=T$ if $A_i \in$ <S-expression>

also $def(\perp)=\perp$ and $list(\perp)=\perp$

Finally notice that $def(t_1,\ldots,t_n)=(\lambda x_1\ldots x_n{:}S.T)(t_1,\ldots,t_n)$

4.8.7 <u>Lemma</u>

Just for this lemma let fn,x,a,e,c,m range over S. Then:

<u>4.8.7.1/</u>

4.8.7.1

```
apply(fn,x,a)=(def(x,a)→ (atom(fn)→ (eq(fn,CAR)→ caar(x),
                                     eq(fn,CAR)→ cdar(x),
                                     eq(fn,CONS)→ cons(car(x),cadr(x)),
                                     eq(fn,ATOM)→ atom(car(x)),
                                     eq(fn,EQ)→ eq(car(x),cadr(x)),
                                         T    → apply(eval(fn,a),x,a)),
                           eq(car(fn),LAMBDA)→ eval(caddr(fn),pairlis(
                                                cadr(fn),x,a)),
                           eq(car(fn),LABEL)→ apply(caddr(fn),x,cons(
                                                cons(cadr(fn),caddr(
                                                fn)),a))))
```

4.8.7.2

```
eval(e,a)=(def(a)→ (atom(e)→ cdr(assoc(e,a)),
                    atom(car(e))→ (eq(car(e),QUOTE)→ cadr(e),
                                   eq(car(e),COND)→ evcon(cdr(e),a),
                                         T        → apply(car(e),evlis(
                                                    cdr(e),a),a)),
                      T→ apply(car(e),evlis(cdr(e),a),a)))
```

4.8.7.3

```
evcon(c,a)=(eval(caar(c),a)→ eval(cadar(c),a),
                  T        → evcon(cdr(c),a))
```

4.8.7.4/

4.8.7.4

```
evlis(m,a)=(null(m)→ NIL,
                T     → cons(eval(car(m),a),evlis(cdr(m),a)))
```

4.8.7.5

caar(x)=car(car(x))

4.8.7.6

cdar(x)=cdr(car(x))

4.8.7.7

cadr(x)=car(cdr(x))

4.8.7.8

caddr(x)=car(cdr(cdr(x)))

4.8.7.9

cadar(x)=car(cdr(car(x)))

4.8.7.10

$$\text{null}(x)=\begin{cases}\perp & x=\perp\\ T & x=\text{NIL}\\ F & \text{otherwise}\end{cases}$$

4.8.7.11/

4.8.7.11

$$\text{equal}(x,y)=\begin{cases}\perp & x=\perp \text{ or } y=\perp\\ T & x,y \in \langle\text{S-expression}\rangle \text{ and } x=y\\ F & x,y \in \langle\text{S-expression}\rangle \text{ and } x\neq y\end{cases}$$

4.8.7.12

$$\text{pairlis}((A_1\ldots A_m),(A'_1\ldots A'_n),a)=\begin{cases}((A_1.A'_1)\ldots((A_m.A'_m).a)\ldots) & \text{if } m\leq n\\ \perp & \text{otherwie}\end{cases}$$

4.8.7.13

$$\text{assoc}(X,((X_1.A_1)\ldots(X_n.A_n)))=\begin{cases}(X_m.A_m) & \text{if } X=X_m \text{ and } \forall i<m.X\neq X_i\\ \perp & \text{if } \forall i.X\neq X_i\end{cases}$$

Proof

4.8.7.1 - 4.8.7.13 are all straightforward to prove.. I shall just do 4.8.7.1 as an example.

Suppose FN,X,A $\in$ <S-expression> then if $r=(FN/fn)(X/x)(A/a)[\![int]\!]$ then:

(1)/

(1) $\text{apply}(FN,X,A)=[\![\text{applyfn}]\!]([\![\text{int}]\!])(FN,X,A)$

$$
\begin{aligned}
&=(\text{atom}(FN)\to(\text{eq}(FN,CAR)\to[\![\text{caar}]\!](r)(X),\\
&\qquad\text{eq}(FN,CDR)\to[\![\text{cdar}]\!](r)(X),\\
&\qquad\text{eq}(FN,CONS)\to\text{cons}(\text{car}(X),[\![\text{cadr}]\!](r)(X)),\\
&\qquad\text{eq}(FN,ATOM)\to\text{atom}(\text{car}(X)),\\
&\qquad\text{eq}(FN,EQ)\to\text{eq}(\text{car}(X),[\![\text{cadr}]\!](r)(X)),\\
&\qquad\quad T\ \to[\![\text{apply}]\!](r)([\![\text{eval}]\!](r)(FN,A),X,A)),\\
&\text{eq}(\text{car}(FN),LAMBDA)\to[\![\text{eval}]\!](r)([\![\text{caddr}]\!](r)(FN),[\![\text{pairlis}]\!](r)\\
&\qquad\qquad([\![\text{cadr}]\!](r)(FN),X,A)),\\
&\text{eq}(\text{car}(FN),LABEL)\to[\![\text{apply}]\!](r)([\![\text{caddr}]\!](r)(FN),X,\\
&\qquad\text{cons}(\text{cons}([\![\text{cadr}]\!](r)(FN),[\![\text{caddr}]\!](r)(FN)),A)))
\end{aligned}
$$

(2) Also $\text{apply}(t_1,t_2,t_3)=\perp$ if $t_1=\perp$ or $t_2=\perp$ or $t_3=\perp$ (by corollary 4.3.4)
Now by proposition 4.7.5 (with $Z=\{\text{apply},\text{eval},\text{evcon},\text{evlis},\text{caar},\text{cdar},$ $\text{cadr},\text{caddr},\text{cadar},\text{null},\text{equal},\text{pairlis},\text{assoc}\}$)

We have

$$
\begin{aligned}
\text{caar}&=[\![\text{caar}]\!](\text{int})=[\![\text{caar}]\!](r)\\
\text{cdar}&=[\![\text{cdar}]\!](\text{int})=[\![\text{cdar}]\!](r)\\
\text{cadr}&=[\![\text{cadr}]\!](\text{int})=[\![\text{cadr}]\!](r)\\
\text{apply}&=[\![\text{apply}]\!](\text{int})=[\![\text{apply}]\!](r)\\
\text{eval}&=[\![\text{eval}]\!](\text{int})=[\![\text{eval}]\!](r)\\
\text{caddr}&=[\![\text{caddr}]\!](\text{int})=[\![\text{caddr}]\!](r)\\
\text{pairlis}&=[\![\text{pairlis}]\!](\text{int})=[\![\text{pairlis}]\!](r)
\end{aligned}
$$

Then by (1), (2) and $\text{atom}(\perp)=\perp$ we have 4.8.7.1.

All the other parts of 4.8.7 are similar or easier.

Q.E.D.

I/

I shall now prove some lemmas which, taken together, imply Theorem 4.8.5. (I recommend that you read them only if you think that Theorem is false.)

4.8.8 <u>Lemma</u>

<u>4.8.8.1</u>

a(z)=A => cdr(assoc(z*,a*))=a(z)

<u>4.8.8.2</u>

a(z)=fn => cdr(assoc(z*,a*))=a(z)*

<u>4.8.8.3</u>

$((A_1/x_1)\ldots(A_m/x_m)a)^*=\text{pairlis}((x_1^*\ldots x_m^*),(A_1\ldots A_n),a^*)$ if $m\leq n$

<u>4.8.8.4</u>

$\text{evlis}((e_1^*\ldots e_n^*),a^*)=\text{list}(\text{eval}(e_1^*,a^*),\ldots,\text{eval}(e_n^*,a^*))$

<u>4.8.8.5</u>

$\text{evcon}(((e_{11}^*e_{12}^*)\ldots(e_{n1}^*e_{n2}^*)),a^*)=(\text{eval}(e_{11}^*,a^*)\rightarrow \text{eval}(e_{12}^*,a^*),$
$\ldots,\text{eval}(e_{n1}^*,a^*)\rightarrow \text{eval}(e_{n2}^*,a^*))$

<u>Proof</u>

<u>4.8.8.1</u>/

4.8.8.1

If $a(z)=A$ then a is of the form:

$(v_1/z_1)\dots(v_n/z_n)(A/z)a'$ (where $\forall i.z\neq z_i$ and $v_i \in$ <S-expression> $\cup$ <function>, $a' \in$ <alist>)

and so if v_i^* are the S-expressions translations of v_i

$a^*=\mathrm{cons}((z_1^*.v_1^*),\mathrm{cons}((z_2^*.v_2^*),\dots,\mathrm{cons}((z^*.A),a'^*)\dots))$

$\Rightarrow \mathrm{assoc}(z^*,a^*)=(z^*.A)$

$\Rightarrow \mathrm{cdr}(\mathrm{assoc}(z^*,a^*))=A=a(z)$

4.8.8.2

If $a(z)=fn$ then a is of the form:

$(v_1/z_1)\dots(v_n/z_n)(fn/z)a'$ (where $\forall i.z\neq z_i$ and $v_i \in$ <S-expression> $\cup$ <function>, $a' \in$ <alist>)

and so if v_i^* are the S-expressions translations of v_i

$a^*=\mathrm{cons}((z_1^*.v_1^*),\mathrm{cons}((z_2^*.v_2^*),\dots,\mathrm{cons}((z^*.fn^*),a'^*)\dots))$

$\Rightarrow \mathrm{assoc}(z^*,a^*)=(z^*.fn^*)$

$\Rightarrow \mathrm{cdr}(\mathrm{assoc}(z^*,a^*))=fn^*=a(z)$ *

4.8.8.3

$((a_1/x_1)\dots(A_m/x_m)a)^*=((x_1^*.A_1)\dots((x_m^*.A_m).a^*)\dots)$

$=\mathrm{pairlis}((x_1^*\dots x_m^*),(A_1\dots A_m\dots A_n),a^*)$

4.8.8.4/

4.8.8.4 Induction on n:

n=0: $evlis(NIL,a^*)=NIL=list()$

n>0: assume true for n-1 then:

$evlis((e^*_1 \ldots e^*_n),a^*)=cons(eval(e^*_1,a^*),evlis((e^*_2 \ldots e^*_n),a^*))$

$\qquad =cons(eval(e^*_1,a^*),list(eval(e^*_2,a^*),\ldots,eval(e^*_n,a^*)))$

(by induction)

$\qquad =list(eval(e^*_1,a^*),\ldots,eval(e^*_n,a^*))$

4.8.8.5 Induction on n:

n=1: $evcon(((e^*_{11}e^*_{12})),a^*)=(eval(e^*_{11},a^*)\rightarrow eval(e^*_{12},a^*),T\rightarrow evcon(NIL,a^*))$

$\qquad =(eval(e^*_{11},a^*)\rightarrow eval(e^*_{12},a^*),T\rightarrow \perp)$

$\qquad =(eval(e^*_{11},a^*)\rightarrow eval(e^*_{12},a^*))$

n>1: assume true for n-1 then:

$evcon(((e^*_{11}e^*_{12})\ldots(e^*_{n1}e^*_{n2})),a^*)$

$=(eval(e^*_{11},a^*)\rightarrow eval(e^*_{12},a^*),$

$\qquad T \rightarrow evcon(((e^*_{21}e^*_{22})\ldots(e^*_{n1}e^*_{n2})),a^*))$

$=(eval(e^*_{11},a^*)\rightarrow eval(e^*_{12},a^*),$

$\qquad T \rightarrow (eval(e^*_{21},a^*)\rightarrow eval(e^*_{22},a^*),\ldots,$

$\qquad\qquad eval(e^*_{n1},a^*)\rightarrow eval(e^*_{n2},a^*)))$

$=(eval(e^*_{11},a^*)\rightarrow eval(e^*_{12},a^*),\ldots,$

$\quad eval(e^*_{n1},a^*)\rightarrow eval(e^*_{n2},a^*))$

Q.E.D.

4.8.9/

4.8.9 <u>Lemma</u>

$$(fn[A_1;\ldots;A_n],a) \overset{*}{>} A \Rightarrow apply(fn^*,(A_1\ldots A_n),a^*)=A$$
$$(e,a) \overset{*}{>} A \Rightarrow eval(e^*,a^*)=A$$

<u>Proof</u>

I use LISP-induction. The lemma may be put in the form:

$$p \overset{*}{>} A \Rightarrow R(p,A)$$

by defining:

$$\begin{aligned}R(p,A) \Leftrightarrow{}& \text{if } p=(fn[A_1;\ldots;A_n],a) \overset{*}{>} A \text{ then}\\ & \quad apply(fn^*,(A_1,\ldots,A_n),a^*)=A\\ & \text{and if } p=(e,a) \overset{*}{>} A \text{ then } eval(e^*,a^*)=A.\end{aligned}$$

BASE:

R(A,A) is vacuously true.

INDUCTION:

Assume LIH and $p=(e,a) \overset{*}{>} A$

I show R(p,A) by cases on p (see lemma 4.4.5).

1. $p=(A,a)$

Then $R(p,A) \Leftrightarrow eval(A^*,a^*)=A$

but $eval(A^*,a^*)=eval((QUOTE\ A),a^*)=A$ (by 4.8.7.2)

2./

2. $p=(x,a)$ and $a(x)=A$

Then $R(p,A) \iff eval(x^*,a^*)=A$

but $eval(x^*,a^*)=cdr(assoc(x^*,a^*))=A$ (by 4.8.7.2 and 4.8.8.1)

3. $p=(F[A_1;\ldots;A_n],a)$ and $F(A_1,\ldots,A_n)=A$

Then $R(p,A) \iff apply(F^*,(A_1\ldots A_n),a^*)=A$ and $eval((F^*\ A_1^*\ldots A_n^*),a^*)=A$

Now apply $(F^*,(A_1\ldots A_n),a^*)=A$ (by 4.8.7.1, $F(A_1,\ldots,A_n)\neq \perp$ and $F \in \{car, cdr, cons, atom, eq\}$)

and $eval((F^*\ A_1^*\ldots A_n^*),a^*)$ $=apply(F^*,evlis((A_1^*\ldots A_n^*),a^*),a^*)$ (by 4.8.7.2)

$=apply(F^*,list(eval(A_1^*,a^*),\ldots,eval(A_n^*,a^*)),a^*)$ (by 4.8.8.4)

$=apply(F^*,(A_1\ldots A_n),a^*)$ (as $eval(A_i^*,a^*)=eval((QUOTE\ A_i),a^*)=A_i$)

$=A$

4. $p=(fn[e_1;\ldots;e_n],a)$ and $\forall\ i.(e_i,a) \overset{*}{\Rightarrow} A_i$ and $\exists\ m.e_m \neq A_m$

Then $R(p,A) \iff eval((fn^*\ e_1^*\ldots e_n^*),a^*)=A$

Now by LIH $\forall i.R((e_i,a),A_i)$ and $R((fn[A_1;\ldots;A_n],a),A)$

so $\forall i.eval(e_i^*,a^*)=A_i$ and

$apply(fn^*,(A_1\ldots A_n),a^*)=A$

hence $eval((fn^*\ e_1^*\ldots e_n^*),a^*)=apply(fn^*,evlis((e_1^*\ldots e_n^*),a^*),a^*)$

$=apply(fn^*,(A_1\ldots A_n),a^*)$ (by 4.8.8.4)

$=A.$

5./

5. $p=([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}],a)$ and $(e_{m1},a)\overset{*}{>}T$ and $\forall i<m.(e_{i1},a)\overset{*}{>}F$

Then $R(p,A) \Leftrightarrow$ eval((COND $(e^*_{11}e^*_{12})\ldots(e^*_{n1}e^*_{n2})$), a^*)=A

Now by LIH $R((e_{m1},a),T)$, $\forall i<m.\ R((e_{i1},a),F)$ and $R((e_{m2},a),A)$

so eval(e^*_{m1},a^*)=T, $\forall i<m.$ eval(e^*_{i1},a^*)=F and eval(e^*_{m2},a^*)=A

hence eval((COND $(e^*_{11}e^*_{12})\ldots(e^*_{n1}e^*_{n2})$), a^*)

=evcon((($e^*_{11}e^*_{12})\ldots(e^*_{n1}e^*_{n2}$)), a^*) (by 4.8.7.2)

=(eval(e^*_{11},a^*)$\rightarrow$ eval(e^*_{12},a^*),...,eval(e^*_{n1},a^*)$\rightarrow$ eval(e^*_{n2},a^*))

(by 4.8.8.5)

=eval(e^*_{m2},a^*)

=A.

To simplify the last three cases notice that if $R(p,A) \Leftrightarrow$ apply(fn*, $(A_1\ldots A_n)$, a^*)=A

<u>and</u> eval((fn* $A^*_1\ldots A^*_n$), a^*)=A

then since eval((fn* $A^*_1\ldots A^*_n$), a^*)=apply(fn*, evlis(($A^*_1\ldots A^*_n$), a^*), a^*)

=apply(fn*, $(A_1\ldots A_n)$, a^*) (by 4.8.8.4 and eval(A^*_i,a^*)=A_i)

it follows that:

$R(p,A) \Leftrightarrow$ apply(fn*, $(A_1\ldots A_n)$, a^*)=A

I use this remark in 6, 7, and 8 below.

6./

6. $p=(f[A_1;\ldots;A_n],a)$ and $a(f)=fn$ then $R(p,A) \Leftrightarrow$ apply(f^*,($A_1\ldots A_n$),a^*)=A

Now by LIH $R((fn[A_1;\ldots;A_n],a),A)$ so apply(fn^*,($A_1\ldots A_n$),a^*)=A

and hence apply(f^*,($A_1\ldots A_n$),a^*)

=apply(eval(f^*,a^*),($A_1\ldots A_n$),a^*) (by 4.8.7.1)

=apply(cdr(assoc(f^*,a^*)),($A_1\ldots A_n$),a^*) (by 4.8.7.2)

=apply(fn^*,($A_1\ldots A_n$),a^*) (by 4.8.8.2)

=A

7. $p=(\lambda[[x_1;\ldots;x_m];e][A_1;\ldots;A_n],a)$ and $m\leq n$

Then $R(p,A) \Leftrightarrow$ apply((LAMBDA ($x_1^*\ldots x_m^*$) e^*),($A_1\ldots A_n$),a^*)=A

Now by LIH $R((e,(A_1/x_1)\ldots(A_m/x_m)a),A)$ so

eval(e^*,((A_1/x_1)...(A_m/x_m)a)*)=A

and hence apply((LAMBDA ($x_1^*\ldots x_m^*$) e^*),($A_1\ldots A_n$),a^*)

=eval(e^*,pairlis(($x_1^*\ldots x_m^*$),($A_1\ldots A_n$),a^*)) (by 4.8.7.1)

=eval(e^*,((A_1/x_1)...(A_m/x_m)a)*) (by 4.8.8.3)

=A

8. $p=(\text{Label}[f;fn][A_1;\ldots;A_n],a)$

Then $R(p,A) \Leftrightarrow$ apply((LABEL f^* fn^*),($A_1\ldots A_n$),a^*)=A

Now by LIH $R((fn[A_1;\ldots;A_n],(fn/f)a),A)$ so

apply(fn^*,($A_1\ldots A_n$),((fn/f)a)*)=A

and hence apply((LABEL f^* fn^*),($A_1\ldots A_n$),a^*)

=apply(fn^*,($A_1\ldots A_n$),((f^*.fn^*).a^*)) (by 4.8.7.1)

=apply(fn^*,($A_1\ldots A_n$),((fn/f)a)*)

=A

Q.E.D.

4.8.10/

4.8.10 <u>Proposition</u>

$$\text{apply}(fn^*,(A_1\ldots A_n),a^*) \sqsupseteq [\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)$$
$$\text{eval}(e^*,a^*) \sqsupseteq [\![e]\!]([\![a]\!])$$

<u>Proof</u>

Since S=flat(<S-expression>) if $t_1,t_2 \in S$ then $t_1 \sqsubseteq t_2 \Leftrightarrow t_1=\perp$ or $t_1=t_2$.

Now by the Main Theorem if $[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)\neq\perp$ and $[\![e]\!]([\![a]\!])\neq\perp$ then $(fn[A_1;\ldots;A_n],a)\overset{*}{\Rightarrow}[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)$ and $(e,a)\overset{*}{\Rightarrow}[\![e]\!]([\![a]\!])$ so by 4.8.9 $\text{apply}(fn^*,(A_1\ldots A_n),a^*)=[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)$

$$\text{eval}(e^*,a^*)=[\![e]\!]([\![a]\!])$$

hence the result.

Q.E.D.

Remark: Proposition 4.8.10 means:

$[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)\neq\perp$

$\Rightarrow \text{apply}(fn^*,(A_1\ldots A_n),a\ )=[\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)$

$[\![e]\!]([\![a]\!])\neq\perp \Rightarrow \text{eval}(e^*,a^*)=[\![e]\!]([\![a]\!])$

Thus <u>if</u> denotations are defined <u>then</u> they can be computed with the manual interpreter.

I would have liked the next lemma to have been that if $(fn[A_1;\ldots;A_n],a)$ and (e,a) were nice then:

(apply/

$$(\text{apply}[fn^*;(A_1\ldots A_n);a^*],\text{int})\overset{*}{>}A \Rightarrow [\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)=A$$
$$(\text{eval}[e^*;a^*],\text{int})\overset{*}{>}A \Rightarrow [\![e]\!]([\![a]\!])=A.$$

Unfortunately, though true, this is not in a form which enables a direct proof by LISP-induction to go through. To enable me to get it into a suitable form I make a definition.

4.8.11 Definition

If a, a' $\in$ <alist> then their concatenation a.a' is defined by structural induction by:

$$\text{NIL}.a'=a'$$
$$((A/z).a).a'=(A/z)(a.a')$$
$$((fn/z).a).a'=(fn/z).(a.a')$$

Remark: $((v_1/z_1)\ldots(v_n/z_n)).((v_{n+1}/z_{n+1})\ldots(v_m/z_m))$
$=(v_1/z_1)\ldots(v_n/z_n)(v_{n+1}/z_{n+1})\ldots(v_m/z_m)$

Call an a $\in$ <alist> safe if:

when Z={apply, eval, evcon, evlis, caar, cdar, cadr, caddr, cadar, null, equal, pairlis, assoc}

$$\forall z \in Z.a(z)=\perp$$

Then by corollary 4.7.4 (with Z as above) if a' is safe then:

(apply/

$$(\text{apply}[fn^*;(A_1\ldots A_n);a^*],\text{int})\overset{*}{>}A$$
$$\Leftrightarrow (\text{apply}[fn^*;(A_1\ldots A_n);a^*],a'.\text{int})\overset{*}{>}A$$
$$(\text{eval}[e^*;a^*],\text{int})\overset{*}{>}A$$
$$\Leftrightarrow (\text{eval}[e^*;a^*],a'.\text{int})\overset{*}{>}A$$

Thus the desired lemma mentioned above is equivalent to lemma 4.8.12 below which admits a mechanical LISP-induction proof, unfortunately this proof, though utterly straightforward, is particularly long and tedious - I advise readers to omit it (the lemma is not all that important anyway!).

4.8.12 Lemma

If $(fn[A_1;\ldots;A_n],a)$ and (e,a) are nice and a' is safe then:

$$(\text{apply}[fn^*;(A_1\ldots A_n);a^*],a'.\text{int})\overset{*}{>}A \Rightarrow [\![fn]\!]([\![a]\!])(A_1,\ldots,A_n)=A$$
$$(\text{eval}[e^*;a^*],a'.\text{int})\overset{*}{>}A \Rightarrow [\![e]\!]([\![a]\!])=A$$

Proof

The lemma has the form:

$$p\overset{*}{>}A \Rightarrow R(p,A)$$

If R is defined by:

R(p,A)/

R(p,A) <=> if p=(apply[fnn*;($A_1 \ldots A_n$);aa*],a'.int)$\overset{*}{\Rightarrow}$A

(where a' is safe) then $[\![$fnn$]\!]$($[\![$aa$]\!]$)($A_1,\ldots,A_n$)=A

and if p=(eval[ee*;aa*],a'.int)$\overset{*}{\Rightarrow}$A

(where a' is safe) then $[\![$ee$]\!]$($[\![$aa$]\!]$)=A

(I use "fnn", "ee" and "aa" here to avoid confusion with the names of the formal parameters of apply and eval - see 4.8.1.1 and 4.8.1.2.)

Before doing the LISP-induction I shall get a horrid calculation out of the way by proving a sublemma. I recommend readers to attempt to intuit this rather than to follow the obvious, but messy, proof.

4.8.12.1 <u>Sublemma</u>

(1): If p=(evlis[($e^*_1 \ldots e^*_n$);aa*],a'.int) where a' is safe then there exist safe $a'_1,\ldots,a'_n$ such that:

$\forall$ i.(eval[e^*_i;aa*],a'_i.int)<*p

(2): If p=(evcon[((e^*_{11} e^*_{12})...(e^*_{n1} e^*_{n2}));aa*],a'.int)$\overset{*}{\Rightarrow}$A, for some A, where a' is safe, then there exist safe $a'_{11},\ldots,a'_{m1},a'_{m2}$ such that:

$\forall$ i<m.(eval[e^*_{i1};aa*],a'_{i1}.int)$\overset{*}{\Rightarrow}$F and (eval[e^*_{i1};aa*];aa*],a'_{i1}.int)<*p

(eval[e^*_{m1};aa*],a'_{m1}.int)$\overset{*}{\Rightarrow}$T and (eval[e^*_{m1};aa*].a'_{m1}.int)<*p

(eval[e^*_{m2};aa*],a'_{m2}.int)$\overset{*}{\Rightarrow}$A and (eval[e^*_{m2};aa*],a'_{m2}.int)<*p

<u>Proof</u>

(1):/

(1):

Let a_1'=((e$_1^*$...e$_n^*$)/m)(aa*/a)a' clearly a_1' is safe as a' is, now:

p=(evlis[(e$_1^*$...e$_n^*$);aa*],a'.int)
*>(cons[eval[car[m];a];evlis[cdr[m];a]],a_1'.int) (by 4.8.1.4)

I now do induction on the length n of (e$_1^*$...e$_n^*$).

n=1:

Then (eval[e$_1^*$;aa*],a_1'.int)<*(eval[car[m];a],a_1'.int)<*p

(since (eval[car[m];a],a_1'.int)->(eval[e$_1^*$;aa*],a_1'.int))

n>1: Assume (1) true for (n-1).

Then (evlis[(e$_2^*$...e$_n^*$);aa*],a_1'.int)<*(evlis[cdr[m];a],a_1'.int)<*p

hence result by induction.

(2):

Let a_{11}'=(((e$_{11}^*$e$_{12}^*$)...(e$_{n1}^*$e$_{n2}^*$))/c)(aa*/a)a' then a_{11}' is clearly safe as a' is.

Now:

p=(evcon[((e$_{11}^*$e$_{12}^*$)...(e$_{n1}^*$e$_{n2}^*$));aa*],a'.int)
*>([eval[caar[c];a]→ eval[cadar[c];a];
T → evcon[cdr[c];a]],a_{11}'.int) (by 4.8.1.3)

I now do induction on the length n of ((e$_{11}^*$e$_{12}^*$)...(e$_{n1}^*$e$_{n2}^*$))

n=1:/

n=1:

If it is <u>not</u> the case that

(eval[caar[c];a],a'_{11}.int)*>T

then it is clearly not possible for p*>A but we assumed this so we must in fact have (eval[caar[c];a],a'_{11}.int)*>T.

Thus as (eval[caar[c];a],a'_{11}.int)->(eval[e^*_{11};aa*],a'_{11}.int)
we must have

(eval[e^*_{11};aa*],a'_{11}.int)*>T, and
(eval[e^*_{11};aa*],a'_{11}.int)<*(eval[caar[c];a],a'_{11}.int)<*p
and as (eval[cadar[c];a],a'_{11}.int)->(eval[e^*_{12};aa*],a'_{11}.int)
we must also have

(eval[e^*_{12};aa*],a'_{11}.int)*>A (as p*>A), and
(eval[e^*_{12};aa*],a'_{11}.int)<*p

so taking $a'_{12}=a'_{11}$ renders (2) true.

n>1: Assume (2) true for (n-1).
If (eval[caar[c];a] , a'_{11}.int)*>T then proceed as above in the "n-1" case, otherwise for p*>A we must have (eval[caar[c];a],a'_{11}.int)*>F
and then

p*/

$p*>(eval[caar[c];a],a'_{11}.int)->(eval[e^*_{11};aa^*],a'_{11}.int)\overset{*}{=}>F$

and

$p*>(evcon[cdr[c];a],a'_{11}.int)$

$->(evcon[((e^*_{21}e^*_{22})...(e^*_{n1}e^*_{n2}));aa^*],a'_{11}.int)\overset{*}{=}>A$

and using induction (2) then follows

Q.E.D.

Now I can get on with the LISP-induction. Note that if p is not of the form (apply[fnn;(A_1...A_n);aa*],a'.int) or (eval[ee*;aa*],a'.int) (where a' is safe) then R(p,A) is vacuously true - this observation copes with the base of the LISP-induction.

BASE:

R(A,A) is vacuously true.

INDUCTION:

Assume LIH and p=(e,a)$\overset{*}{=}$>A where p is of one of the forms mentioned above. I show R(p,A) by cases on p.

1. p=(apply[fnn*;(A_1...A_n);aa*],a'.int)

Then R(p,A) <=> ⟦fnn⟧(⟦aa⟧)(A_1,...,A_n)=A

Let a"=(fnn*/fn)((A_1...A_n)/x)(aa*/a)a' then as a' is safe so is a". I show R(p,A) by considering the various possible cases for p.

1.1. fnn=F and F(A_1,...,A_n)=A.

Then clearly ⟦fnn⟧(⟦aa⟧)(A_1,...,A_n)=A.

1.2./

1.2. fnn=f

Then by the niceness assumption aa(f)=fn' for some fn'. Now

p*>(apply[eval[f*;a];x;a],a",int)

 ->(apply[fn'*;($A_1 \ldots A_n$);aa*],a".int)

(since $\llbracket$eval[f*;a]$\rrbracket$($\llbracket$a".int$\rrbracket$)=fn'*so by the Main Theorem (eval[f;a],a".int)*>fn'*)

Hence by LIH R((apply[fn'*;($A_1 \ldots A_n$);aa*],a".int),A) so $\llbracket$fn'$\rrbracket$($\llbracket$aa$\rrbracket$)($A_1,\ldots,A_n$)=A

and thus

$$\llbracket fnn \rrbracket(\llbracket aa \rrbracket)(A_1,\ldots,A_n)=\llbracket aa \rrbracket(f)(\llbracket aa \rrbracket)\mid FUN(A_1,\ldots,A_n)$$
$$=\llbracket fn' \rrbracket(\llbracket aa \rrbracket)(A_1,\ldots,A_n)$$
$$=A$$

1.3. fnn= λ [[$x_1;\ldots;x_m$];ee]

Then p*>(eval[caddr[fn];pairlis[cadr[fn];x;a]],a".int)

Now $\llbracket$caddr[fn]$\rrbracket$($\llbracket$a".int$\rrbracket$)=ee*

$\llbracket$pairlis[cadr[fn];x;a]$\rrbracket$($\llbracket$a".int$\rrbracket$)=pairlis(cadr(fnn*),($A_1 \ldots A_n$),aa*)

=pairlis(($x_1^* \ldots x_m^*$),($A_1 \ldots A_n$),aa*)

as we are assuming p*>A it follows (from $\llbracket p \rrbracket \neq \perp$ and the Main Theorem) that pairlis(($x_1^* \ldots x_m^*$),($A_1 \ldots A_n$),aa*)$\neq\perp$ and hence $m \leq n$. Also, by 4.8.8.3,

$\llbracket$pairlis[cadr[fn];x;a]$\rrbracket$($\llbracket$a".int$\rrbracket$)=((A_1/x_1)...(A_m/x_m)aa)*

Thus

p/

$p \overset{*}{>} (\text{eval}[\text{cadr}[\text{fn}];\text{pairlis}[\text{cadr}[\text{fn}];x;a]],a''.\text{int})$

$-> (\text{eval}[ee^*;((A_1/x_1)\ldots(A_m/x_m)aa)^*],a''.\text{int})$

$\overset{*}{>} A$

Hence by LIH $[\![ee]\!]([\![(A_1/x_1)\ldots(A_m/x_m)aa]\!])=A.$

But then:

$[\![fnn]\!]([\![aa]\!])(A_1,\ldots,A_n)$

$=(\lambda t_1\ldots t_m.[\![ee]\!]((t_1/x_1)\ldots(t_m/x_m)[\![aa]\!]))(A_1,\ldots,A_n)$

$=[\![ee]\!]((A_1/x_1)\ldots(A_m/x_m)[\![aa]\!])$ (as $m\leq n$)

$=[\![ee]\!]([\![(A_1/x_1)\ldots(A_m/x_m)aa]\!])$

$=A.$

1.4. fnn=Label[f;fn']

Then $p^*>(\text{apply}[\text{caddr}[\text{fn}];x;\text{cons}[\text{cons}[\text{cadr}[\text{fn}];\text{caddr}[\text{fn}]];a]],a'',\text{int})$

$->(\text{apply}[fn'^*;(A_1\ldots A_n);((fn'/f)aa)^*],a''.\text{int})$

Hence by LIH $[\![fn']\!]([\![(fn'/f)aa]\!])(A_1,\ldots,A_n)=A$

But then $[\![fnn]\!]([\![aa]\!])(A_1,\ldots,A_n)$

$=[\![fn']\!](([\![fn']\!]/f)[\![aa]\!])(A_1,\ldots,A_n)$

$=[\![fn']\!]([\![fn'/f)aa]\!])(A_1,\ldots,A_n)$

$=A.$

2./

2. p=(eval[ee*;a*],a'.int)

Then R(p,A) <=> ⟦ee⟧(⟦a⟧)=A.

Let a"=(ee*/e)(aa*/a)a' - clearly as a' is safe so is a". I show R(p,A) by considering the various possible cases for ee.

2.1. ee=A'.

Then p*>A' so A=A' and hence ⟦ee⟧(⟦a⟧)=A.

2.2. ee=x.

Then by the niceness assumption aa(x)=A' (some A') and p*>(cdr[assoc[e;a]],a".int)*>A' so A'=A and ⟦ee⟧(⟦aa⟧)=A.

2.3. ee=fnn[e_1;...;e_n]

Then p*>(apply[car[e];evlis[cdr[e];a];a],a".int)

Now by sublemma 4.8.12.1 (1) there are safe $a''_1,\ldots,a''_n$ such that for each i:

$$\begin{aligned}(\text{eval}[e^*_i;aa^*],a''_i.\text{int}) &<^*(\text{evlis}[(e^*_1\ldots e^*_n);aa^*],a''.\text{int})\\ &<^*(\text{evlis}[\text{cdr}[e];a],a''.\text{int})\\ &<^*p\end{aligned}$$

Hence by LIH $\forall i.⟦e_i⟧(⟦aa⟧)=\text{eval}(e^*_i,aa^*)\neq\perp$ (as p*>A$\neq\perp$) so:

p*>(apply[fnn*;(eval(e^*_1,aa*)...eval(e^*_n,aa*));aa*],a".int) (by Main Theorem and 4.8.8.4)

Hence/

Hence by LIH:

$$[\![fnn]\!]([\![aa]\!])(eval(e_1^*,aa^*),\dots,eval(e_1^*,aa^*))=A$$

but then:

$$\begin{aligned}&[\![fnn[e_1;\dots;e_n]]\!]([\![aa]\!])\\&\quad=[\![fnn]\!]([\![aa]\!])([\![e_1]\!]([\![aa]\!]),\dots,[\![e_n]\!]([\![aa]\!]))\\&\quad=[\![fnn]\!]([\![aa]\!])(eval(e_1^*,aa^*),\dots,eval(e_n^*,aa^*))\\&\quad=A.\end{aligned}$$

2.4. $ee=[e_{11}\rightarrow e_{12};\dots;e_{n1}\rightarrow e_{n2}]$

Then $p^*{>}(evcon[cdr[e];a],a''.int)$

$$\begin{aligned}&\rightarrow{>}(evcon[((e_{11}^*e_{12}^*)\dots(e_{n1}^*e_{n2}^*));aa^*],a''.int)\\&\overset{*}{>}A\end{aligned}$$

So by sublemma 4.8.12.1 (2) there is an m and safe a''_{ij} such that:

$$\begin{aligned}\forall i<m.\ &p^*{>}(eval[e_{i1}^*;aa^*],a''_{i1}.int)\overset{*}{>}F\\&p^*{>}(eval[e_{m1}^*;aa^*],a''_{m1}.int)\overset{*}{>}T\\&p^*{>}(eval[e_{m2}^*;aa^*],a''_{m2}.int)\overset{*}{>}A\end{aligned}$$

Hence by LIH

$\forall i/$

$$\forall\, i<m.\ \llbracket e_{i1}\rrbracket(\llbracket aa\rrbracket)=F$$
$$\llbracket e_{m1}\rrbracket(\llbracket aa\rrbracket)=T$$
$$\llbracket e_{m2}\rrbracket(\llbracket aa\rrbracket)=A$$

so $\llbracket ee\rrbracket(\llbracket aa\rrbracket)=(\llbracket e_{11}\rrbracket(\llbracket aa\rrbracket)\rightarrow \llbracket e_{12}\rrbracket(\llbracket aa\rrbracket),\ldots,\llbracket e_{n1}\rrbracket(\llbracket aa\rrbracket)\rightarrow \llbracket e_{n2}\rrbracket(\llbracket aa\rrbracket))$
$=A$

Q.E.D.

4.8.13 <u>Proposition</u>

For all $fn,e,A_1,\ldots,A_n$ such that $(fn[A_1;\ldots;A_n],a)$ and (e,a) are nice:

$$apply(fn^*,(A_1\ldots A_n),a^*)=\llbracket fn\rrbracket(\llbracket a\rrbracket)(A_1,\ldots,A_n)$$
$$eval(e^*,a^*)=\llbracket e\rrbracket(\llbracket a\rrbracket)$$

<u>Proof</u>

Taking a'=NIL in lemma 4.8.12 yields (via the Main Theorem):

$$apply(fn^*,(A_1\ldots A_n),a^*)\sqsubseteq \llbracket fn\rrbracket(\llbracket a\rrbracket)(A_1,\ldots,A_n)$$
$$eval(e^*,a^*)\sqsubseteq \llbracket e\rrbracket(\llbracket a\rrbracket)$$

The result follows from this and proposition 4.8.10.

Q.E.D.

Combining proposition 4.8.10 and proposition 4.8.13 yields theorem 4.8.5.

5. PROOF OF THE MAIN THEOREM

5.1 Outline of the essential idea

The essential idea underlying my proof of the Main Theorem is derived from Christopher Wadsworth's work on Scott's semantics of the λ-calculus [25] (however I do not need the full power of Wadsworth's technique and I suspect and hope that a simpler proof is possible). Very roughly the theorem is first proved for a class of 'finite' programs and then extended to all programs by a limiting argument. In view of lemma 4.5.1 I only need to prove that $\{p\}=A \Rightarrow p \overset{*}{>} A$, note that this is not in a form suitable for LISP-induction. The main steps in the proof are:

5.1.1

I extend the languages LISP and ALIST to larger languages ELISP and EALIST. This is done by allowing functions to have indices attached to them, the idea being that a function with an index n can only be called recursively to depth n - if more than n recursive calls are attempted the result is undefined. Functions with finite indices should be thought of as 'finite' approximations to the corresponding ones with infinite indices. I extend the semantics of 3.14 to ELISP and EALIST in a way harmonious with (but not identical to) the above intuition and I also extend -> so that indices are manipulated correctly in reductions.

5.1.2/

5.1.2

I define a relation $<$ on the expressions of ELISP and EALIST. $e'<e$ will mean (roughly!) that e is a LISP form and e' is got from e by adding indices at various places, similarly for $a'<a$. $p'<p$ will mean that either $p=p'=A$ or $p=(e,a)$, $p'=(e',a')$ and $e'<e, a'<a$. Thus $p'<p$ means p' is a 'finite' approximation to p.

5.1.3

I will then prove five lemmas - these should be plausible given the above intuitions - they are:

5.1.3.1 $[\![p]\!]=\bigsqcup_{p'<p}[\![p']\!]$

5.1.3.2 $p'<p \Rightarrow p'$ terminates (i.e. there is a p'' such that $p'\overset{*}{\rightarrow}p''$ and for no p''', $p''\rightarrow p'''$)

5.1.3.3 ($p'<p, p'\overset{*}{\rightarrow}p'', p''\neq A''$ and for no p''' does $p''\rightarrow p'''$) $\Rightarrow [\![p']\!]=\perp$

5.1.3.4 $p\overset{*}{\rightarrow}A \Rightarrow [\![p]\!]=A$

5.1.3.5 $p'<p.p'\overset{*}{\rightarrow}A \Rightarrow p\overset{*}{\rightarrow}A$.

5.1.4

From the lemmas of 5.1.3 above $[\![p]\!]=A \Rightarrow p\overset{*}{\rightarrow}A$ can be proved as follows:

If $[\![p]\!]=A$ then by 5.1.3.1 $A=[\![p]\!]=\bigsqcup_{p'<p}[\![p']\!]$ and so as S is flat $A=[\![p']\!]$ for some $p'<p$. By 5.1.3.2 $p'\overset{*}{\rightarrow}p''$ where for no p''' does $p''\rightarrow p'''$; if $p''\neq A''$ for/

for any A" then by$\overset{5.1.}{\wedge}$3.3 $A=[\![p']\!]=\perp$ which is impossible (A ranges over <S-expression> not over S) hence p"=A" for some A" and so by$\overset{5.1.}{\wedge}$3.4 $A''=[\![p']\!]=A$. Thus we have p'<p,p'*>A and so by$\overset{5.1.}{\wedge}$3.5 p*>A.

Q.E.D.

Before I can convert this outline into a rigorous argument I have to say something about the intended solution of Env=Id->[Env->D] - this knowledge is necessary in order to give a precise meaning to the indices. In the next section I shall describe and motivate enough of the properties of this solution to enable the proof of the Main Theorem to be carried out. The full construction of Env occurs in chapter 6. Until this construction is done one cannot know for sure that the properties I want Env to have are consistent; for this reason doing it is necessary.

5.2 Further specification of Env

I want now to appeal to your operational intuition to motivate some axioms I am going to require Env to satisfy, these axioms are given in 5.2.1 below - if you get fed up with the waffle that follows skip to that section.

In order to do the semantics of ELISP and EALIST I need to say what "indexed" functions denote. If fn_m is fn indexed by $m \geq 0$ then I want:

$$[\![fn_m]\!]/$$

$$[\![fn_m]\!](r)(A_1,\dots,A_n)= \begin{cases} [\![fn]\!](r)(A_1,\dots,A_n) & \text{if this can be 'evaluated' with function calls to depth m or less.} \\ \perp & \text{otherwise.} \end{cases}$$

Since $[\![fn_0]\!](r)$ cannot do any looking-up in r (because looking-up functions is calling them) we would expect $[\![fn_0]\!](r)=[\![fn]\!](\perp)$ (at least if fn has no free <u>form</u> variables). Also if $r=([\![fn^1]\!]/f^1)\dots([\![fn^s]\!]/f^s)$ then we would expect $[\![fn_{m+1}]\!](r)\overset{=[\![fn]\!]}{\wedge}(([\![fn^1_m]\!]/f^1)\dots([\![fn^s_m]\!]/f^s)\ )$ since if fn_{m+1} needs to call one of the fn^i $(1\leq i\leq s)$ then this uses up one of fn_{m+1}'s(m+1) calls and so there is only depth m left for fn^i. Now if we define $r_0=\perp$, and for $r=([\![fn^1]\!]/f^1)\dots([\![fn^s]\!]/f^s)$ we define $r_{m+1}=([\![fn^1_m]\!]/f^1)\dots([\![fn^s_m]\!]/f^s)$ then the remarks above can be expressed as:

$$\forall\, m\geq 0.[\![fn_m]\!](r)=[\![fn]\!](r_m)$$

I now shift my attention from indexed functions to indexed environments and derive four properties of these: for the first one

Let $r=([\![fn^1]\!]/f^1)\dots([\![fn^s]\!]/f^s)$

$r'=([\![gn^1]\!]/f^1)\dots([\![gn^s]\!]/f^s)$

Then $r_{m+1}(f^i)(r')=[\![fn^i_m]\!](r')$ (by definition of r_{m+1})

$=[\![fn^i]\!](r'_m)$ (by remarks above)

$=r(f^i)(r'_m)$ (by definition of r)

This/

This suggests that it would be reasonable to require that:

(1) $\forall\, r,z,r'.\ r_{m+1}(z)(r')=r(z)(r'_m)$

For the second property observe that it seems reasonable from the intended meaning of fn_m that:

$$[\![fn_0]\!] \sqsubseteq [\![fn_1]\!] \sqsubseteq \dots \sqsubseteq [\![fn_m]\!] \sqsubseteq \dots \sqsubseteq [\![fn]\!]$$

hence from the definition of r_m:

(2) $\perp = r_0 \sqsubseteq r_1 \sqsubseteq \dots \sqsubseteq r_m \sqsubseteq \dots \sqsubseteq r$

Thirdly notice that if $[\![fn]\!](r)(A_1,\dots,A_n)$ is defined then there is some maximum depth m of function calls done in its evaluation so that

$$[\![fn]\!](r)(A_1,\dots,A_n)=[\![fn_m]\!](r)(A_1,\dots,A_n)$$

and hence by considering all possible computations it seems reasonable that

$$[\![fn]\!]=\bigsqcup_m [\![fn_m]\!]$$

and so

(3) $r=\bigsqcup_m r_m$

Finally we would expect that $[\![(fn_m)_m]\!]=[\![fn_m]\!]$ and hence

(4) $(r_m)_m=r_m$

The intuitions which I have just exploited to derive (1)-(4) are not captured/

captured by the equation Env=Id->[Env->D]; extra axioms are needed to build them in and these I now state.

5.2.1 Axioms

An alist model is a semi-domain Env together with, for $n \geq 0$, mappings $r \mapsto r_n \in$ [Env->Env] such that:

(Env1) $\mathrm{Env} \cong [\mathrm{Id} \to [\mathrm{Env} \to D]]$ ($D = S + [S^* \to S]$ as in 3.14)

(Env2) $\perp = r_0 \sqsubseteq r_1 \sqsubseteq \ldots \sqsubseteq r_n \sqsubseteq \ldots \sqsubseteq r$

(Env3) $r = \bigsqcup_n r_n$

(Env4) $(r_n)_n = r_n$

(Env5) $r_{n+1}(z)(r') = r(z)(r'_n)$

Remark: If Env is an alist model then by Env1 there exist semi-domain isomorphisms f: Env->[Id->[Env->D]], g: [Id->[Env->D]]->Env, I adopt the convention of identifying 'across' f and g so that for $r, r' \in$ Env $r(z)(r')$ really means $f(r)(z)(r')$ etc.

How fully do these axioms specify alist models? I shall show that if Env', Env" are alist models then there are semi-domain isomorphisms between them which preserve not only directed limits but also the application and indexing structure postulated in 5.2.1. Thus there is really only one alist model and so requiring Env to be that renders my semantics completely unambiguous. In 5.2.2.2 below the kind of isomorphism/

isomorphism, up to which alist models are unique, is defined. Readers not interested in this uniqueness (which is not used in the proof of the Main Theorem) should skip to definition 5.2.6.

5.2.2 Definition

5.2.2.1

If Env', Env" are alist models and f: Env'->Env" then f is an homomorphism <=> (1) f is continuous

(2) $\forall r,r' \in Env'. \forall z \in Id. f(r)(z)(f(r'))=r(z)(r')$

(3) $\forall r' \in Env'. \forall n\geq 0. f(r'_n)=f(r')_n$.

5.2.2.2

Alist models Env', Env" are isomorphic <=> there exist homomorphisms f: Env'->Env", g: Env"->Env' which are inverse to each other (i.e. $f\circ g=I_{Env''}$, $g\circ f=I_{Env'}$).

5.2.2.3

Let $Env_0=\{\perp\}$

$Env_{n+1}=[Id->[Env_n->D]]$

5.2.3 Lemma

If Env is an alist model then for each $n\geq 0$ $\{r \in Env \mid r=r_n\}$ is a sub semi-domain of Env and there exist semi-domain isomorphisms

f_n:/

f_n: $Env_n \to \{r \in Env \mid r = r_n\}$

g_n: $\{r \in Env \mid r = r_n\} \to Env_n$

inverse to each other and such that $\forall z \in Id$:

(1) $\forall r \in Env_{n+1}.\ \forall r' \in Env_n.\ f_{n+1}(r)(z)(f_n(r')) = r(z)(r')$

(2) $\forall r \in \{r \mid r = r_{n+1}\}.\ \forall r' \in \{r' \mid r' = r'_n\}.\ g_{n+1}(r)(z)(g_n(r')) = r(z)(r')$

Proof

Define f_n: $Env_n \to \{r \mid r = r_n\}$

g_n: $\{r \mid r = r_n\} \to Env_n$

by $f_0(r) = \perp$, $f_{n+1}(r) = \lambda z{:}Id.\ \lambda r'{:}\ Env.r(z)(g_n(r'_n))$

$g_0(r) = \perp$, $g_{n+1}(r) = \lambda z{:}Id.\ \lambda r'{:}\ Env_n.r(z)(f_n(r'))$

$f_n(r)$ is indeed a member of $\{r \mid r = r_n\}$ since $(f_0(r))_0 = \perp_0 = \perp = f_0(r)$

and for $n \geq 0$

$$\begin{aligned}(f_{n+1}(r))_{n+1} &= \lambda z.\ \lambda r'.\ f_{n+1}(r)_{n+1}(z)(r') && \text{(by extensionality)}\\ &= \lambda z.\ \lambda r'.\ f_{n+1}(r)(z)(r'_n) && \text{(by Env5)}\\ &= \lambda z.\ \lambda r'.\ r(z)(g_n(r'_{nn})) && \text{(by definition of } f_{n+1})\\ &= \lambda z.\ \lambda r'.\ r(z)(g_n(r'_n)) && \text{(by Env4)}\\ &= f_{n+1}(r) && \text{(by definition of } f_{n+1})\end{aligned}$$

I now show, by induction on n, that: $\forall r \in \{r \mid r = r_n\}.\ f_n(g_n(r)) = r$

and $\forall r \in Env_n.\ g_n(f_n(r)) = r$.

n=0:/

n=0: $r=r_0 \Rightarrow r=\perp \Rightarrow f_0(g_0(r))=\perp=r$

$r \in Env_n \Rightarrow r=\perp \Rightarrow g_0(f_0(r))=\perp=r$

n>0: Assume $f_{n-1}(g_{n-1}(r))=r$, $g_{n-1}(f_{n-1}(r))=r$ then:

$f_n(g_n(r))= \lambda z.\ \lambda r'.\ f_n(g_n(r))(z)(r')$ (by extensionality)

$= \lambda z.\ \lambda r'.\ g_n(r)(z)(g_{n-1}(r'_{n-1}))$ (by definition of f_n)

$= \lambda z.\ \lambda r'.\ r(z)(f_{n-1}(g_{n-1}(r'_{n-1})))$ (by definition of g_n)

$= \lambda z.\ \lambda r'.\ r(z)(r'_{n-1})$ (by induction)

$= \lambda z.\ \lambda r'.\ r_n(z)(r')$ (by Env5)

$=r_n$ (by extensionality)

so if $r=r_n$ then $f_n(g_n(r))=r_n=r$. Also we have:

$g_n(f_n(r))= \lambda z.\ \lambda r'.\ g_n(f_n(r))(z)(r')$ (by extensionality)

$= \lambda z.\ \lambda r'.\ f_n(r)(z)(f_{n-1}(r'))$ (by definition of g_n)

$= \lambda z.\ \lambda r'.\ r(z)(g_{n-1}(f_{n-1}(r')_{n-1}))$ (by definition of f_n)

$= \lambda z.\ \lambda r'.\ r(z)(g_{n-1}(f_{n-1}(r')))$ (as $f_{n-1}(r')_{n-1}=f_{n-1}(r')$)

$= \lambda z.\ \lambda r'.\ r(z)(r')$ (by induction)

$=r$

Thus f_n, g_n are isomorphisms inverse to each other. To check (1) we have

$$f_{n+1}(r)(z)(f_n(r'))=r(z)(g_n(f_n(r')_n))=r(z)(g_n(f_n(r')))=r(z)(r')$$

to check (2): $g_{n+1}(r)(z)(g_n(r'))=r(z)(f_n(g_n(r')))=r(z)(r')$

Q.E.D.

5.2.4/

5.2.4 Proposition

If Env', Env" are alist models then they are isomorphic (in the sense of 5.2.2.2).

Proof

By the previous lemma:

$$\{r' \in \text{Env}' \mid r'=r'_n\} \cong \text{Env}_n \cong \{r'' \in \text{Env}'' \mid r''=r''_n\}$$

Let f'_n: $\text{Env}_n \to \{r' \in \text{Env}' \mid r'=r'_n\}$, f''_n: $\text{Env}_n \to \{r'' \in \text{Env}'' \mid r''=r''_n\}$
g'_n: $\{r' \in \text{Env}' \mid r'=r'_n\} \to \text{Env}_n$, g''_n: $\{r'' \in \text{Env}'' \mid r''=r''_n\} \to \text{Env}_n$

be the isomorphisms defined in the proofs of the previous lemma.

Define f: Env'->Env"

g: Env"->Env'

by $f(r') = \bigsqcup_n f''_n(g'_n(r'_n))$

$g(r'') = \bigsqcup_n f'_n(g''_n(r''_n))$

I claim f,g are isomorphisms. To see that they are well defined observe that

$$\begin{aligned} f''_{n+1}(g'_{n+1}(r'_{n+1})) &= \lambda z.\ \lambda r''.\ f''_{n+1}(g'_{n+1}(r'_{n+1}))(z)(r'') && \text{(extensionality)} \\ &= \lambda z.\ \lambda r''.\ g'_{n+1}(r'_{n+1})(z)(g''_n(r''_n)) && \text{(definition of } f''_{n+1}\text{)} \\ &= \lambda z.\ \lambda r''.\ r'_{n+1}(z)(f'_n(g''_n(r''_n))) && \text{(definition of } g'_{n+1}\text{)} \\ &= \lambda z.\ \lambda r''.\ r'(z)(f'_n(g''_n(r''_n))) && \text{(by Env5 and Env4)} \end{aligned}$$

and similarly $f'_{n+1}(g''_{n+1}(r''_{n+1})) = \lambda z.\ \lambda r'.\ r''(z)(f''_n(g'_n(r'_n)))$

so/

so by a quick induction on n we have

$$\perp = f''_0(g'_0(r'_0)) \sqsubseteq f''_1(g'_1(r'_1)) \sqsubseteq \dots \sqsubseteq f''_n(g'_n(r'_n)) \sqsubseteq \dots f(r') \quad (1)$$
$$\perp = f'_0(g''_0(r''_0)) \sqsubseteq f'_1(g''_1(r''_1)) \sqsubseteq \dots \sqsubseteq f'_n(g''_n(r''_n)) \sqsubseteq \dots g(r'') \quad (2)$$

and so the $\bigsqcup$'s defining f,g are over directed sets and hence are defined.

To show that f,g are inverses we have

$$\begin{aligned}
f(g(r'')) &= f(\bigsqcup_n f'_n(g''_n(r''_n))) \\
&= \bigsqcup_m \bigsqcup_n f''_m(g'_m((f'_n(g''_n(r''_n)))_m)) \\
&= \bigsqcup_n f''_n(g'_n((f'_n(g''_n(r''_n)))_n)) && \text{(by (1), (2) above, i.e. cofinality)} \\
&= \bigsqcup_n f''_n(g'_n(f'_n(g''_n(r''_n)))) && \text{(as } f'_n(r')_n = f'_n(r')) \\
&= \bigsqcup_n f''_n(g''_n(r''_n)) && \text{(as } g'_n(f'_n(r')) = r') \\
&= \bigsqcup_n r''_n && \text{(as } f''_n(g''_n(r''_n)) = r''_n) \\
&= r'' && \text{(by Env3)}
\end{aligned}$$

and by symmetry $g(f(r'))=r'$.

To show f is a homomorphism we have:

$$\begin{aligned}
f(r)(z)(f(r')) &= \bigsqcup_n f''_{n+1}(g'_{n+1}(r_{n+1}))(z)(f''_n(g'_n(r'_n))) && \text{(by (1) above - i.e. cofinality)} \\
&= \bigsqcup_n g'_{n+1}(r_{n+1}(z)(g'_n(r'_n)) && \text{(by lemma 5.2.3)} \\
&= \bigsqcup_n r_{n+1}(z)(r'_n) && \text{(by lemma 5.2.3)} \\
&= r(z)(r') && \text{(by Env5, Env4 and Env3)}
\end{aligned}$$

and/

and also $f(r_0)= \perp =f(r)_0$ and if $\forall r'. f(r'_n)=f(r')_n$

then: $f(r_{n+1})(z)(f(r'))=r_{n+1}(z)(r')$ (by the last calculation)

$=r(z)(r'_n)$ (by Env5)

$=f(r)(z)(f(r'_n))$ (by the last calculation)

$=f(r)(z)(f(r')_n)$ (by assumption)

$=f(r)_{n+1}(z)(f(r'))$ (by Env5)

so by induction on n and the fact that f is onto $(r''=f(g(r'')))$

$$\forall n.\ f(r_n)=f(r)_n$$

Thus f is a homomorphism and so, by symmetry, is g also, hence Env', Env" are isomorphic.

Q.E.D.

In view of this proposition it makes sense to talk about Env as the alist model.

The next proposition shows that any semi-domain which satisfies $Env \cong Env\to[Env\to D]$ contains an alist model as a sub semi-domain, thus alist models are (sort of) minimal solutions of $Env=Id\to[Env\to D]$ since minimal solutions of recursion equations are the ones wanted this provides another reason why alist models are appropriate. It also shows that to prove Env1-Env5 are consistent I only have to prove that (Env1) is.

5.2.5/

5.2.5 Proposition

Let Env=[Id->[Env->D]] then Env contains a sub semi-domain which is an alist model.

Proof

For r $\in$ Env let $r_0=\perp$

$$r_{n+1}=\lambda z.\ \lambda r'.\ r(z)(r_n)$$
$$r_\infty=\bigsqcup_n r_n=\lambda z.\ \lambda r'.\ r(z)(r'_\infty)$$
$$\text{Env}_\infty=\{r \mid r=r_\infty\}$$

Then I claim that Env_∞ is a sub semi-domain of Env satisfying (Env1)-(Env5). It is a sub semi-domain of Env since $\perp_\infty=\lambda z.\ \lambda r'.\ \perp(z)(r'_\infty)=\perp$ and if $\{r^i\}_i \subseteq \text{Env}_\infty$ is directed then $(\bigsqcup_i r^i)_\infty=\bigsqcup_i r^i_\infty=\bigsqcup_i r^i$ (as $\forall i.r^i=r^i_\infty$) so $\bigsqcup_i r^i \in \text{Env}_\infty$. To check (Env1)-(Env5) is straightforward:

Env1:

Note that $r_{\infty\infty}=\bigsqcup_n(\bigsqcup_m r_m)_n=\bigsqcup_n\bigsqcup_m r_{nm}=\bigsqcup_n r_{nn}=\bigsqcup_n r_n=r_\infty$ so that if f: $[\text{Id}\to[\text{Env}_\infty\to D]]\to\text{Env}_\infty$ is defined by

$$f(r)=(\lambda z.\ \lambda r'.\ r(z)(r'_\infty))_\infty$$

then $f(r)(z)(r')=(\lambda z.\ \lambda r'.\ r(z)(r'_\infty))_\infty(z)(r')$

$$=(\lambda z.\ \lambda r'.\ r(z)(r'_\infty))(z)(r'_\infty)$$
$$=r(z)(r'_{\infty\infty})$$
$$=r(z)(r'_\infty)$$

and/

and hence the mappings

$$(\lambda r\colon \mathrm{Env}_\infty . r)\colon \mathrm{Env}_\infty \to [\mathrm{Id} \to [\mathrm{Env}_\infty \to D]]$$
$$f \quad \colon [\mathrm{Id} \to [\mathrm{Env}_\infty \to D]] \to \mathrm{Env}_\infty$$

are isomorphisms.

Env2:

By definition $\perp = r_0$, clearly $r_0 \sqsubseteq r_1$. Assume: $r_n \sqsubseteq r_{n+1}$ then

$$\begin{aligned} r_{n+1} &= \lambda z.\ \lambda r'.\ r(z)(r'_n) && \text{(by definition of } r_{n+1}) \\ &\sqsubseteq \lambda z.\ \lambda r'.\ r(z)(r'_{n+1}) && \text{(by assumption)} \\ &= r_{n+2} && \text{(by definition of } r_{n+2}) \end{aligned}$$

hence by induction on n $\perp = r_0 \sqsubseteq r_1 \sqsubseteq \ldots \sqsubseteq r_n \sqsubseteq \ldots \sqsubseteq r_\infty$.

Env3:

$r \in \mathrm{Env}_\infty \Rightarrow r = r_\infty = \bigsqcup_n r_n$.

Env4:

$$\begin{aligned} r_{00} = \perp = r_0,\ (r_{n+1})_{n+1} &= \lambda z.\ \lambda r'.\ r_{n+1}(z)(r'_n) \\ &= \lambda z.\ \lambda r'.\ r(z)(r'_{nn}) \end{aligned}$$

hence by induction on n: $\forall n.\ (r_n)_n = r_n$.

Env5:

$r_{n+1}/$

$r_{n+1}(z)(r')=r(z)(r'_n)$ by the definition of r_{n+1}.

Q.E.D.

Certain simple properties of alist models are needed for the proof of the Main Theorem, the next proposition collects them together.

5.2.6 Definition

For $r \in$ Env let $r_\infty=r$ and for $0\leq n\leq\infty$, $v \in$ [Env->D] define $v_n \in$ [Env->D] by:

$$v_n=\lambda r.\ v(r_n)$$

Remark: for $0\leq n\leq\infty$ $v_n(r)=v(r_n)$ and also $v_\infty=v$

5.2.7 Proposition

If $r \in$ Env, $z \in$ Id, $v \in$ [Env->D] and $0\leq n,m\leq\infty$ then:

5.2.7.1 $r_{n+1}(z)=r(z)_n$

5.2.7.2 $((v/z)r)_{n+1}=(v_n/z)r_{n+1}$

5.2.7.3 $(v/z)r=\bigsqcup_n (v_n/z)r_{n+1}$

5.2.7.4 $(t \text{ in } [\text{Env->D}])_n=(t \text{ in } [\text{Env->D}])$ (for $t \in S$)

5.2.7.5 $(r_n)_m=r_{\min\{n,m\}}$

Proof/

Proof

Trivial if $n=\infty$ or $m=\infty$ (if $\infty=\infty+1=\infty-1$) so suppose $0\leq n,m<\infty$:

5.2.7.1

$$
\begin{aligned}
r_{n+1}(z)(r') &= r(z)(r'_n) && \text{(by Env5)}\\
&= r(z)_n(r') && \text{(by definition 5.2.6)}
\end{aligned}
$$

so $r_{n+1}(z)=r(z)_n$.

5.2.7.2

$$
\begin{aligned}
((v/z)r)_{n+1}(z') &= \lambda r'.\ ((v/z)r)_{n+1}(z')(r') && \text{(by extensionality)}\\
&= \lambda r'.\ ((v/z)r)(z')(r'_n) && \text{(by Env5)}\\
&= \lambda r'.\ ((v/z)r)(z')_n(r') && \text{(by 5.2.6)}\\
&= ((v/z)r)(z')_n && \text{(by extensionality)}\\
&= \begin{cases} \perp_n & \text{if } z=\perp \text{ or } z'=\perp\\ v_n & \text{if } z=z'\neq\perp \qquad \text{(by 3.11.3)}\\ r(z')_n & \text{otherwise}\end{cases}\\
&= (v_n/z)r_{n+1}(z') && \text{(by 3.11.3 and 5.2.7.1)}
\end{aligned}
$$

5.2.7.3

$$
\begin{aligned}
\bigsqcup_n (v_n/z)r_{n+1} &= \bigsqcup_n\bigsqcup_m (v_m/z)r_n && \text{(as } (v_m/z)r_n \text{ is monotonic in } m,n)\\
&= (\bigsqcup_m v_m/z)\bigsqcup_n r_n\\
&= (\bigsqcup_m v_m/z)r && \text{(by Env3)}\\
&= (v/z)r && (\text{as } \bigsqcup_m v_m = \lambda r.\ (\bigsqcup_m v_m)r = \lambda r.\ \bigsqcup_m v(r_m)\\
& && = \lambda r.\ v(\bigsqcup_m r_m)\\
& && = \lambda r.\ v(r)\\
& && = v\ )
\end{aligned}
$$

<u>5.2.7.4</u>

$$
\begin{aligned}
(t \text{ in } [Env\text{->}D])_n(r) &= (t \text{ in } [Env\text{->}D])(r_n)\\
&= (\lambda r.\ t \text{ in } D)(r_n)\\
&= (t \text{ in } D)\\
&= (t \text{ in } [Env\text{->}D])(r)
\end{aligned}
$$

<u>5.2.7.5</u>

I do induction on n,m.

If n=0 or m=0 then $r_{nm}=\perp=r_0=r_{\min\{n,m\}}$

If n>0, m>0 and $r_{(n-1)(m-1)}=r_{\min\{(n-1),(m-1)\}}$ then

$$
\begin{aligned}
r_{nm} &= \lambda z.\ \lambda r'.\ r_{nm}(z)(r') && \text{(by extensionality)}\\
&= \lambda z.\ \lambda r'.\ r_n(z)(r'_{m-1}) && \text{(by Env5)}\\
&= \lambda z.\ \lambda r'.\ r(z)(r'_{(m-1)(n-1)}) && \text{(by Env5)}\\
&= \lambda z.\ \lambda r'.\ r(z)(r'_{\min\{(m-1),(n-1)\}}) && \text{(by induction)}\\
&= \lambda z.\ \lambda r'.\ r(z)(r'_{\min\{n,m\}-1}) && \text{(as } \min\{(m-1),(n-1)\}=\min\{n,m\}-1)\\
&= \lambda z.\ \lambda r'.\ r_{\min\{n,m\}}(z)(r') && \text{(by Env5)}\\
&= r_{\min\{n,m\}} && \text{(by extensionality)}
\end{aligned}
$$

Q.E.D.

5.3 <u>Extended LISP (ELISP) and extended ALIST (EALIST)</u>

I am now going to extend LISP and ALIST to bigger languages ELISP and EALIST, these bigger languages contain the smaller ones and on these/

these [...] and -> are the same as before. To avoid running out of symbols I am going to extend the metavariable conventions to range over expressions from ELISP and EALIST - in what follows the sets <form>, <function> and <alist> are bigger than they were previously. The new metavariable conventions are:

variable	range
A	<S-expression> (as in 3.7.15 above)
x,f,z	<identifier> (as in 3.7.15 above)
e	<form> (as in 5.3.1 below)
fn	<function> (as in 5.3.1 below)
F	<standard function> (i.e. {car,cdr,cons,atom,eq})
t	S (where S=flat(<S-expression>)
r	Env (as in 5.3.1 below)
v	[Env->D] (as in 5.3.1 below)
a	<alist> (as in 5.3.2 below)
p	<term> (as defined by p ::= A | (e,a))
w	<bound exp> (as in 5.3.2 below)

The expressions of ELISP and EALIST are like those of LISP and ALIST except that they can contain indices at various points. The purpose of these indices is to finitise programs by converting 'circles' or 'knots' which could lead to unending computations into 'spirals' which, as one traverses them, eventually 'wind down' to $\perp$. This remark is designed/

designed to demystify some of the definitions and results which follow, if it does not do this ignore it!

5.3.1 ELISP

Syntax

$$e ::= A \mid x \mid fn[e_1;\dots;e_n] \mid [e_{11}\rightarrow e_{12};\dots;e_{n1}\rightarrow e_{n2}]$$

$$fn ::= F \mid f \mid \lambda[[x_1;\dots;x_n];e] \mid \mathrm{Label}^m[f;fn] \mid \mu^m[f;fn] \quad (0\leq m\leq\infty)$$

Semantics

Denotations: $D=S+FUN$

$S=\mathrm{flat}(\langle S\text{-expression}\rangle)$

$FUN=[S^*\rightarrow S]$

Environments: Env an alist model (as in 5.2.1)

Semantic functions: $e\mapsto[\![e]\!]: \langle\text{form}\rangle\rightarrow[Env\rightarrow S]$

$fn\mapsto[\![fn]\!]: \langle\text{function}\rangle\rightarrow[Env\rightarrow FUN]$

Semantic equations:

(S1) $[\![A]\!](r)=A$

(S2) $[\![x]\!](r)=r(x)\mid S$

(S3) $[\![fn[e_1;\dots;e_n]]\!](r)=[\![fn]\!](r)([\![e_1]\!](r),\dots,[\![e_n]\!](r))$

(S4) $[\![[e_{11}\rightarrow e_{12};\dots;e_{n1}\rightarrow e_{n2}]]\!](r)=([\![e_{11}]\!](r)\rightarrow[\![e_{12}]\!](r),\dots,[\![e_{n1}]\!](r)\rightarrow[\![e_{n2}]\!](r))$

(S5)/

(S5) $$[\![car]\!](r)=\underline{\lambda}t.car(t)$$
$$[\![cdr]\!](r)=\underline{\lambda}t.cdr(t)$$
$$[\![cons]\!](r)=\underline{\lambda}t_1t_2.cons(t_1,t_2)$$
$$[\![atom]\!](r)=\underline{\lambda}t.atom(t)$$
$$[\![eq]\!](r)=\underline{\lambda}t_1t_2.eq(t_1,t_2)$$
(S6) $$[\![f]\!](r)=r(f)(r) \mid FUN$$
(S7) $$[\![\lambda[[x_1;\ldots;x_n];e]]\!](r)=\underline{\lambda}t_1\ldots t_n:S.[\![e]\!]((t_1/x_1)\ldots(t_n/x_n)r)$$
(S8) $$[\![Label^m[f;fn]]\!](r)=[\![fn]\!]_m(([\![fn]\!]/f)r) \quad \text{(see note 1 below)}$$
(S9) $$[\![\mu^m[f;fn]]\!](r)=\bigsqcup_{n=0}^{m}(\lambda v:[Env\to D].\lambda r':Env.[\![fn]\!]((v/f)r'))^n(\bot)(r)$$
(see note 2 below)

Notes

Note 1: If $m<\infty$ then $[\![fn]\!]_m=\lambda r.[\![fn]\!](r_m)$ (cf. definition 5.2.6) if $m=\infty$ I define $[\![fn]\!]_\infty=[\![fn]\!]$
Thus $Label^\infty[f;fn]$ denotes in ELISP the same as $Label[f;fn]$ does in LISP.

Note 2: Notice that the meaning of $\mu^\infty[f;fn]$ in ELISP is the same as the meaning of $\mu[f;fn]$ in LISP and that:

(S10) $$[\![\mu^m[f;fn]]\!](r)=\begin{cases}\bot & \text{if } m=0\\ [\![fn]\!](([\![\mu^{m-1}[f;fn]]\!]/f)r) & \text{if } m>0\end{cases}$$

In view of these notes an expression of ELISP in which all the indices are ∞ can be identified with an expression of LISP. Thus ELISP 'contains' LISP and I make the notational conventions that:

$Label^\infty$/

$$\mathrm{Label}^{\infty}[f;fn]=\mathrm{Label}[f;fn]$$
$$\mu^{\infty}[f;fn]=\mu[f;fn]$$

Notice that Note 1 above and conventions 3.11.4 imply that:

$$(\llbracket fn\rrbracket_m \text{ in } [Env\text{->}D])=(\llbracket fn\rrbracket \text{ in } [Env\text{->}D])_m$$

5.3.2 EALIST

Syntax

$$a ::= NIL \mid (w/z)a$$
$$w ::= A \mid fn_m \mid \perp \qquad (0\leq m\leq\infty)$$

Remark: I may omit writing NIL in alists e.g. $(A/x)(fn_m/f)$ means $(A/x)(fn_m/f)NIL$.

Semantics

Semantic functions: $a\mapsto\llbracket a\rrbracket$: <alist>->Env

$w\mapsto\llbracket w\rrbracket$: <bound exp>->[Env->D]

Semantic equations:

(AS1) $\llbracket NIL\rrbracket=\perp$

(AS2-4) $\llbracket(w/z)a\rrbracket=(\llbracket w\rrbracket/z)\llbracket a\rrbracket$

(AS2) $\llbracket A\rrbracket=(A \text{ in } [Env\text{->}D])$

(AS3) $\llbracket fn_m\rrbracket=(\llbracket fn\rrbracket_m \text{ in } [Env\text{->}D])$

(AS4) $\llbracket\perp\rrbracket=\perp_{[Env\text{->}D]}$

Remark/

Remark: The funny numbering of these equations is designed so that it is in harmony with that of the semantic equations in 4.2.

Each expression of ALIST corresponds to an expression of EALIST with infinite indices (though not vice versa), hence I shall use (fn/z) to mean (fn_{∞}/z) and thus regard ALIST as a subset of EALIST.

Notice that in view of definition 3.11.3 $\forall a. [\![a]\!](\bot)=\bot$

5.4 The extension of -> to ELISP and EALIST

In order to simplify extending -> to deal with terms (e,a) where e $\in$ <form> (of ELISP) and a $\in$ <alist> (of EALIST) I make some definitions.

5.4.1 Definition

If a $\in$ <alist>, z $\in$ <identifier> define a(z) $\in$ <bound exp> by:

$$NIL(z)=\bot$$
$$((w/z')a)(z)= \text{if } z=z' \text{ then } w \text{ else } a(z)$$

This definition makes sense in view of the following lemma.

5.4.2 Lemma

$$[\![a]\!](z)=[\![a(z)]\!]$$

Proof

Structural/

Structural inductinn on a:

$[\![\mathrm{NIL}]\!](z)=\perp=[\![\mathrm{NIL}(z)]\!]$. Assume result for a then:

$[\![(w/z')a]\!](z)=(([\![w]\!]/z')[\![a]\!])(z)$ and $z'\neq\perp$ by the definition of <alist>

$=$if $z=\perp$ then $\perp$ elseif $z=z'$ then $[\![w]\!]$ else $[\![a]\!](z)$

(by definition 3.11.3)

$=$if $z=z'$ then $[\![w]\!]$ else $[\![a]\!](z)$ (since $\forall$ a.$[\![a]\!](\perp)=\perp$ and $z'\neq\perp$

$=$if $z=z'$ then $[\![w]\!]$ else $[\![a(z)]\!]$ (by induction)

$=[\![$if $z=z'$ then w else $a(z)]\!]$

$=[\![((w/z')a)(z)]\!]$

Q.E.D.

The following definition is designed to make lemma 5.4.4 (below) true.

5.4.3 Definition

For $w \in$ <bound exp>, $0\leq n\leq\infty$ define $w_n \in$ <bound exp> by:

$0\leq n<\infty$: $A_n=A$

$(fn_m)_n=fn_{\min\{m,n\}}$

$\perp_n=\perp$

$n=\infty$: $w_\infty=w$

5.4.4 Lemma

$[\![w_n]\!]=[\![w]\!]_n$

Proof/

Proof

$0 \leq n < \infty$: $[\![A_n]\!] = [\![A]\!] = (A \text{ in } [Env \to D])$ (by AS2 of 5.3.2)

$= \lambda r.(A \text{ in } D)$

$= (\lambda r.(A \text{ in } D))_n$ (by definition 5.2.6)

$= [\![A]\!]_n$

$[\![(fn_m)_n]\!] = [\![fn_{\min\{n,m\}}]\!]$

$= ([\![fn]\!]_{\min\{n,m\}} \text{ in } [Env \to D])$ (by AS3 of 5.3.2)

$= ([\![fn]\!] \text{ in } [Env \to D])_{\min\{n,m\}}$

$= ([\![fn]\!] \text{ in } [Env \to D])_{mn}$ (by definition 5.2.6 and 5.2.7.5)

$= ([\![fn]\!]_m \text{ in } [Env \to D])_n$

$= [\![fn_m]\!]_n$

$[\![\perp_n]\!] = [\![\perp]\!] = \perp = \perp_n = [\![\perp]\!]_n$

$n = \infty$: $[\![w_\infty]\!] = [\![w]\!] = [\![w]\!]_\infty$

Q.E.D.

The next definition is designed to make lemma 5.4.6 (below) true.

5.4.5 Definition

For $a \in \langle alist \rangle$, $0 \leq n \leq \infty$ define $a_n \in \langle alist \rangle$ by:

$n=0$: $NIL_0 = NIL$

$((w/z)a)_0 = (\perp/z)a_0$

$0 < n \leq \infty$: $NIL_n = NIL$

$((w/z)a)_n = (w_{n-1}/z)a_n$

5.4.6/

5.4.6 Lemma

$$[\![a_n]\!]=[\![a]\!]_n$$

Proof

n=0: $[\![a_0]\!]=\perp=[\![a]\!]_0$ (by definition 5.4.5 and a quick structural induction on a)

$0<n\leq\infty$: I do structural induction on a.

$$\begin{aligned}
[\![\mathrm{NIL}_n]\!]=[\![\mathrm{NIL}]\!]=\perp=\perp_n=[\![\mathrm{NIL}]\!]_n & \quad \text{(by definition 5.4.5)}\\
[\![((w/z)a)_n]\!]=[\![(w_{n-1}/z)a_n]\!] & \quad \text{(by definition 5.4.5)}\\
=([\![w_{n-1}]\!]/z)[\![a_n]\!] & \quad \text{(by AS2-4 of 5.3.2)}\\
=([\![w]\!]_{n-1}/z)[\![a]\!]_n & \quad \text{(by 5.4.4 and induction)}\\
=(([\![w]\!]/z)[\![a]\!])_n & \quad \text{(by 5.2.7.2)}\\
=[\![(w/z)a]\!]_n & \quad \text{(by AS2-4 of 5.3.2)}
\end{aligned}$$

Q.E.D.

Here now is the extended definition of ->.

5.4.7 Definition

Define ->, $\overset{*}{>}$ by:

P1/

P1 $(A,a)\rightarrow A$

P2 $$\frac{a(x)=A}{(x,a)\rightarrow A}$$

P3 $$\frac{F(A_1,\ldots,A_n)=A}{(F[A_1;\ldots;A_n],a)\rightarrow A}$$

P4 $$\frac{\forall i.(e_i,a)\overset{*}{\rightarrow}A_i \text{ and } \exists m.e_m\neq A}{(fn[e_1;\ldots;e_n],a)\rightarrow(fn[A_1;\ldots;A_n],a)}$$

P5 $$\frac{(e_{m1},a)\overset{*}{\rightarrow}T \text{ and } \forall i<m.(e_{i1},a)\overset{*}{\rightarrow}F}{([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}],a)\rightarrow(e_{m2},a)}$$

P6 $$\frac{a(f)=fn_m}{(f[A_1;\ldots;A_n],a)\rightarrow(fn[A_1;\ldots;A_n],a_m)}$$

P7 $$\frac{m\leq n}{(\lambda[[x_1;\ldots;x_m];e][A_1;\ldots;A_n],a)\rightarrow(e,(A_1/x_1)\ldots(A_m/x_m)a)}$$

P8 $(\text{Label}^m[f;fn][A_1;\ldots;A_n],a)\rightarrow(fn[A_1;\ldots;A_n],((fn/f)a)_m)$

P9 $(\mu^{m+1}[f;fn][A_1;\ldots;A_n],a)\rightarrow(fn[A_1;\ldots;A_n],(\mu^m[f;fn]/f)a)$ $(\infty=\omega+1!)$

P10 $$\frac{þ_1\rightarrow þ_2\rightarrow\ldots\rightarrow þ_n \quad (1\leq n)}{þ_1\overset{*}{\rightarrow}þ_n}$$

I hope it is clear that definition 5.4.7 reduces to definition 4.2.2 if all the expressions are from LISP and ALIST (i.e. have infinite indices). LISP-induction extends to the new -> painlessly. Define <* as in definition 4.4.1, but using -> as defined in 5.4.7. If $\underset{n}{\rightarrow}$, $\underset{n}{\overset{*}{\rightarrow}}$ are defined as in definition 4.4.2 but using the new extended P1,P2,P3,P6,P7,P8, of 5.4.7 (instead of those of 4.2.2) then the proof/

proof of 4.4.3 goes through. Thus LISP-induction is valid, I shall now use it to prove the following lemma which generalises 4.5.1 and says that -> is a sound conversion rule. First let the range, <term>, of the metavariable p be given by:

$$p ::= A \mid (e,a)$$

and for $p \in$ <term> define $[\![p]\!] \in S$ by:

$$[\![a]\!]=A$$

$$[\![(e,a)]\!]=[\![e]\!]([\![a]\!])$$

5.4.8 <u>Lemma</u>

$$p \overset{*}{\to} A \Rightarrow [\![p]\!]=A$$

<u>Proof</u>

Same as proof of lemma 4.5.1 except replace 6, 8 and 9 of that proof by 6', 8' and 9' below:

6'. $p=(f[A_1;\ldots;A_n],a)$ and $a(f)=fn_m$

Then by LIH $R((fn[A_1;\ldots;A_n],a_m),A)$ i.e. $[\![fn]\!]([\![a_m]\!])(A_1,\ldots,A_n)=A$

so/

so $\llbracket p\rrbracket=\llbracket f\rrbracket(\llbracket a\rrbracket)(A_1,\dots,A_n)$

$=\llbracket a\rrbracket(f)(\llbracket a\rrbracket) \mid \mathrm{FUN}(A_1,\dots,A_n)$

$=\llbracket a(f)\rrbracket(\llbracket a\rrbracket) \mid \mathrm{FUN}(A_1,\dots,A_n)$ (by lemma 5.4.2)

$=(\llbracket fn\rrbracket_m \text{ in } [\mathrm{Env}\text{->}D])(\llbracket a\rrbracket) \mid \mathrm{FUN}(A_1,\dots,A_n)$ (by AS3 of 5.3.2)

$=\llbracket fn\rrbracket(\llbracket a_m\rrbracket)(A_1,\dots,A_n)$ (by 3.11.4 and 5.4.6)

$=A$

8'. $p=(\mathrm{Label}^m[f;fn][A_1;\dots;A_n],a)$

Then by LIH $R((fn[A_1;\dots;A_n],((fn/f)a)_m),A)$

i.e. $\llbracket fn\rrbracket(\llbracket((fn/f)a)_m\rrbracket)(A_1,\dots,A_n)=A$

so $\llbracket p\rrbracket=\llbracket \mathrm{Label}^m[f;fn]\rrbracket(\llbracket a\rrbracket)(A_1,\dots,A_n)$

$=\llbracket fn\rrbracket_m((\llbracket fn\rrbracket/f)\llbracket a\rrbracket)(A_1,\dots,A_n)$ (by S8 of 5.3.1)

$=\llbracket fn\rrbracket(\llbracket(fn/f)a\rrbracket_m)(A_1,\dots,A_n)$ (by Note 1 of 5.3.1)

$=\llbracket fn\rrbracket(\llbracket((fn/f)a)_m\rrbracket)(A_1,\dots,A_n)$ (by lemma 5.4.6)

$=A$

9'. $p=(\mu^{m+1}[f;fn][A_1;\dots;A_n],a)$

Then by LIH $R((fn[A_1;\dots;A_n],(\mu^m[f;fn]/f)a),A)$

i.e. $\llbracket fn\rrbracket(\llbracket(\mu^m[f;fn]/f)a\rrbracket)(A_1,\dots,A_n)=A$

so $\llbracket p\rrbracket=\llbracket \mu^{m+1}[f;fn]\rrbracket(\llbracket a\rrbracket)(A_1,\dots,A_n)$

$=\llbracket fn\rrbracket((\llbracket \mu^m[f;fn]\rrbracket/f)\llbracket a\rrbracket)(A_1,\dots,A_n)$ (by S10 of Note 2 of 5.3.1)

$=\llbracket fn\rrbracket(\llbracket(\mu^m[f;fn]/f)a\rrbracket)(A_1,\dots,A_n)$ (by AS2-4 of 5.3.2)

$=A$

Q.E.D.

Examination/

Examination of the proof of this lemma shows that in fact it essentially proves the stronger result:

5.4.9 $p \to p' \Rightarrow [\![p]\!]=[\![p']\!]$

I need to use this fact in proving lemma 5.9.2 below. I will not prove 5.4.9 here as it is utterly straightforward to verify it by direct calculation (one just considers the various cases of p; 5.4.8 is needed when p is an application or conditional) - the details occur in the proof of 5.4.8. It would be possible to generalize LISP-induction so that the proof of 5.4.8 would (more or less) constitute a proof of 5.4.9. I have not done this because I cannot see how to do it so that the generalized LISP-induction is not messy and ad hoc looking. I am not happy about this situation - I feel that the essence of "induction on the size of computation" has not been adequately captured in LISP-induction, but I cannot put my finger on exactly what is needed. There is something lurking around here which I need to understand and it has escaped capture so far.

5.5 <u>Approximants and the relation $<$</u>

For e', e ∈ <form>, a', a ∈ <alist> I am going to define what it means for e'<e and a'<a. Roughly e'<e, a'<a means that e', a' are 'finite' approximants to e, a respectively. Here is the precise definition:

5.5.1/

5.5.1 Definition

Define e'<e, fn'<fn, a'<a and p'<p by:

5.5.1.1 on <form>

5.5.1.1.1 A<A

5.5.1.1.2 x<x

5.5.1.1.3 $$\frac{fn'<fn, e'_1<e_1, \ldots, e'_n<e_n}{fn'[e'_1;\ldots;e'_n]<fn[e_1;\ldots;e_n]}$$

5.5.1.1.4 $$\frac{e'_{11}<e_{11}, e'_{12}<e_{12}, \ldots, e'_{n1}<e_{n1}, e'_{n2}<e_{n2}}{[e'_{11}\rightarrow e'_{12};\ldots;e'_{n1}\rightarrow e'_{n2}]<[e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}]}$$

5.5.1.2 on <function>

5.5.1.2.1 F<F

5.5.1.2.2 f<f

5.5.1.2.3 $$\frac{e'<e}{\lambda[[x_1;\ldots;x_m];e']<\lambda[[x_1;\ldots;x_m];e]}$$

5.5.1.2.4 $$\frac{m<\omega, fn'<fn}{Label^m[f;fn']<Label[f;fn]}$$

5.5.1.2.5 $$\frac{m<\omega, fn'<fn}{\mu^m[f;fn']<\mu[f;fn]}$$

5.5.1.3/

5.5.1.3 on <alist>

5.5.1.3.1 NIL<NIL

5.5.1.3.2 $\dfrac{a'<a}{(A/z)a'<(A/z)a}$

5.5.1.3.3 $\dfrac{a'<a}{(\perp/z)a'<(A/z)a}$

5.5.1.3.4 $\dfrac{a'<a, fn'<fn, m<\infty}{(fn'_m/z)a'<(fn/z)a}$

5.5.1.3.5 $\dfrac{a'<a, fn \text{ is a LISP function (i.e. all indices in fn are } \infty)}{(\perp/z)a'<(fn/z)a}$

5.5.1.3.6 $\dfrac{a'<a, fn'<fn, m<\infty}{(\mu^m[f;fn']/f)a'<(\mu[f;fn]/f)a}$

5.5.1.4 on <term>

5.5.1.4.1 A<A

5.5.1.4.2 $\dfrac{e'<e, a'<a}{(e',a')<(e,a)}$

Remarks: (1) Note that the four f's in 5.5.1.3.6 are all the same.

(2) From 5.5.1.1 and 5.5.1.2 it is clear that e'<e,fn'<fn <=>.

(2.1) All indices in e,fn are infinite and all indices in e',fn' are finite.

(2.2) e',fn' can be got from e,fn by replacing all infinite indices in e,fn by finite ones.

The/

The next lemma shows that the denotation of an expression is the limit of the denotations of its 'finite' approximants.

5.5.2 Lemma

5.5.2.1 $e = \bigsqcup_{e' < e} [\![e']\!]$

5.5.2.2 $fn = \bigsqcup_{fn' < fn} [\![fn']\!]$

5.5.2.3 $a = \bigsqcup_{a' < a} [\![a']\!]$

5.5.2.4 $p = \bigsqcup_{p' < p} [\![p']\!]$

Proof

Since $[\![fn]\!] = \bigsqcup_{m=0} [\![fn]\!]_m$,

$[\![Label[f;fn]]\!] = \bigsqcup_{m=0} [\![Label^m[f;fn]]\!]$ and

$[\![\mu[f;fn]]\!] = \bigsqcup_{m=0} [\![\mu^m[f;fn]]\!]$

The result follows by structural induction, the continuity of the (denotations of the) expression building operations and the directedness of $\{e' \mid e' < e\}$, $\{fn' \mid fn' < fn\}$, $\{a' \mid a' < a\}$, $\{p' \mid p' < p\}$ (all these being straightforward to check).

Q.E.D.

5.5.3 Lemma

$a' < a, 0 \leq m \leq \infty \Rightarrow a'_m < a$

Proof

If/

If m=0 then result follows from 5.5.1.3.1, 5.5.1.3.3 and 5.5.1.3.5. If $0<m\leq\infty$ then a'_m can be got from a' by reducing some (perhaps none) of the subscripts (not superscripts) hence by 5.5.1.3.4 the result follows.

Q.E.D.

The next lemma shows that if a 'finite' approximant converts to an S-expression then so does the term it approximates.

5.6 Lemma

$p' \overset{*}{>} A, p' < p \Rightarrow p \overset{*}{>} A$

Proof

Let $R(p',A) \Leftrightarrow \forall p.p'<p \Rightarrow p\overset{*}{>}A$

Then I need to show that:

$p'\overset{*}{>}A \Rightarrow R(p',A)$

I prove this by LISP-induction.

BASE:

$R(A,A) \Leftrightarrow (\forall p.A<p \Rightarrow p\overset{*}{>}A) \Leftrightarrow A\overset{*}{>}A$ which is true.

INDUCTION:

Assume as LIH: $(\forall p''<(e',a').\ p''\overset{*}{>}A'' \Rightarrow R(p'',A''))$. Let $p'=(e',a')\overset{*}{>}A$ then to show $R(p',A)$ I need to show that $(p'<p \Rightarrow p\overset{*}{>}A)$; so/

so assume $p'<p$ and also $e'<e$, $fn'<fn$, $a'<a$ etc. – I verify that $p\ast{>}A$ by cases on p':

1. $p'=(A,a')$

   Then $p=(A,a)$ so $p\ast{>}A$.

2. $p'=(x,a')$ and $a'(x)=A$

   Then $p=(x,a)$ and by 5.5.1.3.2 $a(x)=A$ so $p\ast{>}A$.

3. $p'=(F[A_1;\ldots;A_n],a')$ and $F(A_1,\ldots,A_n)=A$

   Then $p=(F[A_1;\ldots;A_n],a)$ so $p\ast{>}A$.

4. $p'=(fn'[e_1';\ldots;e_n'],a')$ and $\forall i.(e_i',a)\to A_i$

   Then $p=(fn[e_1;\ldots;e_n],a)$,

   by LIH $\forall i.R((e_i',a'),A_i)$ and $R((fn'[A_1;\ldots;A_n],a'),A)$

   so $\forall i.(e_i,a)\ast{>}A_i$ and $(fn[A_1;\ldots;A_n],a)\ast{>}A$

   hence $p\to(fn[A_1;\ldots;A_n],a)\ast{>}A$.

5. $p'=([e_{11}'\to e_{12}';\ldots;e_{n1}'\to e_{n2}'],a')$, $\forall i<m.(e_{i1}',a')\ast{>}F$ and $(e_{m1}',a')\ast{>}T$

   Then $p=([e_{11}\to e_{12};\ldots;e_{n1}\to e_{n2}],a)$,

   by LIH $\forall i<m.R((e_{i1}',a'),F), R((e_{m1}',a'),T)$ and $R((e_{m2}',a'),A)$

   so $\forall i<m.(e_{i1},a)\ast{>}F, (e_{m1},a)\ast{>}T$ and $(e_{m2},a)\ast{>}A$

   hence $p\to(e_{m2},a)\ast{>}A$.

6. $p'=(f[A_1;\ldots;A_n],a')$

   There are two cases to consider.

6.1 $a'(f)=fn_m'$ and $m<\infty$

   Then $p=(f[A_1;\ldots;A_n],a')$ and by 5.5.1.3.4

   $a(f)=fn$. By LIH $R((fn'[A_1;\ldots;A_n],a_m'),A)$

   hence by lemma 5.5.3

p/

p->(fn[A_1;...;A_n],a)*>A

6.2 a'(f)= μ^m[f;fn']= μ^m[f;fn']$_\infty$ and $m<\infty$

Then p=(f[A_1;...;A_n],a') and by 5.5.1.3.6

a(f)= μ[f;fn]. By LIH R((μ^m[f;fn'][A_1;...;A_n],a'),A)

hence p->(μ[f;fn][A_1;...;A_n],a)*>A.

7. p'=(λ[[x_1;...;x_m];e'][A_1;...;A_n],a') and $m\leq n$

Then p=(λ[[x_1;...;x_m];e][A_1;...;A_n],a),

by LIH R((e',(A_1/x_1)...(A_m/x_m)a'),A) so by 5.5.1.3.2

p->(e,(A_1/x_1)...(A_m/x_m)a)*>A.

8. p'=(Label^m[f;fn'][A_1;...;A_n],a') and $m<\infty$

Then p=(Label[f;fn][A_1;...;A_n],a)

by LIH R((fn'[A_1;...;A_n],((fn'/f)a')$_m$),A)

so by lemma 5.5.3, 5.5.1.3.4 (if m>0) and 5.5.1.3.5 (if m=0)

p->(fn[A_1;...;A_n],(fn/f)a)*>A.

9. p'=(μ^{m+1}[f;fn'][A_1;...;A_n],a') and $m<\infty$

Then p=(μ[f;fn][A_1;...;A_n],a)

by LIH R((fn'[A_1;...;A_n],(μ^m[f;fn]/f)a'),A) so by 5.5.1.3.6

p->(fn[A_1;...;A_n],(μ[f;fn]/f)a)*>A.

Q.E.D.

5.7 Some terminology

The definitions below confer rigour upon certain phrases which, up to now/

now, I have used in an informal and intuitive sense.

5.7.1 Definition

If $e' < e, fn' < fn, a' < a, p' < p$ then

(1) e', fn', a', p' are finite

(2) e, fn, a, p are in LISP

(3) e', fn', a', p' are approximants of e, fn, a, p respectively.

5.7.2 Definition

p terminates $\Leftrightarrow$ there does not exist an infinite sequence $p_1, p_2, \ldots$ such that $p = p_1 \to p_2 \to p_3 \to \ldots$.

5.7.3 Definition

$p]$ (read "p is stuck") $\Leftrightarrow$ $p \neq A$ and for no p' does $p \to p'$.

$p \overset{*}{\to} p']$ $\Leftrightarrow$ $p \overset{*}{\to} p'$ and $p']$.

5.7.4 Lemma

p terminates $\Leftrightarrow$ $p \overset{*}{\to} A$ or $p \overset{*}{\to} p']$ for some A or p'.

Proof Trivial Q.E.D.

5.8 Proof that all finite terms terminate

The goal of this section is to show that if p is finite then p terminates. To do this I shall define the rank, $rk(p) \in N$, of p - I had initially hoped to arrange things so that $p \to p' \Rightarrow rk(p) > rk(p')$, unfortunately/

unfortunately this is not quite true, but something similar which is good enough, is. I could have altered the definition of -> to make p->p' => rk(p)>rk(p') however doing that would stop the new -> from being a nice simple extension of the old one. I feel the course I have taken is the lesser of the two evils. The definition of rk given below is the first one I came across which had the desired properties; there is no uniform intuition which will render each clause in its definition meaningful (although thinking of rk(p) as being a bound on the number of conversions which can be done on p might help). It should be intuitively plausible that finite terms terminate though.

5.8.1 <u>Definition</u>

For e $\in$ <form>, fn $\in$ <function>, w $\in$ <bound exp>, z $\in$ <identifier>, a $\in$ <alist> and p $\in$ <term> define rk(e),rk(fn),rk(w,z),rk(a),rk(p) $\in N \cup \{\infty\}$ by structural induction as follows:

<u>5.8.1.1 rk(e)</u>

5.8.1.1.1 $rk(A)=0$

5.8.1.1.2 $rk(x)=1$

5.8.1.1.3 $rk(fn[e_1;\ldots;e_n])=1+rk(fn)+rk(e_1)+\ldots+rk(e_n)$

5.8.1.1.4 $rk([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}])=1+rk(e_{11})+rk(e_{12})+\ldots+rk(e_{n1})+rk(e_{n2})$

<u>5.8.1.2 rk(fn)/</u>

<u>5.8.1.2 rk(fn)</u>

5.8.1.2.1 $rk(F)=0$

5.8.1.2.2 $rk(f)=1$

5.8.1.2.3 $rk(\lambda[[x_1;\ldots;x_n];e])=1+rk(e)$

5.8.1.2.4 $rk(Label^m[f;fn])=(m+1)(rk(fn)+1)$

5.8.1.2.5 $rk(\mu^m[f;fn])=(m+1)(rk(fn)+1)$

<u>5.8.1.3 rk(w,z)</u>

5.8.1.3.1 $rk(A,z)=0$

5.8.1.3.2 $rk(fn_m,z)=\begin{cases} rk(fn) & \text{if } fn \text{ is of the form } \mu^n[z;fn'] \\ (m+1)(rk(fn)+1) & \text{otherwise} \end{cases}$

5.8.1.3.3 $rk(\perp,z)=0$

<u>5.8.1.4 rk(a)</u> $rk(a)=\sum_z rk(a(z),z)$

<u>5.8.1.5 rk(p)</u>

5.8.1.5.1 $rk(A)=0$

5.8.1.5.2 $rk((e,a))=1+rk(e)+rk(a)$

Remark: A binding of the form:

$$(\mu^m[\underbrace{f}_{\text{same}};fn]/\underbrace{f}_{})$$

arises from conversion on a finite term of the form

(/

$(\mu^{m+1}[f;fn][A_1;\ldots;A_n],a)$ and I want it only to make a finite contribution to the rank of any alist in which it occurs. Bindings of the form

$$(\mu^{n}[\underset{\uparrow}{f};fn]/\underset{\uparrow}{g})$$
different

do not arise in this manner and do not need to be treated so subtly. This is the reason for 5.8.1.3.2 and why I do not define rk(w) and then set $rk(a)=\sum_{z} rk(a(z))$.

The following lemma collects together some trivial (but messy!) consequences of the definition of rk. These are needed to prove that finite terms terminate, I suggest that you skip the lemma initially and only refer to it when it is invoked later on.

5.8.2 <u>Lemma</u>

Suppose fn ∈ <function>, a ∈ <alist> and m ∈ N are finite then:

5.8.2.1 p finite => rk(p) finite

5.8.2.2 $\forall z.\ 0\leq n\leq\infty.\ rk(a_n(z),z) \leq rk(a(z),z)$ (hence $rk(a_n)\leq rk(a)$)

5.8.2.3 $rk(a)+rk(a') \geq rk(a.a')$

5.8.2.4 $(m+1)(rk(fn)+1)+rk(a) \geq rk((fn_m/f)a)$

5.8.2.5 $a(f)=fn_m$ where fn <u>not</u> of the form $\mu^n[f;fn]$

$\Rightarrow rk(f)+rk(a) > rk(fn)+rk(a_m)$

5.8.2.6/

5.8.2.6 $a(f)=\mu^{s}[f:fn]_{n}, 0<s<\infty, 0\leq n\leq\infty$,

$\Rightarrow rk(a) > rk(fn)+rk((\mu^{s-1}[f;fn]/f)a_{n})$

5.8.2.7 $rk(Label^{m}[f;fn])+rk(a) > rk(fn)+rk(((fn/f)a)_{m})$

5.8.2.8 $rk(\mu^{m+1}[f;fn])+rk(a) > rk(fn)+rk((\mu^{m}[f;fn]/f)a)$

Proof

5.8.2.1

If all indices in p are finite then clearly so is rk(p).

If p is finite (as defined in 5.7.1) then the only way p could contain infinite indices would be if p=(e,a) and for some z. $a(z)=\mu^{n}[z;fn]_{\infty}$ but then by 5.8.1.3.2 this infinite index gets ignored by rk, hence rk(p) is finite.

5.8.2.2

If n=0 then for all z. $a_{0}(z)=\perp$ so $rk(a_{0}(z),z)=0\leq rk(a(z),z)$.

If n>0 then if $a(z)=A$ or $a(z)=\perp$ then $a_{n}(z)=a(z)$ so $rk(a_{n}(z),z)=rk(a(z),z)$ and, if $a(z)=fn_{m}$ then $a_{n}(z)=fn_{\min\{n-1,m\}}$ so $rk(a_{n}(z),z)\leq rk(a(z),z)$.

5.8.2.3

$$\begin{aligned} rk(a)+rk(a') &= \sum_{z} rk(a(z),z)+\sum_{z'} rk(a'(z'),z') && \text{(by 5.8.1.4)}\\ &\geq \sum_{z} rk((a.a')(z),z) && \text{(by 4.8.11)}\\ &= rk(a.a') \end{aligned}$$

5.8.2.4/

5.8.2.4

$$\begin{aligned}
rk((fn_m/f)a) &= rk(fn_m,f)+\sum_{z\neq f} rk(a(z),z) && \text{(by 5.8.14)}\\
&\leq (m+1)(rk(fn)+1)+\sum_{z} rk(a(z),z) && \text{(by 5.8.1.3.2)}\\
&= (m+1)(rk(fn)+1)+rk(a) && \text{(by 5.8.1.4)}
\end{aligned}$$

5.8.2.5

If $a(f)=fn_m$ and a,fn and m are finite and fn is not of the form $\mu^m[f;fn']$

$$\begin{aligned}
\text{then } rk(f)+rk(a) &= 1+\sum_{z} rk(a(z),z) && \text{(by 5.8.1.2.2 and 5.8.1.4)}\\
&= 1+rk(fn_m,f)+\sum_{z\neq f} rk(a(z),z)\\
&= 1+(m+1)(rk(fn)+1)+\sum_{z\neq f} rk(a(z),z) && \text{(by 5.8.1.3.2)}\\
&> rk(fn)+m(rk(fn)+1)+\sum_{z\neq f} rk(a(z),z)\\
&\geq rk(fn)+m(rk(fn)+1)+\sum_{z\neq f} rk(a_m(z),z) && \text{(by 5.8.2.2)}\\
&= rk(fn)+rk(a_m)
\end{aligned}$$

(consider m=0 and m>0 cases separately)

5.8.2.6

If $a(f)=\mu^s[f;fn]_n$ and s,fn,a are finite and s>0 then:

$$\begin{aligned}
rk(a) &= rk(\mu^s[f;fn]_n,f)+\sum_{z\neq f} rk(a(z),z) && \text{(by 5.8.1.4)}\\
&= rk(\mu^s[f;fn])+\sum_{z\neq f} rk(a(z),z) && \text{(by 5.8.1.3.2)}\\
&= (s+1)(rk(fn)+1)+\sum_{z\neq f} rk(a(z),z) && \text{(by 5.8.1.2.5)}\\
&> rk(fn)+s(rk(fn)+1)+\sum_{z\neq f} rk(a_n(z),z) && \text{(by 5.8.2.2)}\\
&= rk(fn)+rk((\mu^{s-1}[f;fn]/f)a_n)
\end{aligned}$$

5.8.2.7/

5.8.2.7

If m,fn,a are finite then:

$$\begin{aligned}
&\mathrm{rk}(\mathrm{Label}^{m}[f;fn])+\mathrm{rk}(a)\\
&\quad=(m+1)(\mathrm{rk}(fn)+1)+\mathrm{rk}(a) && \text{(by 5.8.1.2.4)}\\
&\quad>\mathrm{rk}(fn)+m(\mathrm{rk}(fn)+1)+\mathrm{rk}(a)\\
&\quad\geq\begin{cases}\mathrm{rk}(fn)+\mathrm{rk}((fn_{m-1}/f)a) & \text{if } m>0 \quad \text{(by 5.8.2.4)}\\ \mathrm{rk}(fn)+\mathrm{rk}(a) & \text{if } m=0\end{cases}\\
&\quad\geq\mathrm{rk}(fn)+\mathrm{rk}(((fn/f)a)_{m}) && \text{(by 5.8.2.2 and definition 5.4.5)}
\end{aligned}$$

5.8.2.8

If m,fn,a are finite then:

$$\begin{aligned}
&\mathrm{rk}(\mu^{m+1}[f;fn])+\mathrm{rk}(a)\\
&\quad=(m+2)(\mathrm{rk}(fn)+1)+\mathrm{rk}(a) && \text{(by 5.8.1.2.5)}\\
&\quad=\mathrm{rk}(fn)+(m+1)(\mathrm{rk}(fn)+1)+\mathrm{rk}(a)\\
&\quad\geq\mathrm{rk}(fn)+\mathrm{rk}((\mu^{m}[f;fn]/f)a) && \text{(by 5.8.1.3.2)}
\end{aligned}$$

Q.E.D.

An easy corollary of the next lemma is that finite terms terminate.

5.8.3 Lemma

If p is finite then:

$$p\to p' \Rightarrow \mathrm{rk}(p)>\mathrm{rk}(p')$$

except/

<u>except</u> when p is of the form: $(f[A_1;\ldots;A_n],a)$ where $a(f)=\mu^s[f;fn]_m$, and then

$$p\rightarrow p'\rightarrow p'' \Rightarrow rk(p)>rk(p'').$$

<u>Proof</u>

I consider the various possible cases of p. Assume $p\rightarrow p'$ then p must be of the form (e,a).

<u>case1</u>: $p=(A,a)$

Then $p'=A$ and $rk(p)=1+rk(a)>0=rk(p')$

<u>case2</u>: $p=(x,a)$ and $a(x)=A$

Then $p'=A$ and $rk(p)=1+1+rk(a)>0=rk(p')$

<u>case3</u>: $p=(F[A_1;\ldots;A_n],a)$ and $F(A_1,\ldots,A_n)=A$

Then $p'=A$ and $rk(p)=1+1+rk(a)>0=rk(A)$

<u>case4</u>: $p=(fn[e_1;\ldots;e_n],a)$, $\forall i.(e_i,a')\overset{*}{\rightarrow}A_i$ and $e_m\neq A_m$

Then $p'=(fn[A_1;\ldots;A_n],a)$

and $rk(p)=1+1+rk(fn)+rk(e_1)+\ldots+rk(e_n)+rk(a)$

$>1+1+rk(fn)+0+\ldots+0+rk(a)$ (as $rk(e_m)>0$)

$=1+rk(fn[A_1;\ldots;A_n])+rk(a)$

$=rk(p')$

<u>case5</u>:/

<u>case5</u>: $p=([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}],a),(e_{m1},a)\stackrel{*}{=}>T$ and $\forall\, i<m.(e_{i1},a)\stackrel{*}{=}>F$

Then $p'=(e_{m2},a)$ and $rk(p)=1+1+rk(e_{11})+\ldots+rk(e_{n2})+rk(a)$

$$>1+rk(e_{m2})+rk(a)$$

$$=rk(p')$$

<u>case6.1</u>: $p=(f[A_1;\ldots;A_n],a),a(f)=fn_m,m<\infty$ and fn <u>not</u> of form $\mu^s[f;fn']$

Then $p'=(fn[A_1;\ldots;A_n],a_m)$

and $rk(p)=1+1+rk(f)+rk(a)$

$>1+1+rk(fn)+rk(a_m)$ (by 5.8.2.5)

$=rk(p')$

<u>case6.2</u>: $p=(f[A_1;\ldots;A_n],a),a(f)=\mu^s[f;fn]_m$ and $p\to>p'\to>p''$

Then $p'=(\mu^s[f;fn][A_1;\ldots;A_n],a_m)$

so $p''=(fn[A_1;\ldots;A_n],(\mu^{s-1}[f;fn]/f)a_m)$ (if $p\to>p'\to>p''$ then $s>0$)

hence $rk(p)=1+1+rk(f)+rk(a)$

$>1+1+rk(f)+rk(fn)+rk((\mu^{s-1}[f;fn]/f)a_m)$ (by 5.8.2.6 - s is finite as p is)

$>1+1+rk(fn)+rk((\mu^{s-1}[f;fn]/f)a_m)$ (as $rk(f)=1$)

$=rk(p'')$

<u>case7</u>:/

<u>case7</u>: $p=(\lambda[[x_1;\ldots;x_m];e][A_1;\ldots;A_n],a)$ and $m\leq n$

Then $p'=(e,A_1/x_1)\ldots(A_m/x_m)a)$

and $rk(p)=1+1+1+rk(e)+rk(a)$ (by 5.8.1.1.3 and 5.8.1.2.3)

$>1+rk(e)+rk(a)$

$\geq 1+rk(e)+rk((A_1/x_1)\ldots(A_m/x_m)a)$ (by 5.8.1.4)

$=rk(p')$

<u>case8</u>: $p=(\mathrm{Label}^m[f;fn][A_1;\ldots;A_n],a)$ and $m<\infty$

Then $p'=(fn[A_1;\ldots;A_n],((fn/f),a)_m)$

so $rk(p)=1+1+rk(\mathrm{Label}^m[f;fn])+rk(a)$

$>1+1+rk(fn)+rk(((fn/f)a)_m)$ (by 5.8.2.7)

$=rk(p')$

<u>case9</u>: $p=(\mu^{m+1}[f;fn][A_1;\ldots;A_n],a)$ and $m<\infty$

Then $p'=(fn[A_1;\ldots;A_n],(\mu^m[f;fn]/f)a)$

so $rk(p)=1+1+rk(\mu^{m+1}[f;fn])+rk(a)$

$>1+1+rk(fn)+rk((\mu^m[f;fn]/f)a)$ (by 5.8.2.8)

$=rk(p')$

Q.E.D.

5.8.4 <u>Lemma</u>

If p is finite then p terminates.

<u>Proof</u>

If/

If p is finite then so is rk(p) (5.8.2.1) hence by the previous lemma there cannot be any infinite sequences:

$p=p_1\text{->}p_2\text{->}\dots$

Q.E.D.

5.9 Terms which lead to an error denote $\perp$

The goal of the section is to prove that if p eventually gets stuck then $[\![p]\!]=\perp$. If p is stuck (i.e. p]) then it is in a sort of 'error' state and the action of the 'interpreter' (i.e. ->) is not defined; I explained why I model this kind of undefinedness by $\perp$ in 3.16. The next lemma shows that 'errors' propagate nicely:

5.9.1 Lemma

If fn,a are finite then $[\![fn]\!]([\![a]\!])(\dots\perp\dots)=\perp$

Proof

I do induction on rk(fn)+rk(a) which is finite as fn and a are.

rk(fn)+rk(a)=0:

Then rk(fn)=rk(a)=0 so fn=F but by the definitions of the standard functions car,cdr,cons,atom,eq if F is one of them then $F(\dots\perp\dots)=\perp$

hence $[\![fn]\!]([\![a]\!])(\dots\perp\dots)=F(\dots\perp\dots)=\perp$

rk(fn)+rk(a)>0:

I consider the various cases of fn:

case1:/

<u>case1</u>: fn=F;

Then $[\![fn]\!]([\![a]\!])(\ldots \perp \ldots)=F(\ldots \perp \ldots)=\perp$ (as above)

<u>case2</u>: fn=f

Then $[\![f]\!]([\![a]\!])=[\![a]\!](f)([\![a]\!]) \mid FUN$ (by S6 of 5.3.1)

$=[\![a(f)]\!]([\![a]\!]) \mid FUN$ (by 5.4.2)

<u>case2.1</u>: $a(f)\neq fn'_m$ for any fn', then $[\![a(f)]\!]([\![a]\!]) \mid FUN=\perp$

so $[\![f]\!]([\![a]\!])(\ldots \perp \ldots)=\perp$

<u>case2.2</u>: $a(f)=fn'_m$

Then $[\![f]\!]([\![a]\!])(\ldots \perp \ldots)=[\![fn_m]\!]([\![a]\!]) \mid FUN(\ldots \perp \ldots)$

$=[\![fn']\!]([\![a_m]\!])(\ldots \perp \ldots)$ (by lemma 5.4.6)

<u>case2.2.1</u>: fn' not of the form $\mu^n[f;fn'']$ then by 5.8.2.5

$rk(f)+rk(a)>rk(fn')+rk(a_m)$

so by induction $[\![fn']\!]([\![a_m]\!])(\ldots \perp \ldots)=\perp$

<u>case2.2.2</u>: $fn'=\mu^0[f;fn'']$

Then $[\![fn']\!]([\![a_m]\!])(\ldots \perp \ldots)=[\![\mu^0[f;fn'']]\!]([\![a_m]\!])(\ldots \perp \ldots)$

$=\perp$ (as $[\![\mu^0[f;fn'']]\!]=\perp$)

<u>case2.2.3</u>: $fn'=\mu^s[f;fn''], 0<s<\infty$

Then $[\![fn']\!]([\![a_m]\!])(\ldots \perp \ldots)=[\![\mu^s[f;fn'']]\!]([\![a_m]\!])(\ldots \perp \ldots)$

$=[\![fn'']\!](([\![\mu^{s-1}[f;fn'']]\!]/f)[\![a_m]\!])(\ldots \perp \ldots)$

(by Note 2 of 5.3.1)

$=[\![fn'']\!]([\![(\mu^{s-1}[f;fn'']/f)a_m]\!])(\ldots \perp \ldots)$

(by AS2-4 of 5.3.2)

$=\perp$ (by 5.8.2.6 and induction)

<u>case3</u>:/

<u>case3</u>: $fn=\lambda[[x_1;\ldots;x_m];e]$

Then $\llbracket fn\rrbracket(\llbracket a\rrbracket)(\ldots\perp\ldots)=(\lambda t_1\ldots t_m.\llbracket e\rrbracket((t_1/x_1)\ldots(t_m/x_m)\llbracket a\rrbracket))(\ldots\perp\ldots)$

$=\perp$ (by definition of λ - 3.11.2)

<u>case4</u>: $fn=\mathrm{Label}^m[f;fn'],m<\infty$

Then $\llbracket fn\rrbracket(\llbracket a\rrbracket)(\ldots\perp\ldots)=\llbracket fn'\rrbracket_m((\llbracket fn'\rrbracket/f)\llbracket a\rrbracket)(\ldots\perp\ldots)$ (by 5.3.1)

$=\llbracket fn'\rrbracket(\llbracket(fn'/f)a\rrbracket_m)(\ldots\perp\ldots)$ (by 5.3.1 - Note

$=\llbracket fn'\rrbracket(\llbracket((fn'/f)a)_m\rrbracket)(\ldots\perp\ldots)$ (by 5.4.6)

$=\perp$ (by 5.8.2.7 and induction)

<u>case5</u>: $fn=\mu^m[f;fn'],m<\infty$

If $m=0$ then $\llbracket fn\rrbracket(\llbracket a\rrbracket)(\ldots\perp\ldots)=\perp$ (as $\llbracket fn\rrbracket=\perp$ - S9 of 5.3.1)

If $m>0$ then $\llbracket fn\rrbracket(\llbracket a\rrbracket)(\ldots\perp\ldots)=\llbracket fn'\rrbracket((\llbracket\mu^{m-1}[f;fn']\rrbracket/f)\llbracket a\rrbracket)(\ldots\perp\ldots)$

(by 5.3.1 - Note 2)

$=\llbracket fn'\rrbracket(\llbracket(\mu^{m-1}[f;fn']/f)a\rrbracket)(\ldots\perp\ldots)$

(by AS2-4 of 5.3.2)

$=\perp$ (by 5.8.2.8 and induction)

Q.E.D.

This lemma enables corollary 4.3.4 to be proved without invoking the Main Theorem. A 'limiting' argument is needed - we first prove the result for finite terms and then take the limit over all such terms. If fn,a are in LISP then:

$\llbracket fn\rrbracket/$

$[\![fn]\!]([\![a]\!])(\ldots \perp \ldots)$

$= \bigsqcup_{fn'<fn} \bigsqcup_{a'<a} [\![fn']\!]([\![a']\!])(\ldots \perp \ldots)$ (by lemma 5.5.2)

$= \perp$ by previous lemma.

I can now show that errors denote $\perp$.

5.9.2 <u>Lemma</u>

If p is finite and $p \overset{*}{>} p']$ then $[\![p]\!] = \perp$

<u>Proof</u>

I do induction on rk(p) which is finite as p is

rk(p)=0:

Then p=A and clearly there is no p' such that $p \overset{*}{>} p']$ so there is nothing to prove.

rk(p)>0:

Assume as induction hypothesis that:

$$\forall p_1.(rk(p_1)<rk(p) \text{ and } \exists p_1'.p_1 \overset{*}{>} p_1']) \Rightarrow [\![p_1]\!] = \perp$$

I show that: $p \overset{*}{>} p'] \Rightarrow [\![p]\!] = \perp$

<u>case1</u> p]

Examination of the definition of -> (5.4.7) shows that the only cases that can arise are:

<u>case1.1/</u>

case1.1 $p=(x,a), a(x) \notin$ <S-expression>

Then $[\![p]\!]=[\![x]\!]([\![a]\!]) \quad =[\![a]\!](x)([\![a]\!]) \mid S$ (by S2 of 5.3.1)

$=[\![a(x)]\!]([\![a]\!]) \mid S$ (by 5.4.2)

$=\perp$

case1.2 $p=(F[A_1;\ldots;A_n],a), F(A_1,\ldots,A_n) \notin$ <S-expression>

Then as $F(A_1,\ldots,A_n) \in S=$<S-expression> $\cup \{\perp\}$ it follows that

$[\![p]\!]=F(A_1,\ldots,A_n)=\perp$

case1.3 $p=(fn[e_1;\ldots;e_n],a)$ and for some $m.(e_m,a)\overset{*}{>}p_m]$

(I have used 5.8.4 and 5.7.4 here.)

Then $rk(p)=1+1+rk(fn)+rk(e_1)+\ldots+rk(e_n)+rk(a)$

$>rk((e_m,a))$

so by induction $[\![(e_m,a)]\!]=[\![e_m]\!]([\![a]\!])=\perp$

hence by lemma 5.9.1

$[\![p]\!]=[\![fn]\!]([\![a]\!])([\![e_1]\!]([\![a]\!]),\ldots,[\![e_n]\!]([\![a]\!]))=\perp$

case1.4 $p=([e_{11}\rightarrow e_{12};\ldots;e_{n1}\rightarrow e_{n2}],a)$ and for no m do we have:

$$(e_{m1},a)\overset{*}{>}T \text{ and } \forall\, i<m.(e_i,a)\overset{*}{>}F$$

Now if there existed an m such that:

$$[\![e_{m1}]\!]([\![a]\!])=T \text{ and } \forall\, i<m.[\![e_{i1}]\!]([\![a]\!])=F$$

Then as $T\neq\perp$, $F\neq\perp$ and since $\forall\, i\leq m.rk((e_{i1},a))<rk(p)$, it would follow by induction that for no p_{i1} $(i\leq m)$ could we have $(e_{i1},a)\overset{*}{>}p_{i1}]$

(i/

($i \leq m$). But by lemma 5.8.4 each (e_{i1},a) ($i \leq m$) terminates and so by lemma 5.7.4 there exist A_i ($i \leq m$) such that $(e_{i1},a) \overset{*}{>} A_i$ ($i \leq m$) - but then by lemma 5.4.8 we would have to have A_m=T and $\forall\, i<m. A_i$=F which contradicts the assumption of this case. Thus there cannot exist an m such that:

$$[\![e_{m1}]\!]([\![a]\!])=T \text{ and } \forall\, i<m. [\![e_{i1}]\!]([\![a]\!])=F$$

and so $[\![p]\!]=([\![e_{11}]\!]([\![a]\!])\rightarrow [\![e_{12}]\!]([\![a]\!]),\ldots,[\![e_{n1}]\!]([\![a]\!])\rightarrow [\![e_{n2}]\!]([\![a]\!]))$

$= \perp$ (by definition 3.7.17)

<u>case1.5</u> $p=(f[A_1;\ldots;A_n],a)$ and $a(f)\neq fn_m$ for any fn

Then $[\![p]\!]=[\![f]\!]([\![a]\!])(A_1,\ldots,A_n)$

$=[\![a]\!](f)([\![a]\!]) \mid FUN(A_1,\ldots,A_n)$ (by S6 of 5.3.1)

$=[\![a(f)]\!]([\![a]\!]) \mid FUN(A_1,\ldots,A_n)$ (by 5.4.2)

$= \perp (A_1,\ldots,A_n)$

$= \perp$

<u>case1.6</u> $p=(\lambda[[x_1;\ldots;x_m];e][A_1;\ldots;A_n],a)$ and $m>n$

Then $[\![p]\!]=(\underline{\lambda} t_1\ldots t_m.[\![e]\!]((t_1/x_1)\ldots(t_m/x_m)[\![a]\!]))(A_1,\ldots,A_n)$

$=\lambda$ (by definition of $\underline{\lambda}$ - 3.11.2)

<u>case1.7</u> $p=(\mu^0[f;fn][A_1;\ldots;A_n],a)$

Then $[\![p]\!]=[\![\mu^0[f;fm]]\!]([\![a]\!])(A_1,\ldots,A_n)$

$= \perp (A_1,\ldots,A_n)$ (by S9 of 5.3.1)

$= \perp$

<u>case2/</u>

<u>case2</u> $p \rightarrow p' \overset{*}{\rightarrow} p'']$

As long as p is not of the form $(f[A_1;\dots;A_n],a)$ where $a(f)=\mu^s[f;fn]_m$ then by lemma 5.8.3 $rk(p)<rk(p')$ and so by induction and 5.4.9 $\llbracket p\rrbracket=\llbracket p'\rrbracket=\perp$. If p is of the form $(f[A_1;\dots;A_n],a)$ where $a(f)=\mu^s[f;fn]_m$ then if $s=0$ then $\llbracket p\rrbracket=\llbracket \mu^0[f;fn]\rrbracket_m(\llbracket a\rrbracket)(A_1,\dots,A_n)=\perp$ and if $s>0$ then $p\rightarrow p'\rightarrow p''=(fn[A_1;\dots;A_n],(\mu^{s-1}[f;fn]/f)a_m)$ and so by 5.4.9 and induction $\llbracket p\rrbracket=\llbracket p'\rrbracket=\perp$.

Q.E.D.

5.10 <u>Final step in the proof of the Main Theorem</u>

Collecting together previous lemmas yields the following which entails the Main Theorem by the deduction described in 5.1.4 above.

5.10.1 <u>Lemma</u>

5.10.1.1 $\llbracket p\rrbracket=\bigsqcup_{p'<p}\llbracket p'\rrbracket$

5.10.1.2 $p'<p \Rightarrow p'$ terminates

5.10.1.3 $p'<p, p'\overset{*}{\rightarrow}p''] \Rightarrow \llbracket p'\rrbracket=\perp$

5.10.1.4 $p\overset{*}{\rightarrow}A \Rightarrow \llbracket p\rrbracket=A$

5.10.1.5 $p'<p, p'\overset{*}{\rightarrow}A \Rightarrow p\overset{*}{\rightarrow}A$

<u>Proof/</u>

<u>Proof</u>

5.10.1.1	follows	directly	from	lemma	5.5.2.
5.10.1.2	"	"	"	"	5.8.4 and 5.7.1
5.10.1.3	"	"	"	"	5.9.2 and 5.7.1
5.10.1.4	"	"	"	"	5.4.8
5.10.1.5	"	"	"	"	5.6

Q.E.D.

All that now remains to be done is for me to render the Main Theorem significant by showing that Env1-Env5 are consistent (i.e. showing that there is something that the Main Theorem applies to!)

6. UNIVERSAL SPACE U

The goal of this chapter is to construct a 'universal space' U and to show how it can be used

(1) To construct an alist model Env

(2) To fit syntax equations (BNF) into the theory of semi-domains in a clean way.

(1) is the most important use of U here since BNF is already very well understood and known to be consistent; I shall only sketch out (2). The idea of universal spaces and their uses such as (1), (2) above is due entirely to Scott. The details described here differ in trivial ways from him though e.g. I use semi-domains rather than domains and interpret BNF in a way which yields finite (rather than possibly infinite) expressions.

In the next section I shall briefly discuss the known ways of solving equations such as Env=Id->[Env->D] and indicate why I think using U is the best approach.

6.1 Solving semi-domain equations

There are (to my knowledge) two methods of solving semi-domain equations such as Env=Id->[Env->D], each of these is due to Scott. The first method is to 'classical' inverse limit technique. This is the method/

method originally used to construct mathematical models of the (type free) λ-calculus and it receives its fullest exposition in [17]. The second method uses the algebra of retracts of a suitably constructed 'universal' space.

The first of these techniques, when applied to Env=Id->[Env->D], works by embedding Env_n in Env_m for $n \leq m$ (where Env_n is defined in 5.2.2) and then closing (via an inverse limit) $\bigcup_{n=0} Env_n$ under directed unions. Although this method works perfectly well (in fact it is the one I, at first, used) it has two defects: firstly it confuses general processes common to the solving of all equations with processes particular to the solving of Env=Id->[Env->D] and secondly it fails to clairfy the analogy between semi-domain equations and ordinary recursion equations - the latter being solvable with Y. The second method overcomes both these defects, one starts by choosing a special universal space U (which must be suitable) and then to solve an equation like Env=Id->[Env->D] one represents Id, D as <u>elements</u> of U, -> as a binary operation on U and then treats the equation as an ordinary recursion equation so that the desired solution is (represented by) Y(λe.Id->[e->D]). This obviously overcomes the second defect mentioned above, it overcomes the first because the general processes common to the solving of all equations are factored out in the construction of U which only has to be done once.

There/

There are several universal spaces around, the older ones (e.g. "logical space") are built via inverse limits whilst the newest one is just the power set of the integers, 2^N, interpreted in a subtle way. This interpretation of 2^N is based on an idea of Gordon Plotkin [15] (rediscovered in a slightly different form by Scott) and although it seems to be the most promising approach at present, I shall not adopt it. The reason for this is mainly that it has come on the scene too late for me to fully digest it in time for incorporation here, but also its details are still in a state of flux. The actual use made of the universal space is rather independent of which one it is; for the algebra of retracts, which is the main tool I use, looks (more or less!) the same regardless of which space the retracts are of. It should, I hope, be quite easy to construe most of what follows as being about the new-fangled thing if that emerges the winner.

6.2 Specification of the universal space U used here

Before constructing the particular U I use,I shall characterize it up to isomorphism The kind of "isomorphism" I mean here is not just continuous bijection but rather a bijection which preserves all the structure postulated on U (see definition 6.5.1 below) this structure is described in the following axioms. These axioms have useful consequences,they are not particularly interesting in themselves.

6.2.1/

6.2.1 Axioms

A universal space is a semi-domain U, together with mappings $x \mapsto x_n \in [U\to U]$, for each $n\geq 0$, such that:

(U1) $U\cong[U\to U]$

(U2) U contains two members tt, ff such that tt $\not\sqsubseteq$ ff, ff $\not\sqsubseteq$ tt and $\{x \mid x=x_0\}=\{\perp, tt, ff\}$

(U3) $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots \sqsubseteq x$

(U4) $x=\bigsqcup_n x_n$

(U5) $(x_n)_n=x_n$

(U6) $x_0=\lambda y.x_0=x(\perp)_0$

(U7) $x_{n+1}=\lambda y.x(y_n)_n$

Remarks: (1) In view of (U1) there are isomorphisms

$f: U\to[U\to U]$

$g: [U\to U]\to U$

I identify $u \in U$ with $f(u) \in [U\to U]$ and $F \in [U\to U]$ with $g(F) \in U$ thus for $x,y \in U$ $x(y)$ means $f(x)(y)$ and so $\lambda y.x(y)=g(\lambda y.f(x)(y))=g(f(x))=x$. I have exploited this convention in U6 and U7.

(2) A universal space, as defined above, is the semi-domain analogue of Scott's "logical space". There are other spaces around (e.g. 2^N) that do not satisfy these axioms which one might want to call "universal", to avoid/

avoid confusion from now on whenever I refer to a universal space I mean a structure as defined in 6.2.1 above.

6.3 Existence of universal spaces

The following proposition shows that universal spaces exist, its proof consists in the obvious generalization to semi-domains of Scott's inverse limit construction. Readers already familiar with inverse limits will find nothing new or interesting in the proof and they are advised to skip it.

6.3.1 Proposition

There exists a universal space U

Proof

I construct U as a straightforward inverse limit.

Let $U_0=\text{flat}(\{tt,ff\})$

$U_{n+1}=[U_n\text{->}U_n]$

What follows roughly consists in embedding U_n in U_{n+1} and then closing $\bigcup_{n=0} U_n$ under directed unions.

Let $i_n\colon U_n\text{->}U_{n+1}$

$j_n\colon U_{n+1}\text{->}U_n$

be/

be defined by:

$$i_0(f)=\lambda y\colon U_0.f,\quad i_{n+1}(f)=i_n\circ g\circ j_n$$
$$j_0(F)=F(\perp),\quad j_{n+1}(F)=j_n\circ F\circ i_n$$

6.3.1.1 <u>Lemma</u>

i_n, j_n are continuous and $j_n(i_n(f))=f$

$i_n(j_n(F)\sqsubseteq F$

<u>Proof</u>

i_n, j_n are obviously continuous. I show $j_n(i_n(f))=f$, $i_n(j_n(F)\sqsubseteq F$ by induction on n:

n=0: $j_0(i_0(f))=j_0(\lambda y.f)=(\lambda y.f)(\perp)=f$

$i_0(j_0(F))=i_0(F(\perp))=\lambda y.F(\perp)\sqsubseteq\lambda y.F(y)=F$

n>0: Assume true for n-1 then:

$j_n(i_n(f))=j_n(i_{n-1}\circ f\circ j_{n-1})$ (by definition of i_n)

$=j_{n-1}\circ i_{n-1}\circ f\circ j_{n-1}\circ i_{n-1}$ (by definition of j_n)

$=f$ (by induction)

$i_n(j_n(F))=i_n(j_{n-1}\circ F\circ i_{n-1})$ (by definition of j_n)

$=i_{n-1}\circ j_{n-1}\circ F\circ i_{n-1}\circ j_{n-1}$ (by definition of i_n)

$\sqsubseteq F$ (by induction)

Q.E.D.

I/

I shall write an infinite sequence $(x^0, x^1, x^2, \ldots)$ as $(x^n)_{n=0}^{\infty}$ or just (x^n). Now let U_∞ be the inverse limit of the sequence:

$$\begin{tikzcd} U_0 \arrow[r, shift left, "i_0"] & U_1 \arrow[l, shift left, "j_0"] \arrow[r, shift left, "i_1"] & \arrow[l, shift left, "j_1"] \cdots \arrow[r, shift left, "i_n"] & U_{n+1} \arrow[l, shift left, "j_n"] \arrow[r, shift left, "i_{n+1}"] & \arrow[l, shift left, "j_{n+1}"] \cdots \end{tikzcd}$$

i.e. let $U_\infty = \{(x^n)_{n=0}^{\infty} \mid \forall n \geq 0 . x^n \in U_n \text{ and } x^n = j_n(x^{n+1})\}$

order U_∞ by $(x^n)_{n=0}^{\infty} \sqsubseteq (y^n)_{n=0}^{\infty} \Leftrightarrow \forall\ n \geq 0 . x^n \sqsubseteq_{U_n} y^n$

I claim U_∞ is a universal space, but it takes some work to show it.

6.3.1.2 Lemma

U is a semi-domain and $\bigsqcup X = (\bigsqcup_{U_m} \{x^m \mid (x^n)_{n=0}^{\infty} \in X\})_{m=0}^{\infty}$

Proof

If X is directed then for each $m \geq 0$ so is $\{x^m \mid (x^n)_{n=0}^{\infty} \in X\}$ hence $(\bigsqcup\{x^m \mid (x^n)_{n=0}^{\infty} \in X\})_{m=0}^{\infty}$ is defined, if it is in U_∞ then it is clearly the least upper bound of X, but it is in U_∞ for

$$\begin{aligned} j_m(\bigsqcup\{x^{m+1} \mid (x^n) \in X\}) &= \bigsqcup\{j_m(x^{m+1}) \mid (x^n) \in X\} \\ &= \bigsqcup\{x^m \mid (x^n) \in X\} \end{aligned}$$

Q.E.D.

6.3.1.3/

6.3.1.3 Definition

For $0 \leq p,q \leq \infty$ define $k_{pq}: U_p \to U_q$ by:

(1) If $p,q < \infty$ then $k_{pq} = \begin{cases} I_{U_p} & \text{if } p=q \\ i_{q-1} \circ \dots \circ i_p & \text{if } p<q \\ j_q \circ \dots \circ j_{p-1} & \text{if } p>q \end{cases}$

(2) If $p<\infty$ then $k_{p\infty}(x) = (k_{pq}(x))_{q=0}^{\infty}$

(3) If $q<\infty$ then $k_{\infty q}((x^n)_{n=0}^{\infty}) = x^q$

(4) $k_{\infty\infty}(x) = x.$

6.3.1.4 Lemma

For $0 \leq p,q,r \leq \infty$

(1) $k_{qr} \circ k_{pq} \sqsubseteq k_{pr}$

(2) $q \geq p$ or $q \geq r \Rightarrow k_{qr} \circ k_{pq} = k_{pr}$

(3) $k_{pr} = \bigsqcup_{q=0}^{\infty} k_{qr} \circ k_{pq}$

Proof

There are a large number of separate cases to consider, each follows straightforwardly (but tediously!) from the definitions and lemma 6.3.1.1. I shall just do the hardest case viz.

$$k_{\infty\infty} = \bigsqcup_{q=0}^{\infty} k_{q\infty} \circ k_{\infty q}$$

To/

To prove this I need to show

$$(x^p)_{p=0}^{\infty}=\bigsqcup_{q=0}^{\infty}k_{q\infty}(x^q) \qquad (\text{as } k_{\infty q}((x^p)_{p=0}^{\infty})=x^q)$$

but $\bigsqcup_{q=0}^{\infty} k_{q\infty}(x^q)=\bigsqcup_{q=0}^{\infty}(k_{qp}(x^q))_{p=0}^{\infty}$ (by definition of $k_{q\infty}$)

$=(\bigsqcup_{q=0}^{\infty} k_{qp}(x^q))_{p=0}^{\infty}$ (by 6.3.1.2)

$=(x^p)_{p=0}^{\infty}$ (since $q\geq p \Rightarrow k_{qp}(x^q)=j_p(\ldots(j_{q-1}(x^q))\ldots)=x^p$ and

$q<p \Rightarrow k_{qp}(x^q)=k_{qp}\circ k_{\infty q}((x^p)_{p=0}^{\infty})$

$\sqsubseteq k_{\infty p}((x^p)_{p=0}^{\infty})$ (by (1) of 6.3.1.4)

$=x^p$)

Q.E.D.

Now I can show U_∞ satisfies (U1)

6.3.1.5 <u>Lemma</u>

$U_\infty \cong [U_\infty \to U_\infty]$

<u>Proof</u>

Define $i: U\to[U\to U]$

$j: [U\to U]\to U$

by $i(x)=\lambda y{:}U_\infty.(k_{\infty(n+1)}(x)(k_{\infty n}(y)))_{n=0}^{\infty}$

$j(f)=(j_n(\lambda y{:}U_n.k_{\infty n}(f(k_{n\infty}(y)))))_{n=0}^{\infty}$

Clearly i,j are continuous, also:

i/

$i(j(f))(y)=(k_{\infty(n+1)}(j(f))(k_{\infty n}(y)))_{n=0}^{\infty}$ (by definition of i)

$=(j_{n+1}(\lambda y:U_{n+1}\cdot k_{\infty(n+1)}(f(k_{(n+1)\infty}(y))))(k_{\infty n}(y)))_{n=0}^{\infty}$

(by definition of j and $k_{\infty(n+1)}$)

$=((j_n\circ(\lambda y:U_{n+1}\cdot k_{\infty(n+1)}(f(k_{(n+1)\infty}(y))))\circ i_n)(k_{\infty n}(y)))_{n=0}^{\infty}$

(definition of j_{n+1})

$=(j_n(k_{\infty(n+1)}(f(k_{(n+1)\infty}(i_n(k_{\infty n}(y)))))))_{n=0}^{\infty}$

$=((j_n\circ k_{\infty(n+1)}\circ f\circ k_{(n+1)\infty}\circ i_n\circ k_{\infty n})(y))_{n=0}^{\infty}$

$=((k_{\infty n}\circ f\circ k_{n\infty}\circ k_{\infty n})(y))_{n=0}^{\infty}$

(by lemma 6.3.1.4 - (2) -)

$=\bigsqcup_{n=0}(k_{n\infty}\circ k_{\infty n}\circ f\circ k_{n\infty}\circ k_{\infty n})(y)$

(by lemma 6.3.1.4 - (3)-)

$=k_{\infty\infty}\circ f\circ k_{\infty\infty}(y)$ (by lemma 6.3.1.4 - (3) -)

$=f(y)$

Thus $i\circ j=I_{[U_\infty\to U_\infty]}$

Now:

$j(i(x))=j(\lambda y.(k_{\infty(n+1)}(x)(k_{\infty n}(y)))_{n=0}^{\infty})$ (by definition of i)

$=(j_m(\lambda y:U_m\cdot k_{\infty m}((k_{\infty(n+1)}(x)(k_{\infty n}(k_{m\infty}(y))))_{n=0}^{\infty})))_{m=0}^{\infty}$ (by definition of j)

$=(j_m(\lambda y:U_m\cdot k_{\infty(m+1)}(x)(k_{\infty m}(k_{m\infty}(y)))))_{m=0}^{\infty}$ (by definition of $k_{\infty m}$)

$=(j_m(\lambda y:U_m\cdot k_{\infty(m+1)}(x)(y)))_{m=0}^{\infty}$ (by lemma 6.3.1.4 - (2) -)

$=(j_m(k_{\infty(m+1)}(x)))_{m=0}^{\infty}$ (by extensionality)

=/

$=(k_{\infty m}(x))_{m=0}^{\infty}$ (by lemma 6.3.1.4 - (2) -)

$= \bigsqcup_{m}^{\infty} k_{m\infty}(k_{\infty m}(x))$ (by lemma 6.3.1.4 - (3) -)

$=x$ (by lemma 6.3.1.4 - (3) -)

so $j \circ i = I_{U_\infty}$

Thus i,j are isomorphisms.

Q.E.D.

To show that U_∞ satisfies U2-U6 define

$$x_n = k_{n\infty}(k_{\infty n}(x))$$

Then $\{x \mid x=x_0\} = \{(x^n)_{n=0}^{\infty} \mid (x^n)_{n=0}^{\infty} = k_{0\infty}(x^0)\}$

$= \{(x^n)_{n=0}^{\infty} \mid \forall n \geq 0.\, x^n = k_{0n}(x^0)\}$

$= \{(k_{0n}(x^0))_{n=0}^{\infty} \mid x^0 \in U_0\}$

$= \{k_{0\infty}(x^0) \mid x^0 \in U_0\}$

Now $\perp_{U_\infty} = k_{0\infty}(\perp_{U_0})$ (as $\forall n.\ \perp_{U_n} = k_{0n}(\perp_{U_0})$)
and if $tt_\infty = k_{0\infty}(tt), ff_\infty = k_{0\infty}(ff)$ then as $tt \not\sqsubseteq ff$ and $ff \not\sqsubseteq tt$
(by definition of U_0) $tt_\infty \not\sqsubseteq ff_\infty$, $ff_\infty \not\sqsubseteq tt_\infty$ and
$\{x \mid x=x_0\} = \{\perp, tt_\infty, ff_\infty\}$ so U2 holds. From now on I shall identify
tt with tt_∞ and ff with ff_∞.

To show U3 we have

$x_{n+1}/$

$$
\begin{aligned}
x_{n+1} &= k_{(n+1)\infty} \circ k_{\infty(n+1)}(x) \\
&\sqsupseteq k_{n\infty} \circ k_{(n+1)n} \circ k_{n(n+1)} \circ k_{\infty n}(x) && \text{(by lemma 6.3.1.4 - (1) -)} \\
&= k_{n\infty}(k_{\infty n}(x)) && \text{(by lemma 6.3.1.4 - (2) -)} \\
&= x_n
\end{aligned}
$$

U4 follows directly from $k_{\infty\infty} = \bigsqcup_{n=0}^{\infty} k_{n\infty} \circ k_{\infty n}$ of lemma 6.3.1.4.

To show U5 we have

$$
\begin{aligned}
(x_n)_n &= (k_{n\infty} \circ k_{\infty n} \circ k_{n\infty} \circ k_{\infty n})(x) \\
&= k_{n\infty} \circ k_{\infty n}(x) \quad \text{(by lemma 6.3.1.4-(2) -)} \\
&= x_n
\end{aligned}
$$

U6 and U7 are a bit more tricky. Here is a lemma to help with U6.

6.3.1.6 <u>Lemma</u>

$$\forall x \in U_0 . \forall y \in U_n . k_{0(n+1)}(x)(y) = k_{0n}(x)$$

<u>Proof</u>

Induction on n:

n=0:

$k_{01}(x)(y) = i_0(x)(y) = x = i_{00}(x)$

n>0/

$n>0$: Assume true for $n-1$ then:

$k_{0(n+1)}(x)(y)=i_n(k_{0n}(x))(y)$ (by definition of $k_{0(n+1)}$ - 6.3.1.3)

$=(i_{n-1}\circ k_{0n}(x)\circ j_{n-1})(y)$ (by definition of i_n)

$=i_{n-1}(k_{0(n-1)}(x))$ (by induction)

$=k_{0n}(x)$ (by definition of k_{0n} - 6.3.1.3)

Q.E.D.

Now to show U6 holds of U_∞ let $x=(x^n)_{n=0}^{\infty}, y=(y^n)_{n=0}^{\infty}$ then

$x_0=k_{0\infty}(k_{\infty 0}(x))$

$=k_{0\infty}(x^0)$ (by definition of $k_{\infty 0}$ - 6.3.1.3)

$=(k_{0n}(x^0))_{n=0}^{\infty}$ (by definition of $k_{0\infty}$ - 6.3.1.3)

so $x_0(y)=(k_{0(n+1)}(x^0)(y^n))_{n=0}^{\infty}$ (as "$x_0(y)$" means $i(x_0)(y)$)

$=(k_{0n}(x^0))_{n=0}^{\infty}$ (by lemma 6.3.1.6)

$=x_0$ (by definition of x_0)

and $x(\perp)_0=k_{0\infty}(k_{\infty 0}(x(\perp)))$

$=k_{0\infty}(x^1(\perp^0))$ (as "$x(\perp)$" means $(x^{n+1}(\perp^n))_{n=0}^{\infty}$)

$=k_{0\infty}(j_0(x^1))$ (by definition of j_0)

$=k_{0\infty}(x^0)$ (as $\forall n.\ x^n=j_n(x^{n+1})$)

$=x_0$

hence U6.

To verify U7 holds another lemma is needed:

6.3.1.7/

6.3.1.7 <u>Lemma</u>

Let $x=(x^n)_{n=0}^{\infty}$, $y=(y^n)_{n=0}^{\infty} \in U_\infty$ then:

(1) $n \geq m \Rightarrow k_{(n+1)(m+1)}(x^{n+1})(y^m) \sqsubseteq k_{nm}(x^{n+1}(y^n))$

(2) $n<m \Rightarrow k_{(n+1)(m+1)}(x^{n+1})(y^m)=k_{nm}(x^{n+1}(y^n))$

<u>Proof</u>

If $n=m$ then both sides of (1) are equal to $x^{n+1}(y^n)$

If $n>m$ then:

$$
\begin{aligned}
k_{(n+1)(m+1)}(x^{n+1})(y^m) &= (j_{m+1}\circ \cdots \circ j_n)(x^{n+1})(j_m\circ \cdots \circ j_{n-1}(y^n)) \\
&\qquad \text{(by definition of } k_{(n+1)(m+1)} \text{ and } \forall n.\, y^n=j_n(y^{n+1})) \\
&= (j_m\circ \cdots \circ j_{n-1}\circ x^{n+1}\circ i_{n-1}\circ \cdots \circ i_m \circ j_m \circ \cdots \circ j_{n-1})(y^n) \\
&\qquad \text{(by definition of } j_{m+1},\ldots,j_n) \\
&\sqsubseteq (j_m\circ \cdots \circ j_{n-1}\circ x^{n+1})(y^n) \\
&\qquad \text{(by repeated use of 6.3.1.1)} \\
&= k_{nm}(x^{n+1}(y^n)) \quad \text{(by definition of } k_{nm})
\end{aligned}
$$

hence (1).

If $n<m$ then:

$k_{(n+1)}$/

so

$$x(y_n)_n=(k_{nm}(k_{\infty n}(x(y_n))))_{m=0}^{\infty}$$

(by definition of $k_{n\infty}$)

$$=(k_{nm}(x^{m+1}(k_{nn}(y^n))))_{m=0}^{\infty}$$

(by previous calculation and definition of $k_{\infty n}$)

$$=(k_{nm}(x^{m+1}(y^n)))_{m=0}^{\infty}$$

(by definition of k_{nn})

hence it follows from lemma 6.3.1.7 that $x_{n+1}(y)=x(y_n)_n$.

Thus U_∞ satisfies U1-U7 and so is a universal space.

Q.E.D.

6.4 Properties of universal spaces

The knowledge that universal spaces exist makes properties of them more interesting than they would otherwise be! The next few lemmas give some useful properties, some of these are needed to prove that (in a strong sense) there is only one universal space.

6.4.1 Lemma

If U is a universal space and $x \in U$ then:

$$x_{nm}=x_{\min\{n,m\}}$$

Proof

I/

I do induction on n and m.

n=m=0: $x_{00}=x_0$ by U5

n>0: suppose $\forall x \in U.\ x_{(n-1)0}=x_0$

$x_{n0}=x_n(\perp)_0$ (by U6)

$=x(\perp_{n-1})_{(n-1)0}$ (by U7)

$=x(\perp)_0$ (by induction)

$=x_0$ (by U6)

m>0: suppose $\forall x \in U.\ x_{0(m-1)}=x_0$ then:

$x_{0m}=\lambda y.x_{0m}(y)$ (by extensionality)

$=\lambda y.x_0(y_{m-1})_{m-1}$ (by U7)

$=\lambda y.x_{0(m-1)}$ (by U6: $x_0(y_{m-1})=x_0$)

$=\lambda y.x_0$ (by induction)

$=x_0$ (by U6)

So by induction $\forall n.x_{0n}=x_{n0}=x_0$.

n>0,m>0: suppose $\forall x \in U.\ x_{(n-1)(m-1)}=x_{\min\{(n-1),(m-1)\}}$ then:

$x_{nm}=\lambda y.x_{nm}(y)$ (by extensionality)

$=\lambda y.x_n(y_{m-1})_{m-1}$ (by U7)

$=\lambda y.x(y_{(m-1)(n-1)})_{(m-1)(n-1)}$ (by U7)

$=\lambda y.x(y_{\min\{(m-1),(n-1)\}})_{\min\{(m-1),(n-1)\}}$ (by induction)

$=\lambda y.x_{(\min\{(m-1),(n-1)\}+1)}(y)$ (by U7)

$=x_{\min\{n,m\}}$ (by extensionality and $\min\{n,m\}=\min\{(m-1),(n-1)\}+1$)

Q.E.D.

6.4.2/

6.4.2 Lemma

If U is a universal space and $x \in U$ then:

$$x = \lambda y.x \Leftrightarrow x = x_0$$

Proof

By U6 $x=x_0 \Rightarrow x=x_0=\lambda y.x_0=\lambda y.x$

I show the converse by proving by induction on n that:

$$\forall x \in U.\ x=\lambda y.x \Rightarrow x_n=x_0$$

For n=0 this is trivial, suppose it is true for n and $x=\lambda y.x$.

$$\begin{aligned} \text{Then } x_{n+1} &= \lambda y.x(y_n)_n && \text{(by U7)}\\ &= \lambda y.x_n && \text{(by assumption on x)}\\ &= \lambda y.x_0 && \text{(by induction)}\\ &= x_0 && \text{(by U6)} \end{aligned}$$

Thus $x=\lambda y.x \Rightarrow \forall n \geq 0.\ x_n=x_0 \Rightarrow \bigsqcup_n x_n=x_0 \Rightarrow x=x_0$ (by U4)

Q.E.D.

6.4.3 Lemma

If U is a universal space and $x \in U$ then:

tt/

$tt \sqsubseteq x \Rightarrow tt=x$

$ff \sqsubseteq x \Rightarrow ff=x$

(i.e. tt,ff are maximal)

<u>Proof</u>

I prove by induction on n that:

$$\forall x \in U. tt \sqsubseteq x \Rightarrow tt=x_n$$

n=0:

$tt \sqsubseteq x \Rightarrow tt_0 \sqsubseteq x_0 \Rightarrow tt \sqsubseteq x_0$ (by U2: $tt=tt_0$)

$\Rightarrow tt=x_0$ (by U2)

n>0: Assume true for (n-1) then as $tt=tt_0 \sqsubseteq \dots \sqsubseteq tt_n \sqsubseteq \dots \sqsubseteq tt$ (by U3):

$tt \sqsubseteq x \Rightarrow tt(y_{n-1})_{n-1} \sqsubseteq x(y_{n-1})_{n-1}$

$\Rightarrow tt_n(y) \sqsubseteq x(y_{n-1})_{n-1}$ (by U7)

$\Rightarrow tt \sqsubseteq x(y_{n-1})_{n-1}$ (as $tt_n(y)=tt(y)=tt$)

$\Rightarrow tt =x(y_{n-1})_{(n-1)(n-1)}$ (by induction)

$\Rightarrow tt =x_n(y)$ (by U5 and U7)

$\Rightarrow tt =x_n$ (as $\lambda y.\ tt(y)=tt$)

Thus the induction goes through. Now:

$$tt \sqsubseteq x \Rightarrow \forall n. tt=x_n \Rightarrow tt=\bigsqcup_n x_n=x$$

Q.E.D.

Remark/

Remark: This lemma is not true about "logical space" (T_∞) for in that we have $tt \sqsubseteq \top$ but $tt \neq \top$. As a result of this the conditional (see definition 6.6.1) has a less pleasant definition in "logical space" than that in its semi-domain analogue.

6.4.4 <u>Lemma</u>

If U is a universal space and $x \in U$ then:

$x_0=tt \Rightarrow x=tt$

$x_0=ff \Rightarrow x=ff$

<u>Proof</u>

$x_0=tt \Rightarrow tt=x_0 \sqsubseteq x \Rightarrow x=tt$ (by lemma 6.4.3)

$x_0=ff \Rightarrow ff=x_0 \sqsubseteq x \Rightarrow x=ff$ (by lemma 6.4.3)

Q.E.D.

6.4.5 <u>Definition</u>

Let $U_0=\text{flat}(\{tt,ff\})$

$U_{n+1}=[U_n \to U_n]$

Remark: U_n is the same as in the proof of proposition 6.3.1.

6.4.6 <u>Lemma</u>

If U is a universal space then for each $n \geq 0$ $\{x \mid x=x_n\}$ is a sub semi-/

semi-domain of U isomorphic to U_n. In fact there exist isomorphisms:

$$f_n: U_n \to \{x \mid x = x_n\}$$
$$g_n: \{x \mid x = x_n\} \to U_n$$

such that: (1) $\forall\, x \in U_{n+1}.\ \forall\, y \in U_n.\ f_n(x(y)) = f_{n+1}(x)(f_n(y))$

(2) $\forall\, x \in \{x \mid x = x_{n+1}\}.\ \forall\, y \in \{x \mid x = x_n\}.\ g_n(x(y)_n) = g_{n+1}(x)(g_n(y))$

Proof

By U2 we can take f_0, g_0 to be the identity mappings. Then define f_{n+1}, g_{n+1} inductively by:

$$f_{n+1}(x) = (\lambda u{:}U.f_n(x(g_n(u_n))))_{n+1}$$
$$g_{n+1}(x) = \lambda u{:}U_n.g_n(x(f_n(u))_n)$$

Clearly f_0, g_0 are isomorphisms. Assume f_n, g_n are isomorphisms. I show f_{n+1}, g_{n+1} are also:

$$\begin{aligned}
f_{n+1}(g_{n+1}(x))(u) &= (\lambda u{:}U.\ f_n(g_{n+1}(x)(g_n(u_n))))(u_n)_n \quad \text{(by U7)}\\
&= f_n(g_{n+1}(x)(g_n(u_{nn})))_n\\
&= f_n(g_{n+1}(x)(g_n(u_n))) \quad \text{(by U5)}\\
&= f_n(g_n(x(f_n(g_n(u_n)))_n)) \quad \text{(by definition of } g_{n+1})\\
&= x(u_n)_n \quad \text{(by induction)}\\
&= x_{n+1}(u) \quad \text{(by U7)}\\
&= x(u) \quad \text{if } x \in \{x \mid x = x_{n+1}\}
\end{aligned}$$

also/

also:

$$
\begin{aligned}
g_{n+1}(f_{n+1}(x))(u) &= g_n(f_{n+1}(x)(f_n(u))_n) \\
&= g_n((\lambda u{:}U.\ f_n(x(g_n(u_n))))(f_n(u)_n)_n) \quad \text{(by U7)} \\
&= g_n(f_n(x(g_n(f_n(u)_{nn})))_n) \\
&= g_n(f_n(x(g_n(f_n(u))))) \quad \text{(by U5)} \\
&= x(u) \quad \text{(by induction)}
\end{aligned}
$$

Thus $\forall$ n. f_n, g_n are semi-domain isomorphisms.

To show (1) calculate as follows:

$$
\begin{aligned}
f_{n+1}(x)(f_n(y)) &= f_n(x(g_n(f_n(y)_n)))_n \quad \text{(by definition of } f_{n+1} \text{ and U7)} \\
&= f_n(x(g_n(f_n(y)))) \quad \text{(by U5)} \\
&= f_n(x(y))
\end{aligned}
$$

To show (2) we have:

$$
\begin{aligned}
g_{n+1}(x)(g_n(y)) &= g_n(x(f_n(g_n(y)))_n) \quad \text{(by definition of } g_{n+1}) \\
&= g_n(x(y)_n)
\end{aligned}
$$

Q.E.D.

6.4.7 Definition

An element x of a semi-domain is finite <=>

$$Z \text{ directed},\ x \sqsubseteq \bigsqcup Z \Rightarrow \exists z \in Z.\ x \sqsubseteq z$$

Remark/

Remark: This is an abstract notion of finiteness which is analogous to the usual one; it generalizes that since in the semi-domain of subsets of a set (ordered by inclusion) a subset is finite in the above sense iff it is finite in the usual one. Intuitively definition 6.4.7 makes an element <u>not</u> finite if it is a non-trivial limit. Notice that every member of a finite semi-domain is finite.

6.4.8 <u>Lemma</u>

If U is a universal space and $x \in U$ then:

$$x \text{ is finite} \Leftrightarrow \exists n.\ x=x_n$$

<u>Proof</u>

If x is finite then as $x=\bigsqcup_n x_n$ definition 6.4.7 implies that $x \sqsubseteq x_n$ and hence $x=x_n$ by U3. Conversely if $x=x_n \sqsubseteq \bigsqcup Z$ where $Z \subseteq U$ is directed then $x=x_n \sqsubseteq (\bigsqcup Z)_n = \bigsqcup\{z_n \mid z \in Z\}$. But the sub semi-domain $\{x \in U \mid x=x_n\}$ is finite (by 6.4.6) so each x_n is finite <u>in it</u>; now $\{z_n \mid z \in Z\}$ is directed so $x_n \sqsubseteq z_n \sqsubseteq z \in Z$ for some $z_n \in \{z_n \mid z \in Z\}$, thus x is also finite in U.

Q.E.D.

6.5/

6.5 Uniqueness of universal space

The axioms of 6.2.1 were chosen so that anything satisfying them would essentially 'be' U_∞ to make this precise here is a definition.

6.5.1 Definition

A mapping $f: U' \to U''$ between two universal spaces U', U'' is a homomorphism $\Leftrightarrow$ (1) f is continuous

(2) $\forall\, x', y' \in U'. f(x'(y')) = f(x')(f(y'))$

(3) $\forall\, x' \in U'.\ \forall\, n \geq 0. f(x'_n) = f(x')_n$

Two universal space U', U'' are isomorphic $\Leftrightarrow$ there exist homomorphisms $f: U' \to U''$, $g: U'' \to U'$ which are inverse to each other (i.e. $f \circ g = I_{U''}, g \circ f = I_{U'}$).

Remark: Thus a homomorphism from one universal space to another is a mapping which preserves directed unions, the applicative structure and the projections (i.e. $x \mapsto x_n$). Isomorphic universal spaces are isomorphic as semi-domains (as in definition 3.7.2) but they are related in a stronger sense than just this.

6.5.2 Proposition

If U', U'' are universal spaces then they are isomorphic.

Proof/

<u>Proof</u>

Let $f'_n: U_n \to \{x' \in U' \mid x' = x'_n\}$, $f''_n: U_n \to \{x'' \in U'' \mid x'' = x''_n\}$

$g'_n: \{x' \in U' \mid x' = x'_n\} \to U_n$, $g''_n: \{x'' \in U'' \mid x'' = x''_n\} \to U_n$

be the mappings of lemma 6.4.6. Then define:

$f: U' \to U''$

$g: U'' \to U'$

by $f(x) = \bigsqcup_n f''_n(g'_n(x'_n))$

$g(x) = \bigsqcup_n f'_n(g''_n(x''))$

I claim that f,g are homomorphisms inverse to each other. To show that they are well defined I need to show that the unions defining them are over directed sets.

Now $f''_1(g'_1(x'_1))(u) = f''_0(g'_1(x'_1(g''_0(u_{00})))_0$ (by definition of f''_1 and U7)

$= f''_0(g'_0(x'_1(f'_0(g''_0(u_0)))_0))$ (by definition of g'_1 and U5)

$= f''_0(g'_0(x'(f'_0(g''_0(u_0)))_0))$ (by U7 and U5)

$\sqsupseteq f''_0(g'_0(x'(\perp)_0))$

$= f''_0(g'_0(x'_0))$ (by U6)

$= f''_0(g'_0(x'_0))(u)$ (by U6)

hence $f''_0(g'_0(x'_0)) \sqsubseteq f''_1(g'_1(x'_1))$. Now suppose, as induction hypothesis, that: $\forall x. f''_n(g'_n(x'_n)) \sqsubseteq f''_{n+1}(g'_{n+1}(x'_{n+1}))$ then:

f''_{n+1}/

$$\begin{aligned} f''_{n+1}(g'_{n+1}(x'_{n+1}))(u) &= f''_n(g'_{n+1}(x'_{n+1})(g''_n(u_{nn})))_n && \text{(by definition of } f''_{n+1} \text{ andU7)} \\ &= f''_n(g'_n(x'_{n+1}(f'_n(g''_n(u_n)))_n)) && \text{(by definition of } g'_{n+1} \text{ and U5)} \\ &= f''_n(g'_n(x'(f'_n(g''_n(u_n)))_n)) && \text{(by U7 and U5)} \\ &\sqsubseteq f''_{n+1}(g'_{n+1}(x'(f'_{n+1}(g''_{n+1}(u_{n+1})))_{n+1})) && \text{(by induction)} \\ &= f''_{n+2}(g'_{n+2}(x'_{n+2}))(u) && \text{(reverse first 3 lines of this} \\ & && \text{calculation with n+1 for n)} \end{aligned}$$

Hence by induction on n:

$$\forall x.\ \forall n. f''_n(g'_n(x'_n)) \sqsubseteq f''_{n+1}(g'_{n+1}(x'_{n+1})) \qquad (1)$$

and by symmetry:

$$\forall x.\ \forall n. f'_n(g''_n(x''_n)) \sqsubseteq f'_{n+1}(g''_{n+1}(x''_{n+1})) \qquad (2)$$

Thus the unions defining f,g are over directed sets (in fact over chains) and so these functions are well defined. They are the limits of continuous functions and hence are continuous. To show that they are inverses of each other we have:

$$\begin{aligned} f(g(x'')) &= \bigsqcup_n f''_n(g'_n(g(x'')_n)) && \text{(by definition of f)} \\ &= \bigsqcup_n \bigsqcup_m f''_n(g'_n(f'_m(g''_m(x''_m)))_n)) && \text{(by definition of g)} \\ &= \bigsqcup_n f''_n(g'_n(f'_n(g''_n(x''_n)))_n)) && \text{(by (1), (2) and a cofinality argument)} \\ &= \bigsqcup_n f''_n(g'_n(f'_n(g''_n(x''_n)))) && \text{(by U5)} \\ &= \bigsqcup_n x''_n && \text{(by 6.4.6)} \\ &= x'' && \text{(by U4)} \end{aligned}$$

and/

and by symmetry $g(f(x'))=x'$.

f,g preserve the applicative structure on U for:

$$\begin{aligned}
f(x'(y')) &= \bigsqcup_n f''_n(g'_n(x'(y')_n)) && \text{(by definition of } f\text{)}\\
&= \bigsqcup_n f''_n(g'_n(x'_{n+1}(y'_n)_n)) && \text{(by U7, U4 and a cofinality argument)}\\
&= \bigsqcup_n f''_n(g'_{n+1}(x'_{n+1})(g'_n(y'_n))) && \text{(by 6.4.6 - (2) -)}\\
&= \bigsqcup_n f''_{n+1}(g'_{n+1}(x'_{n+1}))(f''_n(g'_n(y'_n)))\\
&= f(x)(f(y')) && \text{(by definitions of } f,g \text{ and cofinality)}
\end{aligned}$$

and by symmetry $g(x''(y''))=g(x'')(g(y''))$. Thus to complete the proof I just have to show that

$$\forall x' \in U'.\ \forall n\geq 0.\ f(x'_n)=f(x')_n$$
$$\forall x \in U''.\ \forall n\geq 0.\ g(x''_n)=g(x'')_n$$

By symmetry I need only verify one of these, say $f(x'_n)=f(x')_n$, I do this by induction on n.

$n=0$:

I must show $f(x'_0)=f(x')_0$

Now $f(x'_0)(f(y'))=f(x'_0(y'))$

$=f(x'_0)$ (by U6)

so $f(x'_0)=\lambda y''.f(x'_0)$ (as f is onto)

so by lemma 6.4.2 $f(x'_0)=f(x'_0)_0 \sqsubseteq f(x')_0$ and hence by U2

$f(x')_0$/

$f(x')_0$, $f(x'_0) \in \{\perp, tt, ff\}$. If $f(x')_0 = \perp$ then $f(x'_0) \sqsubseteq f(x')_0 = \perp$ so $f(x'_0) = \perp = f(x'_0)$ and if $f(x')_0 = tt$ then by lemma 6.4.4 $f(x') = tt$ so $x' = g(tt)$

$$\Rightarrow x'_0 = g(tt)_0$$
$$\Rightarrow f(x'_0) = f(g(tt)_0)$$

by symmetry $f(x')_0 = ff \Rightarrow f(x'_0) = f(g(ff)_0)$. But I shall show below that $f(tt) = tt$, $g(tt) = tt$ and so $f(g(tt)_0) = tt$, $f(g(ff)_0) = ff$ and this will complete the proof of the n=0 case.

Now $f(tt) = \bigsqcup_n f''_n(g'_n(tt_n))$ (by definition of f)

$= \bigsqcup_n f''_n(g'_n(tt))$ (as $tt = tt_0 \sqsubseteq tt_n \sqsubseteq tt$)

Suppose $f''_n(g'_n(tt)) = f''_0(g'_0(tt))$ then

$f''_{n+1}(g'_{n+1}(tt))(u) = f''_n(g'_n(tt(f'_n(g''_n(u_n)))_n))$ (by the calculation used to establish (1) with tt for x').

$= f''_n(g'_n(tt))$ (by U6)

$= f''_0(g'_0(tt))$ (by assumption)

Thus by induction on n: $\forall\, n \geq 0. f''_n(g'_n(tt)) = f''_0(g'_0(tt)) = tt$ hence $f(tt) = tt$ and by symmetry $g(tt) = tt$.

$n > 0$:

Assume $\forall x' \in U'.\ f(x'_{n-1}) = f(x')_{n-1}$ then

$f(x'_n)$/

$f(x_n')(f(y))=f(x_n'(y))$ (as shown above)

$=f(x'(y_{n-1})_{n-1})$ (by U7)

$=f(x'(y_{n-1}))_{n-1}$ (by induction)

$=f(x')(f(y_{n-1}))_{n-1}$ (as shown above)

$=f(x')(f(y)_{n-1})_{n-1}$ (by induction)

$=f(x')_n(f(y))$ (by U7)

hence as f is onto $f(x_n')=f(x')_n$.

Q.E.D.

In view of this lemma it makes sense to talk about the universal space, or simply just "universal space" U.

6.6 Representing things in universal space

In this section I shall show how to represent various useful kinds of entities as elements in U.

6.6.1 Definition

6.6.1.1 $(z\supset x,y)=\begin{cases} x & \text{if } z=tt \\ y & \text{if } z=ff \\ \perp & \text{otherwise}\end{cases}$

6.6.1.2 $(x,y)=\lambda z.(z\supset x,y)$

6.6.1.3 $fst=\lambda u.u(tt)$

$snd=\lambda u.u(ff)$

6.6.1.4/

<u>6.6.1.4</u> $\mathrm{def}(x) = \begin{cases} tt & \text{if } x \neq \perp \\ \perp & \text{otherwise} \end{cases}$

<u>6.6.1.5</u> $\langle x,y\rangle = (\mathrm{def}(x) \supset (\mathrm{def}(y) \supset (x,y), \perp), \perp)$

Remarks: (1) read "$(z \supset x,y)$" as "if z then x else y".

(2) $\mathrm{fst}((x,y))=x$, $\mathrm{snd}((x,y))=y$

6.6.2 <u>Lemma</u>

The expressions defined in 6.6.1 are continuous in all their variables.

<u>Proof</u>

<u>6.6.1.1</u>:

Let $Z \subseteq U$ be directed I show $(\bigsqcup Z \supset x,y) = \bigsqcup_{z \in Z} (z \supset x,y)$ by cases

case1:

If $\bigsqcup Z = tt$ then as tt is finite $tt \sqsubseteq z$ for some $z \in Z$ and so by lemma 6.4.3 $tt = z \in Z$. If $ff \in Z$ then $ff \sqsubseteq \bigsqcup Z = tt$ which is impossible by U2, so $ff \notin Z$. Hence $(\bigsqcup Z \supset x,y) = x = \bigsqcup_{z \in Z} (z \supset x,y)$.

case2:

If $\bigsqcup Z = ff$ then proceed as in case1 above with tt replaced by ff.

case3:

If $\bigsqcup Z \neq tt$ and $\bigsqcup Z \neq ff$ then by lemma 6.4.3 $tt \notin Z$ and $ff \notin Z$ so $(\bigsqcup Z \supset x,y) = \perp = \bigsqcup_{z \in Z} (z \supset x,y)$.

Showing/

Showing $(z \supset x,y)$ is continuous in x and y is trivial.

<u>6.6.1.2</u>: Trivial.

<u>6.6.1.3</u>: Trivial.

<u>6.6.1.4</u>:

If Z is directed then $\bigsqcup Z \neq \perp \iff \exists\, z \in Z.\, z \neq \perp$ hence $\mathrm{def}(\bigsqcup Z) = \bigsqcup_{z \in Z} \mathrm{def}(z)$

<u>6.6.1.5</u>: Trivial.

Q.E.D.

Recall that one of the purposes of introducing U was to get a space in which semi-domains could be represented as elements so that semi-domain equations could be solved with Y.

If $u \in U$ what semi-domain does u represent? Well, with u is naturally associated the equation

$$x=u(x)$$

and if x 'belongs' to u in the sense of being a solution to this then I shall write $x \in u$ and then take the semi-domain represented by u to be $\{x \mid x \in u\}$, I shall call this semi-domain $\mathrm{fix}(u)$ so that in general:

If $u \in U$ then $\mathrm{fix}(u)=\{x \mid x=u(x)\}$

6.6.3/

6.6.3 Lemma

If $u \in U$ then fix(u) is a semi-domain and least upper bounds in U and fix(u) coincide. If u is strict (i.e. $u(\perp)=\perp$) the fix(u) is a sub semi-domain of U.

Proof

Let $Z \subseteq \text{fix}(u)$ be directed then $u(\bigsqcup Z)=\bigsqcup_{z\in Z} u(z)=\bigsqcup_{z\in Z} z=\bigsqcup Z$
so $\bigsqcup Z \in \text{fix}(u)$ and clearly it is the least upper bound (in both U and fix(u)) of Z.
If u is strict then $\perp \in \text{fix}(u)$ and so fix(u) is a sub semi-domain of U.

Q.E.D.

If we think of applying u to x as 'pushing' x into its 'best approximation' in fix(u) then we would intuit $u(x) \in$ ~~[illegible]~~ fix(u), i.e. $\forall x. u(u(x))=u(x)$ or $u=u\circ u$. Now this way of thinking works wonders (I cannot yet quite put my finger on why - it still seems magic to me!) so I shall always use u's such that $u=u\circ u$ to represent semi-domains. Hence the next definition.

6.6.4 Definition

$u \in U$ is a retraction $\Leftrightarrow u=u\circ u$.
If a semi-domain D is isomorphic to fix(u) for some retraction u then D is a retract of U.

6.6.5/

6.6.5 Lemma

$\{u \mid u$ is a retraction$\}$ is a sub semi-domain of U.

Proof

$\{u \mid u$ is a retraction$\}=\mathrm{fix}(\lambda u.u\circ u)$. And since $\perp=\perp\circ\perp$ this is a sub semi-domain of U by lemma 6.6.3.

Q.E.D.

In view of lemma 6.6.3 it would be nice if we could always assume that the retractions we use to represent semi-domains are strict. This can be done without loss of generality since if we make the following definition

6.6.6 Definition

If $u \in U$ then define $\underline{u}$ by:

$$\underline{u}(x)=\begin{cases}\perp & \text{if } u(x)=u(\perp)\\ u(x) & \text{otherwise.}\end{cases}$$

Then we have:

6.6.7/

6.6.7 Lemma

1. $u \in U \Rightarrow \underline{u} \in U$

2. u a retraction $\Rightarrow$ $\underline{u}$ a strict retraction.

3. u a retraction $\Rightarrow$ $u = u \circ \underline{u} \circ u$ and $\underline{u} = \underline{u} \circ u \circ \underline{u}$

4. u a retraction $\Rightarrow$ $\mathrm{fix}(u) \cong \mathrm{fix}(\underline{u})$

Proof

1: I need to show $\underline{u}$ is continuous. Let $Z \subseteq U$ be directed.

1.1:

If $u(\bigsqcup Z) = u(\bot)$ then $\bigsqcup_{z \in Z} u(z) = u(\bot)$ so $\forall z \in Z. u(z) \sqsubseteq u(\bot)$ hence as $\bot \sqsubseteq z$ we have $\forall z \in Z. u(\bot) \sqsubseteq u(z) \sqsubseteq u(\bot)$ i.e. $u(z) = u(\bot)$. Hence $\underline{u}(\bigsqcup Z) = \bot = \bigsqcup_{z \in Z} \underline{u}(z)$

1.2:

If $u(\bigsqcup Z) \neq u(\bot)$ then for some $z \in Z. u(z) \neq u(\bot)$ so $Z^+ = \{z \in Z \mid u(z) \neq u(\bot)\}$ is non-empty. As Z is directed so is Z^+ for if $x, y \in Z^+$ then $x \sqsubseteq z, y \sqsubseteq z$ for some $z \in Z$, but then $z \in Z^+$ (otherwise $u(x), u(y) \sqsubseteq u(z) = u(\bot)$ so $x, y \notin Z^+$). Thus

$\underline{u}(\bigsqcup Z) = u(\bigsqcup Z) = \bigsqcup_{z \in Z} u(z) = \bigsqcup_{z \in Z^+} u(z) = \bigsqcup_{z \in Z^+} \underline{u}(z) = \bigsqcup_{z \in Z} \underline{u}(z)$

1.1 and 1.2 show $\underline{u}$ is continuous and so is in U.

2: Suppose $u = u \circ u$ then

$$\underline{u}(\underline{u}(x)) = \begin{cases} \underline{u}(\bot) = \bot = \underline{u}(x) & \text{if } u(x) = u(\bot) \\ \\ \underline{u}(u(x)) = u(u(x)) = u(x) = \underline{u}(x) & \text{if } u(x) \neq u(\bot) \end{cases}$$

so/

so $\underline{u}$ is a retraction, it is clearly strict.

3: Suppose $u=u\circ u$ then

3.1:

If $u(x)=u(\perp)$ then $u(u(x))=u(u(\perp))=u(\perp)$ so

$u(\underline{u}(u(x)))=u(\perp)=u(x)$

$\underline{u}(u(\underline{u}(x)))=\underline{u}(u(\perp))=\perp=\underline{u}(x)$

3.2:

If $u(x)\neq u(\perp)$ then $u(u(x))=u(x)\neq u(\perp)$ so

$u(\underline{u}(u(x)))=u(u(u(x)))=u(x)$

$\underline{u}(u(\underline{u}(x)))=\underline{u}(u(u(x)))=\underline{u}(u(x))=u(u(x))=u(x)=\underline{u}(x)$

4: By 3 above $\underline{u}$:fix(u)->fix($\underline{u}$),u:fix($\underline{u}$)->fix(u) are isomorphisms.

Q.E.D.

This lemma shows that any retract of U is isomorphic to fix(u) where u is a strict retraction, by lemma 6.6.3. It follows that any retract of u is isomorphic to a sub semi-domain of u.

6.6.8 Definition

If A is a semi-domain then $a \in U$ represents A <=>

(1) a is a strict retraction

(2) $A \cong$ fix(a)

Remark: Any retract of U can be represented by an element of U.

Given/

Given elements $a,b \in U$ representing A,B what elements represent $[A\to B], A\times B, A\otimes B, A+B, A\oplus B$...etc. The answer to this is given in definition 6.6.10 below, first another definition.

6.6.9 Definition

If A is a semi-domain then define the semi-domain A^+ by

$$A^+ = A \,\dot{\cup}\, \{\perp\} \quad (\dot{\cup} = \text{disjoint union})$$

and for $x,y \in A^+$. $x \sqsubseteq y \iff x=\perp$ or $x,y \in A$ and $x \sqsubseteq_A y$.

Remark: The element $\perp_A \in A^+$ is not $\perp_{A^+}$ but is just 'above' it.

6.6.10 Definition

For $a,b \in U$ define:

6.6.10.1 $a\to b = \lambda u.\, b\circ u\circ a$

6.6.10.2 $a\times b = \lambda u.(a(fst(u)),b(snd(u)))$

6.6.10.3 $a\otimes b = \lambda u.\langle a(fst(u)),b(snd(u))\rangle$

6.6.10.4 $a+b = \lambda u.(fst(u)\supset (tt,a(snd(u))),(ff,b(snd(u))))$

6.6.10.5 $a\oplus b = \lambda u.(fst(u)\supset \langle tt,a(snd(u))\rangle,\langle ff,b(snd(u))\rangle)$

6.6.10.6 $a^+ = \lambda u.(fst(u)\supset (tt,a(snd(u))),\perp)$

6.6.11 Lemma

If a,b are strict retractions then so are $a\to b$, $a\times b$, $a\otimes b$, $a+b$, $a\oplus b$, a^+ and:

6.6.11.1/

<u>6.6.11.1</u> $f \in a\text{->}b \iff f\circ a = b$ and $\forall x \in a.\ f(x) \in b$

<u>6.6.11.2</u> $x \in a\times b \iff x=(u,v),\ u \in a,\ v \in b$

<u>6.6.11.3</u> $x \in a\otimes b \iff x=\perp$ or $x=(u,v),\ \perp\neq u \in a,\ \perp\neq v \in b$

<u>6.6.11.4</u> $x \in a+b \iff x=\perp$ or $x=(tt,u), u \in a$ or $x=(ff,v),\ v \in b$

<u>6.6.11.5</u> $x \in a\oplus b \iff x=\perp$ or $x=(tt,u),\ \perp\neq u \in a$ or $x=(ff,v),\ \perp\neq v \in b$

<u>6.6.11.6</u> $x \in a^{+} \iff x=\perp$ or $x=(tt,u),\ u \in a$

<u>Proof</u>

If u is a retraction then $x \in u \iff x=u(x)$, the lemma follows from this by tedious, but utterly straightforward, calculation. I omit the details.

Q.E.D.

6.6.12 <u>Lemma</u>

Suppose a represents A and b represents B then:

<u>6.6.12.1</u>	a->b	represents	[A->B]
<u>6.6.12.2</u>	$a\times b$	"	$A\times B$
<u>6.6.12.3</u>	$a\otimes b$	"	$A\otimes B$
<u>6.6.12.4</u>	$a+b$	"	$A+B$
<u>6.6.12.5</u>	$a\oplus b$	"	$A\oplus B$
<u>6.6.12.6</u>	a^{+}	"	A^{+}

<u>Proof/</u>

Proof

Follows directly from the previous lemma.

Q.E.D.

I still have to represent Id, S, S* and Env in U. To represent Id and S notice that they are isomorphic to the integers N where

$N=\text{flat}(\{0,1,2,\ldots\})$

Remark: Previously I used N for the set $\{0,1,2,\ldots\}$, context should make clear when I use N for this and when I use it for $\text{flat}(\{0,1,2,\ldots\})$.

6.6.13 Definition

Let $\underline{0}=(tt,tt)$

$\underline{n+1}=(ff,\underline{n})$

The next lemma shows that $\{\perp,\underline{0},\underline{1},\underline{2},\ldots\}\cong N$.

6.6.14 Lemma

$\underline{n} \sqsubseteq \underline{m} \Leftrightarrow n=m$

Proof

Clearly $n=m \Rightarrow \underline{n} \sqsubseteq \underline{m}$ to see the converse we have:

$$\begin{aligned}\underline{n}\sqsubseteq\underline{m} &\Rightarrow \text{snd}(\underline{n}) \sqsubseteq \text{snd}(\underline{m})\\ &\Rightarrow \underline{n-1} \sqsubseteq \underline{m-1}\\ &\vdots\\ &\not\Rightarrow\end{aligned}$$

$\Rightarrow \underline{0} \sqsubseteq \underline{m-n}$

$\Rightarrow \underline{0} = \underline{m-n}$ (otherwise $tt=fst(\underline{0})=fst(\underline{m-n})=ff$)

$\Rightarrow \underline{0+n} = \underline{m-n+n}$

$\Rightarrow \underline{n} = \underline{m}$

Q.E.D.

6.6.15 Definition

Let $N=Y(\lambda u.\underline{0}\oplus u) \in U$

Remark: This use of N will be distinguished from the others by context.

By the fixed point property of Y we have:

$N=\underline{0}\oplus N$

By lemma 6.6.5 and 6.6.11 $N=\bigsqcup_{n}(\lambda u.\underline{0}\oplus u)^n(\perp)$ is a strict retraction.

6.6.16 Lemma

$x \in N \Leftrightarrow x \in \{\perp,\underline{0},\underline{1},\ldots\}$

Proof

$x \in N \Leftrightarrow x=N(x)=Y(\lambda u.\underline{0}\oplus u)(x)$

$=\bigsqcup_{n=0}(\lambda u.\underline{0}\oplus u)^n(\perp)(x)$

$=\bigsqcup_{n=0}N_{(n)}(x)$ where $N_{(n)}=(\lambda u.\underline{0}\oplus u)^n(\perp)$

I/

I show by induction on n that $x \in N_{(n)} \Leftrightarrow x \in \{\perp, \underline{0}, \ldots, \underline{n-1}\}$

$n=0$:

$x \in N_{(0)} \Leftrightarrow x=\perp$

Assume $x \in N_{(n)} \Rightarrow x \in \{\perp, \underline{0}, \ldots, \underline{n-1}\}$ then:

$x \in N_{n+1} \Leftrightarrow x=N_{(n+1)}(x)$

$\Leftrightarrow x=(\underline{0}\oplus N_{(n)})(x)$

$\Leftrightarrow x=\perp$ or $x=(tt,u)$, $\perp\neq u \in \underline{0}$ or $x=(ff,v)$, $\perp\neq v \in N_{(n)}$

(by 6.6.11.5)

$\Leftrightarrow x=\perp$ or $x=\underline{0}$ or $x \in \{\underline{1}, \ldots, \underline{n}\}$

(by induction and fact that $u \in \underline{0} \Leftrightarrow x \in \{\perp, tt\}$)

$\Leftrightarrow x \in \{\perp, \underline{0}, \ldots, \underline{n}\}$

To finish the proof I shall show $x \in N \Leftrightarrow \forall n.x \in N_{(n)}$. This is clearly true if $x=\perp$ so suppose $\perp\neq x \in N$ then $x=\bigsqcup_{n} N_{(n)}(x)$ so $\exists$ n such that $N_{(n)}(x)\neq\perp$, but then $N_{(n)}(x) \in \{\underline{0}, \ldots, \underline{n-1}\}$; now for $m\geq n$ we have $N_{(n)}(x) \sqsubseteq N_{(m)}(x)$ and $\perp\neq N_{(m)}(x) \in N_{(m)}=\{\perp, \underline{0}, \ldots, \underline{m-1}\}$ so by lemma 6.6.14 $N_{(n)}(x)=N_{(m)}(x)$ and hence:

$$x=\bigsqcup_{m} N_{(m)}(x)=N_{(n)}(x) \in \{\underline{0}, \ldots, \underline{n-1}\}$$

Conversely suppose $\perp\neq x \in N_{(n)}$ then $x=\underline{m}$ for some m ($<n$)

Now $N(\underline{0}) =(\underline{0}\oplus N)(\underline{0})$ (as $N=\underline{0}\oplus N$)

$=(\underline{0}\oplus N)((tt,tt))$ (as $\underline{0}=(tt,tt)$)

$=\langle tt,\underline{0}(tt)\rangle$ (by 6.6.10.5)

$=(tt,tt)$ (by 6.6.1.5 and $tt=\underline{0}(tt)\neq\perp$)

$=\underline{0}$

so $\underline{0} \in N$

Also/

Also if $\underline{n} \in N$ then $N(\underline{n+1})=(0 \oplus N)((ff,\underline{n}))$ (by 6.6.13)

$=\langle ff,N(\underline{n})\rangle$ (by 6.6.10.5)

$=(ff,\underline{n})$ (by 6.6.1.5 and $\underline{n} \in N$)

$=\underline{n+1}$ (by 6.6.13)

so $\underline{n+1} \in N$

hence by induction $\forall n\geq 0.\underline{n} \in N$. So in particular $x=\underline{m} \in N$.

Q.E.D.

6.6.17 Lemma

There are elements id, s $\in U$ which represent Id, S respectively.

Proof

Just take disjoint copies of N e.g. let id=$\underline{tt \times N}$

s=$\underline{ff \times N}$

(these work by 6.6.7 and 6.6.11.2)

Q.E.D.

I shall now show how to represent A* given a representation of A.

6.6.18 Definition

Define inductively:

$()=\underline{0}$, $(x_1,\ldots,x_n)=(ff,(x_1,(x_2,\ldots,x_n)))$

$\langle\rangle=\underline{0}$, $\langle x_1,\ldots,x_n\rangle=\langle ff,\langle x_1,\langle x_2,\ldots,x_n\rangle\rangle\rangle$

Remark/

Remark: When n=2 this definition clashes with definition 6.6.1, context should always disambiguate things.

6.6.19 <u>Lemma</u>

$(x_1,\dots,x_n) \sqsubseteq (y_1,\dots,y_m) \Leftrightarrow n=m$ and $\forall\, \underline{i \leq} n . x_i \sqsubseteq y_i$

$\langle x_1,\dots,x_n\rangle \sqsubseteq \langle y_1,\dots,y_m\rangle \Leftrightarrow \exists\, i . x_i = \perp$ or ($n=m$ and $\forall\, \underline{i \leq} n . x_i \sqsubseteq y_i$)

<u>Proof</u>

Clearly $(\forall\, \underline{i \leq} n . x_i \sqsubseteq y_i) \Rightarrow (x_1,\dots,x_n) \sqsubseteq (y_1,\dots,y_n)$

Now suppose $(x_1,\dots,x_n) \sqsubseteq (y_1,\dots,y_m)$.

If $n<m$ then $(x_1,\dots,x_n) \sqsubseteq (y_1,\dots,y_m) \Rightarrow (x_2,\dots,x_n) \sqsubseteq (y_2,\dots,y_m)$

$$\vdots$$

$$\Rightarrow (x_n) \sqsubseteq (y_n,\dots,y_m)$$

$$\Rightarrow \underline{0} \sqsubseteq (y_{n+1},\dots,y_m)$$

$$\Rightarrow tt \sqsubseteq ff$$

which is impossible.

Similarly if $n>m$ then $(x_1,\dots,x_n) \sqsubseteq (y_1,\dots,y_m) \Rightarrow ff \sqsubseteq tt$ which is also impossible hence $n=m$.

Then $(x_1,\dots,x_n) \sqsubseteq (y_1,\dots,y_n) \Rightarrow x_1 \sqsubseteq y_1$ and $(x_2,\dots,x_n) \sqsubseteq (y_2,\dots,y_n)$

$$\vdots$$

$$\Rightarrow x_1 \sqsubseteq y_1,\dots,x_n \sqsubseteq y_n$$

The result for $\langle x_1,\dots,x_n\rangle$ follows since if $x_i = \perp$ for some i then

$\langle x_1$/

$\langle x_1,\dots,x_n\rangle=\perp$ otherwise $\langle x_1,\dots,x_n\rangle=(x_1,\dots,x_n)$ (and similarly for $\langle y_1,\dots,y_m\rangle$).

Q.E.D.

6.6.20 Definition

For $a \in U$ define: $a^{\circledast}=Y(\lambda u.\underline{0}\oplus(a\otimes u))$

Remark: $a^{\circledast}=\underline{0}\oplus(a\otimes a^{\circledast})$

6.6.21 Lemma

If a is a strict retraction then so is $a^{\circledast}$ and

$$x \in a^{\circledast} \Leftrightarrow x=\perp \text{ or } x=() \text{ or } x=(x_1,\dots,x_n),\ \perp\neq x_i \in a$$

Proof

$$x \in a^{\circledast} \Rightarrow x=a^{\circledast}(x)=\bigsqcup_n (\lambda u.\underline{0}\oplus(a\otimes u))^n(\perp)(x)$$
$$=\bigsqcup_n a^{\circledast}_{(n)}(x) \quad \text{where } a^{\circledast}_{(n)}=(\lambda u.\underline{0}\oplus(a\otimes u))^n(\perp)$$

By lemmas 6.6.5 and 6.6.11 $a^{\circledast}$ is a strict retraction.

Now clearly $x \in a^{\circledast}_{(0)} \Leftrightarrow x=\perp$, I show by induction on n that:

$$x \in a^{\circledast}_{(n+1)} \Leftrightarrow x=\perp \text{ or } x=() \text{ or } x=\{(x_1,\dots,x_m) \mid \perp\neq x_i \in a,\ \underline{1\leq i\leq m\leq n}\}$$

n=0:/

n=0:

$x \in a^{\circledast}{}_{(1)}$

$\Leftrightarrow x=(\underline{0}\oplus(a\otimes a^{\circledast}{}_{(0)}))(x)$

$\Leftrightarrow x=\perp$ or $x=(tt,u)$, $\perp\neq u \in \underline{0}$ or $x=(ff,v)$, $\perp\neq v \in (a\otimes a^{\circledast}{}_{(0)})$

(by 6.6.11.5)

$\Leftrightarrow x=\perp$ or $x=()$ or $x=(ff,(w,y))$, $\perp\neq w \in a$, $\perp\neq y \in a^{\circledast}{}_{(0)}$

(by 6.6.11.3)

$\Leftrightarrow x=\perp$ or $x=()$ (as $\perp\neq y \in a^{\circledast}{}_{(0)}$ is impossible)

Now assume true for n, I show it's true for (n+1):

$x \in a^{\circledast}{}_{(n+1)}$

$\Leftrightarrow x=\perp$ or $x=()$ or $x=(ff,(w,y))$, $\perp\neq w\in a$, $\perp\neq y \in a^{\circledast}{}_{(n)}$

(by a calculation like that above)

$\Leftrightarrow x=\perp$ or $x=()$ or $x=(ff,(w,()))$, $\perp\neq w \in a$

or $x \in \{(ff,(w,(x_1,\ldots,x_m))) \mid \perp\neq w \in a,\ \perp\neq x_i \in a,\ 1\leq i\leq m\leq n\}$

(by induction)

$\Leftrightarrow x=\perp$ or $x=()$ or $x \in \{(x_1,\ldots,x_n) \mid \perp\neq x_i \in a,\ 1\leq i\leq m\leq n+1\}$

To complete the proof I show that:

$$x \in a^{\circledast} \Leftrightarrow \exists n . x \in a^{\circledast}{}_{(n)}$$

This is clearly true for $x=\perp$, suppose $\perp\neq x \in a^{\circledast}$ then

$x=a^{\circledast}/$

$x=a^{\circledast}(x)=\bigsqcup_n a^{\circledast}_{(n)}(x)$. Then there exists an $n>0$ such that $\perp\neq a^{\circledast}_{(n)}(x)\in a^{\circledast}_{(n)}$ and by lemma 6.6.19 $\forall\, \underline{m\geq n}.a^{\circledast}_{(n)}(x)=a^{\circledast}_{(m)}(x)$ so $x=a^{\circledast}_{(n)}(x)\in a^{\circledast}_{(n)}$.

Conversely suppose $\perp\neq x\in a^{\circledast}_{(n)}$ then $x=(x_1,\ldots,x_m)$ $(\underline{m\leq n})$, I show $x\in a^{\circledast}$ by induction on m:

m=0:

$$\begin{aligned}x=():a^{\circledast}(())&=a^{\circledast}(\underline{0})\\&=(\underline{0}\oplus(a\otimes a^{\circledast}))(\underline{0})\\&=(tt,tt)\\&=\underline{0}\\&=()\end{aligned}$$

Suppose true for m I show it is true for m+1.

$$\begin{aligned}a^{\circledast}((x_1,\ldots,x_{m+1}))&=(\underline{0}\oplus(a\otimes a^{\circledast}))((ff,(x_1,(x_2,\ldots,x_{m+1})))) && \text{(by 6.6.18)}\\&=(ff,a\otimes a^{\circledast}((x_1,(x_2,\ldots,x_{m+1})))) && \text{(by 6.6.10.5)}\\&=(ff,(a(x_1),a^{\circledast}((x_2,\ldots,x_{m+1})))) && \text{(by 6.6.10.3)}\\&=(ff,(x_1,(x_2,\ldots,x_{m+1}))) && \text{(by induction and } x_1\in a)\\&=(x_1,\ldots,x_{m+1}) && \text{(by 6.6.18)}\end{aligned}$$

Hence $\forall\, \underline{m\geq 0}.(x_1,\ldots,x_m)\in a^{\circledast}$ (if $\forall\, \underline{i\leq m}.x_i\in a$)

Q.E.D.

6.6.22/

6.6.22 <u>Definition</u>

For $a \in U$ let $a^{*}=(a^{+})^{\circledast}$

6.6.23 <u>Lemma</u>

If a represents A then a^{*} represents A^{*}

<u>Proof</u>

By lemma 6.6.21:

$$x \in a^{*} \Leftrightarrow x=\perp \text{ or } x=() \text{ or } x=(x_1,\ldots,x_n),\quad \perp \neq x_i \in a^{+}$$

Now $A \cong \{x \mid \perp \neq x \in A^{+}\}$ and so

$$A^{n}=\{(x_1,\ldots,x_n) \mid \perp \neq x_i \in A^{+}\}$$

The result follows from 6.6.19.

Q.E.D.

6.6.24 <u>Definition</u>

Let s represent S (as in 6.6.17) then define:

fun=s*->s

d=s+fun

6.6.25/

6.6.25 Lemma

fun represents FUN=[S*->S]

d represents D=S+FUN

Proof

By lemmas 6.6.12, 6.6.17 and 6.6.23.

Q.E.D.

It is now easy to represent the alist model Env, full details follow:

6.7 Representing Env in universal space

6.7.1 Definition

Let env=Y(λe.id->(e->d))

env=id->(env->d), so env represents a semi-domain Env which satisfies Env$\cong$[Id->[Env->D]] (i.e. satisfies Env1). Thus by proposition 5.2.5 this Env contains an alist model as a sub semi-domain and so alist models exist. In fact Env itself is an alist model as the next lemma proves.

6.7.2 Lemma

env represents an alist model.

Proof/

Proof

$\mathrm{env}=Y(\lambda e.\mathrm{id}\text{->}(\mathrm{env}\text{->}d))=\bigsqcup_{n=0}\mathrm{env}_{(n)}$ where $\mathrm{env}_{(n)}=(\lambda e.\mathrm{id}\text{->}(e\text{->}d))^n(\perp)$

6.7.2.1 Lemma

$\forall n\geq 0.\mathrm{env}_{(n)}\circ\mathrm{env}=\mathrm{env}\circ\mathrm{env}_{(n)}=\mathrm{env}_{(n)}$

Proof

Induction on n:

n=0: $\mathrm{env}_{(0)}=\perp$ so $\mathrm{env}_{(0)}\circ\mathrm{env}=\mathrm{env}\circ\mathrm{env}_{(0)}=\perp$

Now assume true for n then:

$(\mathrm{env}_{(n+1)}\circ\mathrm{env})(x)=\mathrm{env}_{(n+1)}(\mathrm{env}(x))$

$=(\mathrm{env}_{(n)}\text{->}d)\circ\mathrm{env}(x)\circ\mathrm{id}$ (by 6.6.10.1)

$=(\mathrm{env}_{(n)}\text{->}d)\circ(\mathrm{env}\text{->}d)\circ x\circ\mathrm{id}\circ\mathrm{id}$ (by 6.6.10.1)

$=(\mathrm{env}_{(n)}\text{->}d)\circ(\mathrm{env}\text{->}d)\circ x\circ\mathrm{id}$ (as id is a retraction)

now $(\mathrm{env}_{(n)}\text{->}d)\circ(\mathrm{env}\text{->}d)=\lambda u.d\circ((\mathrm{env}\text{->}d)(u))\circ\mathrm{env}_{(n)}$ (by extensionality and 6.6.10.1)

$=\lambda u.d\circ d\circ u\circ\mathrm{env}\circ\mathrm{env}_{(n)}$

$=\lambda u.d\circ u\circ\mathrm{env}_{(n)}$ (by induction)

$=(\mathrm{env}_{(n)}\text{->}d)$ (by extensionality and 6.6.10.1)

so/

so $(env_{(n+1)}\circ env)(x) = (env_{(n)}\text{->}d)\circ x\circ id$

$= (id\text{->}(env_{(n)}\text{->}d))(x)$ (by 6.6.10.1)

$= env_{(n+1)}(x)$ (by 6.6.10.1)

similarly $(env\circ env_{(n+1)})(x)=(env\text{->}d)\circ(env_{(n)}\text{->}d)\circ x\circ id$

and $(env\text{->}d)\circ(env_{(n)}\text{->}d)= \lambda u.d\circ d\circ u\circ env_{(n)}\circ env$

$= \lambda u.d\circ u\circ env_{(n)}$

$=env_{(n)}\text{->}d$

so $env\circ env_{(n+1)}(x)=(env_{(n)}\text{->}d)\circ x\circ id$

$=env_{(n+1)}(x)$

Q.E.D.

Thus if for $r \in env$ we let $r_{(n)}=env_{(n)}(r)$ then $r_{(n)} \in env$.

I now show that $r \mapsto r_{(n)}$ satisfies Env2-Env5.

Env2: $\perp=env_{(0)} \sqsubseteq env_{(1)} \sqsubseteq \dots \sqsubseteq env_{(n)} \sqsubseteq \dots \sqsubseteq env$

hence $r \in env \Rightarrow \perp=r_{(0)} \sqsubseteq r_{(1)} \sqsubseteq \dots \sqsubseteq r_{(n)} \sqsubseteq \dots \sqsubseteq r$

Env3: $env= \bigsqcup_n env_{(n)}$

hence $r \in env \Rightarrow r=\bigsqcup_n r_{(n)}$

Env4: $r \in env \Rightarrow (r_{(n)})_{(n)}=env_{(n)}(env_{(n)}(r))$

$=env_{(n)}(r)$ (as $env_{(n)}$ is a retraction)

$=r_{(n)}$

Env5: $env_{(n+1)}(r)(x)(env(r'))=((env_{(n)}\text{->}d)\circ r\circ id)(x)(env(r'))$

$=(d\circ r(id(x))\circ env_{(n)})(env(r'))$

$=(d\circ r(id(x))\circ env_{(n)})(r')$ (by previous lemma)

also/

also $env(r)(x)(env_{(n)}(r'))$ $=((env\text{->}d)\circ r\circ id)(x)(env_{(n)}(r'))$

$=(d\circ r(id(x))\circ env)(env_{(n)}(r'))$

$=(d\circ r(id(x))\circ env_{(n)})(r')$ (by previous lemma)

hence $r,r' \in env, x \in id \Rightarrow r_{(n+1)}(x)(r')=r(x)(r'_{(n)})$

It follows that env represents an alist model.

6.8 BNF as recursion equations in universal space

I shall now show how the apparatus described in 6.6 can be used to 'solve' BNF syntax equations within the theory of semi-domains. The point of doing this is to make things more uniform so the same theory can be applied to syntax equations, semantic equations and recursive definitions. We would like to say that what gets defined by e.g.

a ::= NIL | (A/z)a | (fn/z)a

is Y(λa.NIL | (A/z)a | (fn/z)a)

To achieve this it is first necessary to make the primitive syntactic classes into semi-domains and then to interpret the BNF symbols ::=, | and concatenation in such a way that BNF definitions become recursion equations. A primitive syntactic class <prim> has naturally associated with it the flat semi-domain prim=flat (<prim>) and if we interpret ::=, | and concatenation as =, ⊕ and ⊗ respectively things work out. Thus the BNF

a/

$a ::= \mathrm{NIL} \mid (A/z)a \mid (fn/z)a$

becomes the recursive definition

$$\begin{aligned} a = \mathrm{NIL}\ &\oplus(\ulcorner(\urcorner \otimes \text{s-expression} \otimes \ulcorner/\urcorner \otimes \text{identifier} \otimes \ulcorner)\urcorner \otimes a) \\ &\oplus(\ulcorner(\urcorner \otimes \text{function} \otimes \ulcorner/\urcorner \otimes \text{identifier} \otimes \ulcorner)\urcorner \otimes a) \end{aligned}$$

where:

$\mathrm{NIL}=\mathrm{flat}(\{\mathrm{NIL}\})$

$\ulcorner(\urcorner=\mathrm{flat}(\{\text{"("}\})$

$\ulcorner/\urcorner=\mathrm{flat}(\{\text{"/"}\})$

$\ulcorner)\urcorner=\mathrm{flat}(\{\text{")"}\})$

s-expression=flat(<S-expression>)

identifier=flat(<identifier>)

function=flat(<function>)

using the methods of proof of lemmas 6.6.16 and 6.6.21 it is straightforward to show that this equation (when solved with Y) defines a so that

$$\begin{aligned} u \in a \iff u=\perp\ &\text{or } u=(\text{"("},A,\text{"/"},z,\text{")"},a) \text{ where } \perp \neq A \in \text{s-expression} \\ &\qquad \perp \neq z \in \text{identifier} \\ &\qquad \perp \neq a \in a \text{ (two different a's here!)} \\ &\text{or } u=(\text{"("},fn,\text{"/"},z,\text{")"},a) \text{ where } \perp \neq fn \in \text{function} \\ &\qquad \perp \neq z \in \text{identifier} \\ &\qquad \perp \neq a \in a \text{ (two different a's here!)} \end{aligned}$$

i.e./

i.e. fix(a)=flat(<alist>) so a represents flat(<alist>).
Of course the symbols "(","/",")" need not occur in the abstract syntax so we could have got a to represent flat(<alist>) by the neater definition:

$$a=NIL \oplus (\text{s-expression} \otimes \text{identifier} \otimes a) \oplus (\text{function} \otimes \text{identifier} \otimes a)$$

I put the extra symbols in just to show that doing it presents no problems. The reason that $\otimes$, $\oplus$ are used rather than $\times$,+ is to exclude infinite expressions (such as an a which satisfies a=(A/z)a) - this is essential if structural induction is to be a valid mode of reasoning. If we think of all BNF in this way then the semantic functions e $\mapsto$ $[\![e]\!]$ etc. become members of [form->[Env->D]] etc. (where form=flat(<form>)) and so the theory of continuous functions can be brought to bear on them (e.g. semantic equations can be solved with Y). Although this extra rigour does not seem useful in the cases discussed in this report, it can be, especially if one <u>does</u> want to admit infinite expressions (i.e. use +,$\times$) as well as finite ones (e.g. as in [21]). Such infinite programs cannot be defined in BNF but can be using recursive semi-domain equations, thus by regarding BNF in the above way we got both the benefit of its undoubted lucidity plus the ability to extend it to deal with infinite expressions when needed.

7. EXTENSIONS OF THE SEMANTICS TO MORE OF LISP 1.5

The semantic equations described so far only deal with pure LISP. Unfortunately very few interesting LISP programs lie in that subset. Until there is a fully debugged semantics of most of LISP 1.5 one cannot know for sure whether the approach adopted here is a dead end or the tip of a useful iceberg. In this chapter I shall describe semantic equations which purport to handle more of LISP 1.5 than just pure LISP; I have done no analysis of these equations and so their superficial plausibility may turn out to be ill-founded. I shall give a sequence of three sets of equations, each set handling more than its predecessor. I shall not describe the corresponding extensions of -> as it is fairly straightforward (I think!) to express the algorithm implicit in the extended eval function (given in appendix B of the LISP 1.5 Manual [6] in 'calculus form' - if this is not straightforward I do not know how to do it. I think it is important that -> be extended and compared with the semantic equations below so that one can know whether 'higher type' operational and denotational intuitions about LISP are consistent with one another. Chris Wadsworth has investigated an analogous problem for the λ-calculus [25] and Gordon Plotkin [16] has adapted that work to cope with a 'strictness test' approach to call-by-value in the λ-calculus.

7.1 Functional arguments (funargs)

Functional arguments or "funargs" are a very useful feature of LISP; they/

they (among other things) enable "canned loops" such as maplist to be defined. The obvious way to handle them semantically is to change D from S+[S*->S] to S+[D*->S]; I believe this works but care is needed as the examples below show. Consider $e=\lambda[[f];NIL][Label[g;g]]$, this should evaluate to NIL so the following calculation <u>must not</u> be valid:

$$[\![e]\!](r)=[\![\lambda[[f];NIL]]\!](r)([\![Label[g;g]]\!](r)) \qquad (1)$$

$$=(\underline{\lambda}f.NIL)(\perp) \qquad (2)$$

$$=\perp \qquad \text{(by definition of } \underline{\lambda} - 3.11.2)$$

However if $e'=\lambda[[f];NIL][Label[g;g][NIL]]$ we <u>do</u> want:

$$[\![e']\!](r)=[\![\lambda[[f];NIL]]\!](r)([\![Label[g;g]]\!](r)(NIL)) \qquad (1')$$

$$=(\underline{\lambda}f.NIL)(\perp) \qquad (2')$$

$$=\perp \qquad (3')$$

In the semantics below these examples come out right because I have arranged that only $\perp_S$ is identified with $\perp_D$ whilst $\perp_{FUN}\neq\perp_D$. To do this here is a definition

7.1.1 <u>Definition</u>

If D_1,D_2 are semi-domains and $D=D_1+D_2$ then for $x_i \in D_i$ $(i=1,2)$ define $(x_i\ \underline{in}\ D) \in D$ by:

$$(x_i \underline{\text{in}}\ D) = \begin{cases} \perp_D & \text{if } x = \perp_{D_i} \\ (x_i \text{ in } D) & \text{otherwise} \end{cases}$$

Remark: This definition should be compared with that of $(x_i \text{ in } D)$ viz. definition 3.7.9 - the difference is just that $\perp_D = (\perp_{D_i} \underline{\text{in}}\ D)$ but $\perp_D \neq \perp_{D_i} = (\perp_{D_i} \text{ in } D)$.

It is easy to see that $x \mapsto (x \underline{\text{in}}\ D)$ is continuous.

7.1.2 Semantics

Syntax

New metavariable: arg ranges over <argument>

Syntactic equations:

$$\begin{aligned} e &::= A \mid x \mid fn[arg_1;\ldots;arg_n] \mid [e_{11} \to e_{12};\ldots;e_{n1} \to e_{n2}] \\ fn &::= F \mid f \mid \lambda[[z_1;\ldots;z_n];e] \mid \text{Label}[f;fn] \mid \mu[f;fn] \\ arg &::= e \mid fn \end{aligned}$$

Semantics

Denotations $D = S + FUN$

$S = \text{flat}(\langle\text{S-expression}\rangle)$

$FUN = [D^* \to S]$ (see Note 1 below)

Environments: $Env = Id \to [Env \to D]$ - an alist model (5.2.1)

Semantic/

Semantic functions: $e \mapsto [\![e]\!]$: <form>->[Env->S]

$fn \mapsto [\![fn]\!]$: <function>->[Env->FUN]

$arg \mapsto A[\![arg]\!]$: <argument>->[Env->D]

Semantic equations:

(S1) $[\![A]\!](r)=A$

(S2) $[\![x]\!](r)=r(x)(r)\mid S$

(S3) $[\![fn[arg_1;\ldots;arg_n]]\!](r)=[\![fn]\!](r)([\![arg_1]\!](r),\ldots,[\![arg_n]\!](r))$

(S4) $[\![[e_{11}\to e_{12};\ldots;e_{n1}\to e_{n2}]]\!](r)=([\![e_{11}]\!](r)\to[\![e_{12}]\!](r),\ldots,[\![e_{n1}]\!](r)\to[\![e_{n2}]\!](r))$

(S5) $[\![car]\!](r)=\lambda t{:}D.car(t\mid S)$

$[\![cdr]\!](r)=\lambda t{:}D.cdr(t\mid S)$

$[\![cons]\!](r)=\lambda t_1 t_2{:}D.cons(t_1\mid S,t_2\mid S)$

$[\![atom]\!](r)=\lambda t{:}D.atom(t\mid S)$

$[\![eq]\!](r)=\lambda t_1 t_2{:}D,eq(t_1\mid S,t_2\mid S)$

(S6) $[\![f]\!](r)=r(f)(r)\mid FUN$

(S7) $[\![\lambda[[z_1;\ldots;z_n];e]]\!](r)=\lambda t_1\ldots t_n{:}D.[\![e]\!]((t_1/z_1)\ldots(t_n/z_n)r)$

(see Note 2 below)

(S8) $[\![Label[f;fn]]\!](r)=[\![fn]\!](([\![fn]\!]/f)r)$

(S9) $[\![\mu[f;fn]]\!](r)=Y(\lambda v{:}[Env\to FUN].\lambda r'.[\![fn]\!]((v/f)r'))(r)$

(S10) $A[\![e]\!](r)=([\![e]\!](r)$ <u>in</u> $D)$

(S11) $A[\![fn]\!](r)=([\![fn]\!](r)$ in $D)$

Note 1: Further specification of the solution of D=S+[D*->S] is required (cf. 5.2.1 and 6.2.1) - I have not investigated this further.

Note 2/

Note 2: If $t \in$ FUN, $z \in$ Id, $r \in$ Env then $(t/z)r$ means $((\lambda r'.(t \text{ in } D))/z)r$.

7.1.3 Example calculation of denotations

7.1.3.1: $fn = \mu[g;g]$

Then $[\![fn]\!](r) = [\![\mu[g;g]]\!](r)$

$= Y(\lambda v.\ \lambda r'.[\![g]\!]((v/g)r'))(r)$

$= Y(\lambda v.\ \lambda r'.v((v/g)r'))(r)$

$= \perp_{FUN}$ (as$(\lambda v.\ \lambda r'.v((v/g)r'))(\perp) = \lambda r'.\perp = \perp$)

7.1.3.2: $e = \lambda[[f];NIL][\mu[g;g]]$

Then $[\![e]\!](r) = [\![\lambda[[f];NIL]]\!](r)(A[\![\mu[g;g]]\!])$

$= (\underline{\lambda} t.NIL)(\perp_{FUN} \text{ in } D)$

$= NIL$

7.1.3.3: $e' = \lambda[[f];NIL][\mu[g;g][NIL]]$

Then $[\![e']\!](r) = [\![\lambda[[f];NIL]]\!](r)(A[\![\mu[g;g][NIL]]\!])$

$= (\underline{\lambda} t.NIL)([\![\mu[g;g][NIL]]\!] \ \underline{\text{in}}\ D)$

$= (\underline{\lambda} t.NIL)(\perp_S \ \underline{\text{in}}\ D)$

$= (\underline{\lambda} t.NIL)(\perp_D)$

$= \perp$

Thus assuming μ and Label are appropriately related, the examples mentioned above get handled correctly.

7.2/

7.2 Functional results

Having coped with functional arguments the obvious thing to do next is functional results, this presents no obvious difficulties - one just allows a function to be a form. Here are the details.

Syntax

$$e ::= A \mid x \mid fn \mid fn[e_1;\dots;e_n] \mid [e_{11}\rightarrow e_{12};\dots;e_{n1}\rightarrow e_{n2}]$$
$$fn ::= F \mid f \mid \lambda[[z_1;\dots;z_n];e] \mid \text{Label}[f;fn] \mid \mu[f;fn]$$

Semantics

Denotations: D=S+FUN

S=flat(<S-expression>)

FUN=[D*->D] (see Note 1 below)

Environments: Env=Id->[Env->D] - an alist model (5.2.1)

Semantic functions: $e \mapsto V[\![e]\!]$: <form>->[Env->D]

$fn \mapsto W[\![fn]\!]$: <function>->[Env->FUN]

Semantic equations:

(S1) $V[\![A]\!](r)=(A \text{ in } D)$

(S2) $V[\![x]\!](r)=r(x)(r)$

(S3) $V[\![fn]\!](r)=(W[\![fn]\!](r) \text{ in } D)$

(S4) $V[\![fn[e_1;\dots;e_n]]\!](r)=W[\![fn]\!](r)(V[\![e_1]\!](r),\dots,V[\![e_n]\!](r))$

(S5) $V[\![[e_{11}\rightarrow e_{12};\dots;e_{n1}\rightarrow e_{n2}]]\!](r)=(V[\![e_{11}]\!](r)\rightarrow V[\![e_{12}]\!](r),\dots,V[\![e_{n1}]\!](r)\rightarrow$

(see Note 2 below)

(S6)/

(S6) $$W[\![car]\!](r)= \underline{\lambda} t:D.(car(t \mid S)\ \underline{in}\ D)$$

$$W[\![cdr]\!](r)= \underline{\lambda} t:D.(cdr(t \mid S)\ \underline{in}\ D)$$

$$W[\![cons]\!](r)= \underline{\lambda} t_1 t_2:D.(cons(t_1 \mid S, t_2 \mid S)\ \underline{in}\ D)$$

$$W[\![atom]\!](r)= \underline{\lambda} t:D.(atom(t \mid S)\ \underline{in}\ D)$$

$$W[\![eq]\!](r)= \underline{\lambda} t_1 t_2:D.(eq(t_1 \mid S, t_2 \mid S)\ \underline{in}\ D)$$

(S7) $$W[\![f]\!](r)=r(f)(r) \mid FUN$$

(S8) $$W[\![\lambda[[z_1;\ldots;z_n];e]]\!](r)= \underline{\lambda} t_1\ldots t_n:D.V[\![e]\!]((t_1/z_1)\ldots(t_n/z_n)r$$

(see Note 3 below)

(S9) $$W[\![Label[f;fn]]\!](r)=W[\![fn]\!]((W[\![fn]\!]/f)r)$$

(S10) $$W[\![\mu[f;fn]]\!](r)=Y(\lambda v:[Env\to FUN].\ \lambda r'.W[\![fn]\!]((v/f)r'$$

Note 1: Further specification of the solution of $D=S+[D^*\to D]$ is required, again I have not investigated this further.

Note 2: For $t_{ij} \in D$ $(t_{11}\to t_{12};\ldots;t_{n1}\to t_{n2})$ means $((t_{11} \mid S\to t_{12} \mid S,\ldots,t_{n1} \mid S\to t_{n2} \mid S)\ \underline{in}\ S)$

Note 3: If $t \in FUN$ then (t/z) means $((\lambda r'.(t\ in\ D))/z)r$.

7.3 QUOTE'd arguments

Up to now (i.e. in 7.1, 7.2) I have only considered functional arguments formed with FUNCTION (i.e. closures). An important and useful property of LISP 1.5 is that one may keep the free variables of functional arguments fluid to allow the meaning of a function to depend on its activation environment as well as its definition environment (see [13]). To cope with this in the semantics is straightforward. *

Syntax/

Syntax

$e ::= A \mid x \mid fn \mid fn[arg_1;\dots;arg_n] \mid [e_{11} \to e_{12};\dots;e_{n1} \to e_{n2}]$

$fn ::= F \mid f \mid \lambda[[z_1;\dots;z_n];e] \mid Label[f;fn] \mid \mu[f;fn]$

$arg ::= e \mid QUOTE(fn)$

Semantics

Denotations: D=S+FUN

S=flat(<S-expression>)

FUN=[[Env->D]*->D] (see Note 1)

Environment: Env=Id->[Env->D] - an alist model (5.2.1)

Semantic functions: $e \mapsto V[\![e]\!]$: <form>->[Env->D]

$fn \mapsto W[\![e]\!]$: <function>->[Env->FUN]

$arg \mapsto A[\![arg]\!]$: <argument>->[Env->[Env->D]]

Semantic equations:

(S1) $V[\![A]\!](r)=(A \text{ in } D)$

(S2) $V[\![x]\!](r)=r(x)(r)$

(S3) $V[\![fn]\!](r)=(W[\![fn]\!](r) \text{ in } D)$

(S4) $V[\![fn[arg_1;\dots;arg_n]]\!](r)=W[\![fn]\!](r)(A[\![arg_1]\!](r),\dots,A[\![arg_n]\!](r))$

(S5) $V[\![[e_{11}\to e_{12};\dots;e_{n1}\to e_{n2}]]\!](r)=(V[\![e_{11}]\!](r)\to V[\![e_{12}]\!](r),\dots,V[\![e_{n1}]\!](r)\to V[\![e_{n2}]\!](r)$

(see Note 2)

(S6)/

(S6) $W\llbracket car\rrbracket(r)=\underline{\lambda}t:[Env\text{->}D].(car(t(r)\mid S)\ \underline{in}\ D)$

$W\llbracket cdr\rrbracket(r)=\underline{\lambda}t:[Env\text{->}D].(cdr(t(r)\mid S)\ \underline{in}\ D)$

$W\llbracket cons\rrbracket(r)=\underline{\lambda}t_1t_2:[Env\text{->}D].(cons(t_1(r)\mid S,t_2(r)\mid S)\ \underline{in}\ D)$

$W\llbracket atom\rrbracket(r)=\underline{\lambda}t:[Env\text{->}D].(atom(t(r)\mid S)\ \underline{in}\ D)$

$W\llbracket eq\rrbracket(r)=\underline{\lambda}t_1t_2:[Env\text{->}D].(eq(t_1(r)\mid S,t_2(r)\mid S)\ \underline{in}\ D)$

(S7) $W\llbracket f\rrbracket(r)=r(f)(r)\mid FUN$

(S8) $W\llbracket\lambda[[z_1;\ldots;z_n];e]\rrbracket(r)=\underline{\lambda}t_1\ldots t_n:[Env\text{->}D].V\llbracket e\rrbracket((t_1/z_1)\ldots(t_n/x_n)r)$

(S9) $W\llbracket Label[f;fn]\rrbracket(r)=W\llbracket fn\rrbracket((W\llbracket fn\rrbracket/f)r)$

(S10) $W\llbracket\mu[f;fn]\rrbracket(r)=Y(\lambda v:[Env\text{->}FUN].\lambda r'.W\llbracket fn\rrbracket((v/f)r'))(r)$

(S11) $A\llbracket e\rrbracket(r)=\lambda r'.V\llbracket e\rrbracket(r)$

(S12) $A\llbracket QUOTE(fn)\rrbracket(r)=(W\llbracket fn\rrbracket\ in\ [Env\text{->}D])$

Note 1: I have not investigated the further specification of D.

Note 2: Here $(t_{11}\to t_{12};\ldots;t_{n1}\to t_{n2})$ means $((t_{11}\mid S\to t_{12}\mid S\ ,\ldots,t_{n1}\mid S\to t_{n2}\mid S)\ \underline{in}\ D)$

Here is an example to show the difference between funargs and QUOTEd arguments.

Let fn= $\lambda[[f];\lambda[[x];f[x]][NIL]]$ so that

$W\llbracket fn\rrbracket(r)=\underline{\lambda}t_1.V\llbracket\lambda[[x];f[x]][NIL]\rrbracket((t_1/f)r)$

$=\underline{\lambda}t_1.(\underline{\lambda}t_2.V\llbracket f[x]\rrbracket((t_2/x)(t_1/f)r))(\lambda r'.(NIL\ in\ D))$

$=\underline{\lambda}t_1.V\llbracket f[x]\rrbracket((NIL/x)(t_1/f)r)$ (by convention 3.11.4 - (1) -)

$=\underline{\lambda}t_1.t_1((NIL/x)(t_1/f)r)(\lambda r'.(NIL\ in\ D))$

Then/

Then (1) $$\begin{aligned} V\llbracket fn[\lambda[[y];x]]\rrbracket(r) &= W\llbracket fn\rrbracket(r)(A\llbracket\lambda[[y];x]\rrbracket(r)) \\ &= (\lambda r'.V\llbracket\lambda[[y];x]\rrbracket(r))(\ldots)(\lambda r'.(NIL \text{ in } D)) \\ &= (\underline{\lambda} t.V\llbracket x\rrbracket((t/y)r))(\lambda r'.(NIL \text{ in } D)) \\ &= V\llbracket x\rrbracket((NIL/y)r) \\ &= r(x)((NIL/y)r) \end{aligned}$$

<u>but</u> (2) $$\begin{aligned} & V\llbracket fn[QUOTE(\lambda[[y];x])]\rrbracket(r) \\ &= W\llbracket fn\rrbracket(r)(A\llbracket QUOTE(\lambda[[y];x])\rrbracket(r)) \\ &= W\llbracket\lambda[[y];x]\rrbracket((NIL/x)(t_1/f)r)(\lambda r'.(NIL \text{ in } D)) \\ &= V\llbracket x\rrbracket((NIL/y)(NIL/x)(t_1/f)r) \\ &= NIL \text{ (where } t_1=(W\llbracket\lambda[[y];x]\rrbracket \text{ in } D)) \end{aligned}$$

In (2) x gets looked up 'later' than in (1) as intended.

7.4 <u>Further extensions</u>

So far I have been able to avoid having to simulate sequencing mechanisms in the semantic equations. Call-by-value is essentially a sequential notion - in essence it consists in evaluating arguments <u>before</u> passing them, however, I have been able to cope because (in simple cases) all the conclusions that this sequencing entails can be deduced from just one of them - namely that $\llbracket\lambda[[x_1;\ldots;x_n];e]\rrbracket(r)(\ldots\perp\ldots)=\perp$ i.e. strictness. Unfortunately this simple approach (which, to my knowledge, first appeared in [2]) no longer works when evaluations can have side effects which record the time at which things occurred: in such/

such cases one has to build sequencing in. Fortunately though, doing this is well understood, and it is another achievement of Chris Wadsworth [24] (see [18] also) to show how, using "continuations", one can make semantic equations just 'operational enough' to cope with this and other sequential notions such as jumping.

Because I do not want to go into the theory of continuations I shall not describe how side effects (e.g. assignment) PROG's and GOTO's are handled - in any case I have done absolutely no work on the analysis of the resulting equations (which I have not even written down!).

Another practically useful facility in LISP is the ability to construct function definitions at run time and then interpret them. Doing this is very operational and I do now know how best to handle it with denotational semantics, however, even if an operational 'semantics' is essential this should still be doable within the theory of semi-domains (see [18]).

I hope that the extensions described in this chapter show that there is still lots to be done before we can conclude that Env=Id->[Env->D] is a useful equation for handling fluid variables.

8. SUGGESTIONS FOR FUTURE RESEARCH

During the course of this work many problems have come up which I have not investigated, in this section I shall list some of them. My future plans are such that I doubt if I shall do any work on them.

8.1 Analysis of more sophisticated semantic equations

The semantic equations given in chapter 7 need to be examined to see if they are in accord with reality (as defined by existing interpreters). I think that, given a suitable extension of ->, the main theorem should be extendable to cope with funargs, QUOTEd arguments and functional results. If the proof idea used here is to work for the extended semantics a more subtle set of approximants will be needed. This is because there are more possibilities for non-terminating computations. (e.g. $\lambda[[f];f[f]][\lambda[[f];f[f]]]$) so to render approximants terminating a denser sprinkling of indices is called for. I think one will have to combine the 'classical' kind of approximants used by Wadsworth [25], with the kind I have used; the 'classical' approximants being used to finitize those parts of computations which are really λ-calculus reductions, and my approximants finitizing those parts which manipulate alists in an essential way.

There is also a need to investigate the relation between Label and μ for these extended semantics, I suspect that the best way to do this might not be via a generalized LISP-induction, but rather using the techniques of Robert Milne [8].

8.2/

8.2 Fluid variables at higher type: a calculus

In order to help with the study of QUOTEd arguments I give below an extension of the λ-calculus which has them. The hope is that this calculus exhibits the central features (and difficulties) of fluid variables in a form which makes their investigation easier. There is a considerable danger, when studying such abstract calculi, that one will ignore difficult problems because one has left them behind in the abstracting process. There is some evidence that call-by-value suffered this fate: it is only recently, with the study of real languages, that the differences between the λ-calculus and higher order programming languages have come to light (see e.g. [18]). Nevertheless study of the λ-calculus did lead to extraordinary insights so I do not feel too irresponsible in describing the calculus below:

8.2.1 The Q - λ-calculus

Syntax

Metavariables:	x	ranges	over	<identifier>
	e	"	"	<exp>
	arg	"	"	<argument>

Syntax equations:

$$e ::= x \mid e(arg) \mid \lambda x.e$$

$$arg ::= e \mid Q(e)$$

Semantics/

Semantics

Denotations: $D=[Env\to D]\to D$

Environments: $Env=Id\to[Env\to D]$ $(Id=flat(\langle identifier\rangle))$

Semantic functions: $V: \langle exp\rangle\to[Env\to D]$

$A: \langle argument\rangle\to[Env\to[Env\to D]]$

Semantic equations:

(S1) $V[\![x]\!](r)=r(x)(r)$

(S2) $V[\![e(arg)]\!](r)=V[\![e]\!](r)(A[\![arg]\!](r))$

(S3) $V[\![\lambda x.e]\!](r)=\lambda t:[Env\to D].V[\![e]\!]((t/x)r)$

(S4) $A[\![e]\!](r)=\lambda r':Env.V[\![e]\!](r)$

(S5) $A[\![Q(e)]\!](r)=V[\![e]\!]$

Some questions which naturally arise about the $Q-\lambda$-calculus are:

1. If no expressions are quoted (i.e. of the form $Q(e)$) does the $Q-\lambda$-calculus essentially collapse to the ordinary λ-calculus?

2. What relation obtains between:

$V[\![(\lambda x.x)(Q(e))]\!](r)=V[\![x]\!]((V[\![e]\!]/x)r)=V[\![e]\!]((V[\![e]\!]/x)r)$
and $Y(\lambda v:[Env\to D].\lambda r':Env.V[\![e]\!]((v/x)r'))(r)$?

3. What does the paradoxical combinator

$Y=\lambda f.(\lambda y.f(y(y)))(\lambda y.f(y(y)))$

denote?

These/

These questions may be trivial or uninteresting - I have not given them much thought.

8.3 The true relation between μ and Label

I proved in 4.6 that for all r of the form $[\![a]\!]$

$$[\![\mu[f;fn]]\!](r)=[\![Label[f;fn]]\!](r)$$

It would be interesting to know more about the set of r's for which this is true. Is there some mathematically definable subset of Env which includes $\{[\![a]\!] \mid a \in \langle alist\rangle\}$ and whose members make the equation true? The answer to this may be found in Robert Milne's work - perhaps there exist 'self-referential' predicates such that the r's that satisfy these constitute the desired subset of Env?

My counterexample in 4.6 does not rule out the possibility that $[\![\mu[f;fn]]\!] \equiv [\![Label[f;fn]]\!]$, I suspect this may be true (though my intuition here is weak) and it would be nice to know if it is.

8.4 Axioms to disambiguate semi-domain equations

In section 5.2 I showed how the equation Env=Id->[Env->D] could be rendered unambiguous, by requiring Env to satisfy Env2-Env5. These axioms characterize the semi-domain corresponding to the minimal solution obtained by regarding Env=Id->[Env->D] as a retraction equation in U (see 6.7). An interesting problem is to work out how to/

to move uniformly from semi-domain equations in general to axioms which characterize that solution which is obtained by regarding the equations as being about retractions of U.

8.5 Rules of inference for LISP: LISP logic

LISP-induction is an inference rule which I found ul, it is not clear though that my formulation is as general or as convenient to use as possible. To investigate this lots of examples need to be examined. It would be interesting to develop a logic specially tailored for reasoning about LISP. To design the formulae of such a logic one needs to know the 'shapes' of the theorems that one wants to prove - this requires lots of field work. Is it convenient to develop such a logic within LCF [10] or LAMBDA [23]? Perhaps one should start ab initio to prevent preconceived ideas warping things?

8.6 Fixpoint treatment of simultaneous recursion definitions

As mentioned in 3.10 and 4.8 I have not investigated the solution, via Y, of sets of mutually recursing definitions (e.g. those of apply,eval, ...etc); I feel that theorem 4.6.4 should be extendable to cope with this though the details might get messy - perhaps category-theoretic notation would help here? It would be interesting if it did because it might help to show the rationality of the worship of algebraic theories and other high-power tools by the adherants of the category theory cult.

8.7/

8.7 When does the naive semantics work?

Presumably for suitably simple programs (e.g. with no free variables or non-terminating sub-expressions) the semantics of 3.8 would work. I thought for a time that proving this was trivial but I am not so pessimistic now and I think it raises interesting problems. One approach would be to develop a calculus appropriate to the semantics of 3.8, say define ->> analogous to ->, so that an analogue of the Main Theorem held for ->>. Then one could try and show by induction on the size of computation that

p->A <=> p->>A

Another approach would be to use Robert Milne's techniques, I rather suspect that in fact this will be the best way to do it.

Appendix 1

"It is vain to do with more what can be done with fewer"

William of Occam.

In this report I have worked entirely with semi-domains rather than domains (i.e. complete lattices). I have found that this has made the theory much simpler and less messy (in an earlier version I worked with domains). Using semi-domains is, of course, not a new idea, and a number of people are toying with the pros and cons of it. In this appendix I list some of the advantages I have found, as yet no disadvantages have come to light. I do not feel that the advantages given below constitute conclusive evidence in favour of semi-domains and I should like to see a list of advantages of domains for comparison.

Advantages of semi-domains over domains:

1:

There is no need to have a top element $\top$, so there is no problem of intuiting the meaning of $\top$. Standard functions do not have to manipulate $\top$ so problems such as: what does "car(cons($\top,\bot$))" mean? do not arise. Also the absence of $\top$ can eliminate from proofs special ad hoc arguments needed to deal with it.

2:/

2:

There is only one contender for the conditional function and this has the obvious definition viz.

$$(z \supset x,y) = \begin{cases} x & z=tt \\ y & z=ff \\ \perp & \text{otherwise} \end{cases}$$

In logical space (the domain version of U_∞) tt, ff are not maximal and consequently both of the two possible conditionals have opaque definitions viz.

$$(z \supset x,y) = \begin{cases} x \sqcup y & z=\top \\ x & tt \sqsubseteq z \neq \top \\ y & ff \sqsubseteq z \neq \top \\ \perp & \text{otherwise} \end{cases}$$

$$(z :\supset x,y) = \begin{matrix} \top & z=\top \\ x & tt \sqsubseteq z \neq \top \\ y & ff \sqsubseteq z \neq \top \\ \perp & \text{otherwise} \end{matrix}$$

Then one also has the problem of when to use $\supset$ and when to use $:\supset$

3:

The construction of U_∞ is simpler than that of logical space because to prove U_∞ is a semi-domain we only need to show it is closed under directed unions and this is easy in view of the formula:

$\bigsqcup_i (x_i^n)/$

$$\bigsqcup_i (x_i^n)_{n=0}^{\infty} = (\bigsqcup_i x_i^n)_{n=0}^{\infty}$$

which tells the whole story for semi-domains (i.e. directed unions) but does not work for arbitrary unions.

<u>4</u>:

It is easy to show that the set of fixed points of $f \in [D\to D]$ is a semi-domain since for all <u>directed</u> $X \subseteq D$ $f(\bigsqcup X) = \bigsqcup f(X)$. This formula does not hold for arbitrary X and so cannot be used to prove that the set of fixed points of f is a domain - the proof of this is tricky. Also in semi-domains $\bigsqcup_{fix(f)} = \bigsqcup_D$ which is not true for domains.

<u>5</u>:

Every retract of U is isomorphic to a sub semi-domain of U (see 6.6.7) - I do not know if this is true for logical space. Thus we only need one symbol $\bigsqcup$ (cf. 4 above). The set of retractions of U is a sub semi-domain of it and so when working with retractions we do not need to distinguish $\bigsqcup_U X$ from $\bigsqcup_{fix(\lambda u.u\ u)} X$ as they are the same.

<u>6</u>:

The coalesced sum, $\oplus$, and coalesced product, $\otimes$, have natural definitions as retractions in U and this makes some standard semi-domains easy to represent e.g. (see 6.6.16)

$$N = \underline{O} \oplus N$$

I/

I cannot see how to do this in logical space. I found a* pretty hard to define in logical space (several complex auxiliary functions seemed needed), in U we have the lucid definition

$$a^* = \underline{0} \oplus (a^{+} \otimes a^*)$$

7:

The coalesced sum of two <u>continuous</u> semi-domains is a <u>continuous</u> semi-domain. That fact that the coalesced sum of continuous domains need not be continuous has been taken as evidence that the separated sum is what is needed (see e.g. [17]). I feel that both are useful (see 6.6.15, 6.6.20 and 6.8 for uses) and if one uses semi-domains then one can have both the continuity axiom and $\oplus$.

References

[1] de Bakker, J.W. and de Roever, W.P. A calculus for recursive program schemes. Proc. IRIA Symposium on Automata, Formal Languages and Programming. Amsterdam: North Holland.

[2] Cadiou, J.M. Recursive definitions of partial functions and their computations. Stanford Artificial Intelligence Project Memo. AIM-163, STAN-CS-266-72. (1972).

[3] Gordon, M.J.C. An investigation of lit: where $lit((A_1 \ldots A_n), A_{n+1}, f) = f(A_1, f(A_2, \ldots, f(A_n, A_{n+1})))$. Research Memorandum MIP-R-101. Edinburgh: School of Artificial Intelligence, University of Edinburgh. (1973).

[4] Lakatos, I. Proofs and refutations. Brit. J. Phil. of Sci. 14. (1963-4).

[5] Landin, P.J. A λ-calculus approach. Advances in Programming and Non-numerical Computation. Pergamon Press. (1966).

[6] McCarthy, J. et. al. LISP 1.5 Programmers Manual. MIT Press. (1969).

[7] McGowan, C. Correctness results for lambda calculus interpreters. Ph.D. Thesis. Cornell University. (1971).

[8]/

[8] Milne, R. The formal semantics of programming languages and their implementations. Ph.D. Thesis. Cambridge University (to appear).

[9] Milne, R. Private communication.

[10] Milner, R. Implementation and applications of Scott's logic for computable functions. Proceedings of an ACM Conference on Proving Assertions about Programs. ACM (1972).

[11] Milner, R. An approach to the semantics of parallel programs. Internal Memorandum. Edinburgh: Department of Computer Science, University of Edinburgh.

[12] Morris, J.H. Jr. Lambda calculus models of programming languages. MAC-TR-57, MIT Project MAC, Cambridge, Massachusetts. (1968).

[13] Moses, J. The function of FUNCTION in LISP or why the FUNARG problem should be called the Environment Problem. MAC-M-428, MIT Project MAC, Cambridge, Massachusetts. (1970).

[14] Mosses, P.D. Forthcoming Programming Research Group Technical Monograph, Oxford University Computing Laboratory.

[15]/

[15] Plotkin, G.D. A set-theoretical definition of application. Research Memorandum MIP-R-95. Edinburgh: School of Artificial Intelligence, University of Edinburgh. (1972).

[16] Plotkin, G.D. Forthcoming Research Memorandum, School of Artificial Intelligence, University of Edinburgh.

[17] Reynolds, J.C. Notes on a lattice-theoretic approach to the theory of computation. Systems and Information Science Syracuse University. (1972).

[18] Reynolds, J.C. Definitional interpreters for higher-order programming languages. Proc. 25th National ACM Conference, Boston. (1972).

[19] Reynolds, J.C. On the interpretations of Scott's domains. Convegno d'Informatica Teorica, Instituto Nazionale di Alta Matematica (città Universitaria), Rome, Italy. February 6, (1973).

[20] Scott, D. Outline of a mathematical theory of computation. Proc. Fourth Annual Princeton Conf. on Information Sciences and Systems. (1970).

[21]/

[21] Scott, D. The lattice of flow diagrams. Symposium on Semantics of Algorithmic Languages (ed. E. Engeler) Springer Lecture Note Series No. 188, Springer-Verlag, Heidelberg. (1971).

[22] Scott, D. and Strachey, C. Towards a mathematical semantics for computer languages. Proc. Symposium on Computers and Automata. Microwave Research Institute Symposia Series, Vol. 21, Polytechnic Institute of Brooklyn. (1972).

[23] Scott, D. Data types as lattices. Notes. Amsterdam. (1972).

[24] Wadsworth, C.P. Notes on Continuations. Private communication.

[25] Wadsworth, C.P. The relationship between lambda-expressions and their denotations in Scott's models for the lambda-calculus. Proceedings of the Orleans Conference (eds. Calais, Denick and Sabbagh). To appear.