3rd D R A F T

T H E D Y N A M I C O P E R A T I O N A L S E M A N T I C S O F S T A N D A R D M L

Robin Milner, April 1985

1 INTRODUCTION

This paper formally defines the dynamic operational semantics of the Standard ML core language, as reported in a forthcoming report of the Computer Science Department, Edinburgh University (referred to below as CoreML). The semantics is presented in the structural axiomatic style which is most fully developed in Plotkin (A Structural Approach to Operational Semantics, Report DAIMI FN-19, Computer Science Department, University of Aarhus), having its origin in the reduction rules of the lambda calculus.

The core language is reducible to a "bare" language, by translating convenient syntactic forms into their bare equivalent. It is therefore only necessary to treat the bare language here. The syntax of the bare language is tabulated below, in a very slightly modified version of the original, which appeared as Section 2.8 of CoreML. The modifications are: (1) Some syntax classes have been given more descriptive names; (2) Some information - useful only for parsing and type-checking - has been omitted, namely precedence rules, optional parentheses, explicit type constraints and the separate syntax classes apat, aexp for atomic patterns and atomic expressions.

The static semantics is not covered in this paper. However, two aspects of the static semantics are relevant here. First, it determines whether each identifier occurrence is a variable, a constructor, a type variable, a type constructor, a record label or an exception name. We therefore assume this knowledge here. Second, and more significantly, the static semantics determines whether each top-level phrase is well-typed; if so, it provides the types of all its component phrases. This allows the meaning of certain overloaded function identifiers (e.g. +) to be settled. Apart from this, evaluation of well-typed phrases depends in no way upon their type, and so our evaluation rules make no mention of type (if a phrase is ill-typed we do not care what evaluation, if any, is given to the phrase by our rules). To justify this, we expect in later work to formulate and prove a theorem which asserts that, if a phrase is well-typed, then its evaluation according to the present rules agrees, in an appropriate sense, with evaluation according to a more refined scheme in which values carry their types, and in which instances of ill-typing are detected "at run-time".

This approach - factoring apart the static and dynamic semantics - agrees with the usual way of implementing ML, where type-checking and evaluation are separate phases. The Type Environment, which is used and modified during type-checking, is of no further concern in this paper; occurrences of types and type bindings are treated by our rules as though they were absent.

The syntax table defines the principal syntax classes of ML in terms of the following primitive classes, representing the six uses of identifiers:

var	value variables
con	value constructors
tyvar	type variables
tycon	type constructors
lab	record labels
exn	exception names

Note that value constants (resp. type constants) are a subclass of value constructors (resp. type constructors). Essentially, a value constant con is a value constructor whose argument type is "unit"; a value constant is declared by omitting the "of" part in a generic type binding. Thereafter every occurrence of con, in its role as a value constant, is to be regarded as an abbreviation for "con()" - the construction resulting from applying con to the null tuple.

THE BARE SYNTAX OF ML

Conventions:

(1) <<..>> means optional.

(2) Repetition of iterated phrases is represented by "__".

(3) For any syntax class s, define

```
s_seq ::=  s
           (s1, __ ,sn)   (n>1)
```

EXPRESSIONS exp

```
exp ::=
  var                       (variable)
  con                       (constructor)
  { lab1=exp1 , __          (record,n>0)
    labn=expn }
  exp exp'                  (application)
  exp handle handler        (handle exc'ns)
  raise exn with exp        (raise exc'n)
  let dec in exp end        (local dec'n)
  fun match                 (function)

match ::=
  mrule1 | __ | mrulen      (n>1)

rule ::=
  pat => exp

handler ::=
  hrule1 || __ || hrulen    (n>1)

hrule ::=
  exn with match
  ? => exp
```

DECLARATIONS dec

```
dec ::=
  val valbind               (values)
  type tybind               (types)
  datatype databind         (datatypes)
  abstype databind
      with dec end          (abs. types)
  exception excbind         (exceptions)

  local dec in dec' end     (local dec'n)
  dec1 {;} __ decn {;}      (sequence,n>0)
```

| PROGRAMS : dec1 ; ..decn ; |

PATTERNS pat

```
pat ::=
  _                         (wildcard)
  var                       (variable)
  con                       (constant)
  { lab1=pat1 , __ ,        (record,n>0)
    labn=patn<<,...>>}
  con pat                   (construction)
  var as pat                (layered)
```

VALUE BINDINGS valbind

```
valbind ::=
  pat = exp                 (simple)
  valbind1 and __
      and valbindn          (multiple,n>2)
  rec valbind               (recursive)
```

TYPE BINDINGS tybind

```
tybind ::=
  <<tyvar_seq>>tycon
      = ty                  (simple)
  tybind1 and __
      and tybindn           (multiple,n>2)
```

DATATYPE BINDINGS databind

```
databind ::=
  <<tyvar_seq>>tycon
      = constrs             (recursive)
  databind1 and __
      and databindn         (multiple,n>2)

constrs ::=
  con1<<of ty1>>| __ | conn<<of tyn>>
```

EXCEPTION BINDINGS excbind

```
excbind ::=
  exn<<= exn'>>             (simple)
  excbind1 and __
      and excbindn          (multiple,n>2)
```

TYPES ty

```
ty ::=
  tyvar                     (type variable)
  <<ty_seq>>tycon           (type constr'n)
  { lab1:ty1 , __           (record type,n>0)
```

---

labn:tyn }
ty -> ty' (function type)

2 SEMANTIC CONSTRUCTIONS

2.1 Object Classes

Before giving the operational semantic rules, we must define the objects in terms of which they are expressed. Some of these objects are syntactic, and these were defined in the previous section. For the further object classes we first need to postulate two primitive sets; these are Addr , the addresses (or references), and Exc , the exceptions. Each of these sets is denumerably infinite; the nature of their elements is immaterial except that we assume that they may be tested for equality. (An implementation also needs to be able to generate an element different from any of an arbitrary finite set of elements.)

We also need the set Basfun, the names of basic functions. These are the functions which are bound to value variables in the standard environment, which are not definable in ML itself, and which have no side-effects. See Section 2.3 on the standard environment.

In the following definitions of object classes, we use the cartesian product (x) and disjoint union (+) of sets, and MAP(S,S') - the set of finite partial functions from S to S'. The object classes Val of values and Valenv of value environments are defined by mutual recursion. On the right of the page, after each object class, is a typical variable name (or notation) which will denote a member of the class.

VALUES

Addr		addr
Record	= MAP (Lab , Val)	{lab1=val1,__,labn=valn}
Fun	= {ASS,REF} + Basfun + Closure	
Closure	= Match x Env x Valenv	[match,E,VE]
Val	= Con + (Con x Val) + Record + Addr + Fun	val

ENVIRONMENTS

Valenv	= MAP (Var , Val)	VE
Excenv	= MAP (Var , Exc)	EE
Env	= Valenv x Excenv	E or (VE,EE)

STORES

Mem	= MAP (Addr , Val)	mem
Excs	= MAP (Exc , Exn)	excs
Store	= Mem x Excs	store or mem,excs

PACKETS

Pack	= Exc x Val	pack or <exc,val>

Notes

1) A function value which is a closure, [match,E,VE] , is the value of an ML expression "fn match"; see 2.2 below.

2) The first component mem of a store (mem,excs) is a normal map from addresses to values. The second component excs , a map from exceptions to exception names, serves two purposes. First, it records all exceptions which have been generated. Second, for each such exception, it records the exception name to which it was first bound by a declaration; this permits the exception name to be reported at top-level whenever an exception packet is raised but not handled.

3) An exception packet consists of an exception paired with an "excepted" value. The exception component determines which handling rule (if any) of a handler will handle the packet; then the match part of this rule will be applied to the excepted value.

If m is any map in MAP(X,Y), then m(x) denotes y whenever (x,y) is a member of m. Maps are often written explicitly as {(x1,y1),__,(xn,yn)} . For two maps m and m', the map m+m' is defined as follows:

$$(m+m')(x) = \begin{cases} m'(x) & \text{if } m'(x) \text{ is defined,} \\ m(x) & \text{otherwise} \end{cases}$$

For environments E = (VE,EE) and E' = (VE',EE'), we write E+E' for (VE+VE',EE+EE'). We also write E+VE' for (VE+VE',EE), and E+EE' for (VE,EE+EE').

2.2 Closures

The environment component E of a closure [match,E,VE] is the environment in which the function was defined, i.e. the one in which its body must be evaluated when it is applied to a value. In addition, if the function was declared by (simultaneous) recursion, then the third component VE of the closure will be non-empty and will contain value bindings which are to be interpreted recursively when the function is applied. This treatment of function values is discussed in detail in the Appendix, where an "unrolling" operation on Value environments

```
REC : Valenv -> Valenv
```

is described informally. Here we give its formal definition, since it is used in semantics rules for function declaration and application:

```
  If  VE = { (var1,val1), __, (varn,valn) }
then  REC(VE) = { (var1,recVE(val1)), __, (varn,recVE(valn)) }
```

where the operation recVE on values is defined, for given VE, as follows:

```
recVE(con) = con
recVE(con val) = con(recVE(val))
recVE({lab1=val1,__,labn=valm}) = {lab1=recVE(val1),__,labn=recVE(valm)}
recVE(addr) = addr
recVE([match,E',VE']) = [match,E',VE]
recVE(f) = f                                   (f in Fun\Closure)
```

2.3 The Standard Environment

Since we are not concerned here with static semantics, the standard Type Environment is of little relevance; all we need to know is that it establishes the following value constructors: nil, ::, true and false (nil, true and false being constants). Note that the identifier ref is also allowed in constructor position in patterns, but is treated by a special rule.

For the purpose of dynamic semantics, the Standard Environment - called ENV0 in CoreML - consists of the standard value environment VENV0 and the standard exception environment EENV0. VENV0 contains three kinds of binding:

(1) Each basic function identifier, i.e. each member of Basfun, is bound to itself. Recall that these identifiers are indeed values, as defined in Section 2.1. For each F in Basfun, the result of applying F to a value v is independent of the context of evaluation, and is written "APPLY(F,v)" in our rules of evaluation. The members of Basfun are as follows:

```
div  mod  *  /  +  -  ~  abs  floor  real  sqrt  sin  cos  arctan
 exp  ln  size  chr  ord  explode  implode  =  <>  <  >  <=  >=
```

(2) The function identifiers := and ref are bound to ASS and REF respectively. Since these standard "functions" cause side-effects on the memory, there are special rules dealing with their application to a value.

(3) The following declarations are also assumed to contribute to VENV0; they declare those standard functions which are easily definable in ML. (For clarity we show the standard infix status for these functions; of course several members of Basfun also have infix status.)

```
fun map f nil = nil
  | map f (x::L) = (f x)::(map f L)

val rev = let fun rev' L' nil = L'
                | rev' L' (x::L) = rev' (x::L') L
          in rev' nil end

infix 5 fun   nil M = M
  | (x::L) M = x::(L M

infix 6
fun s  s' = implode [explode s, explode s']

fun not true  = false
  | not false = true

fun ! (ref x) = x

infix 3 o
fun (f o g) x = f (g x)
```

In EENV0, distinct exceptions are bound to the following standard exception names: match, bind, interrupt, ord, chr, *, /, div, mod, +, -, floor, sqrt, exp, ln. The conditions under which these exceptions are raised are given in Section 5.3 of CoreML.

3 EVALUATION RULES

In a given environment, and for each syntax class, the evaluation of each phrase of the class yields a result (whose nature is dependent upon the syntax class). This may be expressed by a formal sentence whose simplest form is

$$E \vdash \text{phrase} \Longrightarrow \text{result}$$

However, for some syntax classes, the evaluation requires another parameter. For example, the evaluation of a match requires as an extra parameter a value (to which the match is being applied); in this case the form of the sentence is

$$E \vdash \text{match, val} \Longrightarrow \text{result}$$

We shall sometimes omit the environment in such sentences, when it does not affect the evaluation.

Evaluation is formally defined by inference rules, from which sentences may be inferred. When all the rules are given, the evaluation relation "===>" is then fully defined, and we call its members (the inferrable sentences) the evaluations. The general form of an evaluation rule is

$$\frac{E1 \vdash item1 \Longrightarrow result1 \quad \ldots \quad En \vdash itemn \Longrightarrow resultn}{E \vdash item \Longrightarrow result} \qquad (1)$$

The sentences above the line are the hypotheses, and that below the line is the conclusion, of the rule. In an instance of the rule, whose conclusion evaluates a certain phrase, the hypotheses are usually evaluations of subphrases; we therefore call them subevaluations. When there are no hypotheses (n=0) then the horizontal line is omitted, and we may call the rule an axiom; its instances are atomic evaluations, having no subevaluations.

So far we have not explicitly considered side-effects of evaluation, in particular changes to the store by assignment. These changes can indeed be accommodated by allowing every item (to be evaluated) and every result (of evaluation) to contain a store component. Since this component is always present, we render it explicit by adopting the following full form of sentence:

$$E \vdash \begin{array}{c} item \\ \hline store \end{array} \Longrightarrow \begin{array}{c} result \\ \hline store' \end{array}$$

But it would be tedious to write this full form in all our rules, since so few of them contribute side-effects directly. We therefore adopt the convention that when a rule is written in the form (1) above, it is intended to mean that the side-effects of all the subevaluations, taken in order from left to right, comprise the total side-effect of the main evaluation. That is, (1) is an abbreviation for the following rule:

$$\frac{E1 \vdash \begin{array}{c} item1 \\ \hline store0 \end{array} \Longrightarrow \begin{array}{c} result1 \\ \hline store1 \end{array} \quad \ldots \quad En \vdash \begin{array}{c} itemn \\ \hline store(n-1) \end{array} \Longrightarrow \begin{array}{c} resultn \\ \hline storen \end{array}}{E \vdash \begin{array}{c} item \\ \hline store0 \end{array} \Longrightarrow \begin{array}{c} result \\ \hline storen \end{array}}$$

Note that in the case of an axiom (n=0) this degenerates to

```
                item          result
E |-  ======  ===>  =======
      store0        store0
```

which is an (atomic) evaluation causing no side-effect.

Concerning exceptions there is a uniform principle that, whenever the result of a subevaluation is a packet, then no further subevaluations occur, and the packet is also the result of the main evaluation. This principle applies to all evaluations except - of course - the evaluation of a handle expression, whose purpose is precisely to trap certain packets and hence to violate the principle. Thus we adopt the convention that, for every rule whose full form is

```
      item1       result1                    itemn       resultn
E1 |- =====  ===> =======    ...    En |-  =====  ===> =======
      store1      store1'                   storen      storen'
-------------------------------------------------------------------
                    item       result
               E |- =====  ===> ======
                    store      store'
```

- except for the rule for evaluating a handle expression - we assume the presence of n further rules, one for each k ($1<k<n$) as follows:

```
      item1       result1                    itemk        pack
E1 |- =====  ===> =======    ...    Ek |-  =====  ===> =======
      store1      store1'                   storek      storek'
-------------------------------------------------------------------
                    item         pack
               E |- ====   ===> =======
                    store       storek'
```

on condition that result1,..,result(k-1) are not packets. Note particularly that the store resulting from the main evaluation is exactly that which results from the first subevaluation which returns a packet.

With the help of these conventions, the presentation of the rules becomes reasonably brief. The ensuing subsections deal with separate phrase classes in order: patterns, matches, handlers, expressions, value bindings, exception bindings and declarations.

3.1 Matching a Pattern to a Value

In matching, a pair consisting of a pattern and a value evaluates either to a value environment or to fail.

$$\vdash \text{pat} , \text{val} \Longrightarrow \text{VE}$$
$$\vdash \text{pat} , \text{val} \Longrightarrow \text{FAIL}$$

The second case (evaluation to FAIL) holds just in the case that no evaluation can be inferred for the pair (pat,val) from the rules below. Matching is independent of the environment.

$$\vdash \_ , \text{val} \Longrightarrow \{\}$$

$$\vdash \text{var} , \text{val} \Longrightarrow \{(\text{var},\text{val})\}$$

$$\vdash \text{con} , \text{con} \Longrightarrow \{\}$$

$$\frac{\vdash \text{pat} , \text{val} \Longrightarrow \text{VE}}{\vdash \text{con pat} , \text{con val} \Longrightarrow \text{VE}}$$

$$\frac{\vdash \dfrac{\text{pat} , \text{val}}{\text{mem,excs}} \Longrightarrow \dfrac{\text{VE}}{\text{mem,excs}}}{\vdash \dfrac{\text{ref pat} , \text{addr}}{\text{mem,excs}} \Longrightarrow \dfrac{\text{VE}}{\text{mem,excs}}} \quad (\text{mem}(\text{addr})=\text{val}) \quad **$$

$$\frac{\vdash \text{pat} , \text{val} \Longrightarrow \text{VE}}{\vdash \text{var as pat} , \text{val} \Longrightarrow \{(\text{var},\text{val})\} + \text{VE}}$$

$$\frac{\vdash \text{pat1} , \text{val1} \Longrightarrow \text{VE1} \quad \_\_ \quad \vdash \text{patn} , \text{valn} \Longrightarrow \text{VEn}}{\vdash \{\text{lab1=pat1},\_\_,\text{labn=patn}\},\{\text{lab1=val1},\_\_,\text{labn=valn}\} \Longrightarrow \text{VE1}+\_\_+\text{VEn}} \quad (n>0)$$

$$\frac{\vdash \text{pat1} , \text{val1} \Longrightarrow \text{VE1} \quad \_\_ \quad \vdash \text{patn} , \text{valn} \Longrightarrow \text{VEn}}{\vdash \{\text{lab1=pat1},\_\_,\text{labn=patn},\ldots\},\{\text{lab1=val1},\_\_,\text{labm=valm}\} \Longrightarrow \text{VE1}+\_\_+\text{VEn}} \quad (m>n>0)$$

** In this rule the store is made explicit, since the evaluation depends upon it. Note that the store is assumed to be unchanged by the subevaluation; it is easy to show that pattern matching has no side-effects.

3.2 Applying a Match

In the application of a match to a value, a pair consisting of a match and a value evaluates either to a value, or to a packet. As part of this evaluation, the rules of the match (here called mrules) are applied singly to the value. A pair consisting of a mrule and a value evaluates either to a value, or to a packet, or to FAIL.

```
E |-  mrule , val ===> val' // pack // FAIL
E |-  match , val ===> val' // pack
```

The rules for applying an mrule are as follows:

$$\frac{\vdash pat\ ,\ val \Longrightarrow VE \qquad E+VE \vdash exp \Longrightarrow val'}{E \vdash pat \Rightarrow exp\ ,\ val \Longrightarrow val'}$$

$$\frac{\vdash pat\ ,\ val \Longrightarrow FAIL}{E \vdash pat \Rightarrow exp\ ,\ val \Longrightarrow FAIL}$$

The rules for applying a match are as follows:

$$\frac{E \vdash mrule1\ ,\ val \Longrightarrow FAIL \quad\_\_\quad E \vdash mrule(k-1)\ ,\ val \Longrightarrow FAIL \qquad E \vdash mrulek\ ,\ val \Longrightarrow val'}{E \vdash mrule1 \mid\_\_\mid mrulen\ ,\ val \Longrightarrow val'} \quad (\ 1<k<n\ )$$

$$\frac{E \vdash mrule1\ ,\ val \Longrightarrow FAIL \quad\_\_\quad E \vdash mrulen\ ,\ val \Longrightarrow FAIL}{E \vdash mrule1 \mid\_\_\mid mrulen\ ,\ val \Longrightarrow \langle ematch,() \rangle} \quad (\ n>1\ )$$

3.3 Applying a Handler

In the application of a handler to a packet, a pair consisting of a handler and a packet evaluates either to a value, or to a packet. As part of this evaluation, handling rules (hrules) of the handler are applied singly to the packet; a pair consisting of a hrule and a packet evaluates either to a value, or to a packet, or to FAIL.

```
E |- hrule , pack ===> val // pack' // FAIL
E |- handler , pack ===> val // pack'
```

The rules for applying a hrule are as follows:

```
VE,EE |- exn with match , <exc,val> ===> FAIL                ( EE(exn)#exc )

     VE,EE |- match , val ===> val'
    ---------------------------------                        ( EE(exn)=exc )
VE,EE |- exn with match , <exc,val> ===> val'

        E |- exp ===> val'
    ------------------------
E |- ? => exp , pack ===> val'
```

The rules for applying a handler are as follows:

```
E |- hrule1 , pack ===> FAIL   __   E |- hrule(k-1) , pack ===> FAIL
                E |- hrulek , pack ===> val'
        ---------------------------------------              ( 1<k<n )
        E |- hrule1||__||hrulen , pack ===> val'

E |- hrule1 , pack ===> FAIL   __   E |- hrulen , pack ===> FAIL
        ---------------------------------------              ( n>1 )
        E |- hrule1||__||hrulen , pack ===> pack
```

3.4 Evaluating an Expression

An expression evaluates either to a value, or to a packet.

$$E \vdash exp \Longrightarrow val \;//\; pack$$

The rules are as follows:

$$VE, EE \vdash var \Longrightarrow val \qquad (\ VE(var)=val\ )$$

$$E \vdash con \Longrightarrow con$$

$$\frac{E \vdash exp1 \Longrightarrow val1 \quad .. \quad E \vdash expn \Longrightarrow valn}{E \vdash \{lab1=exp1,\_\_,labn=expn\} \Longrightarrow \{lab1=val1,\_\_,labn=valn\}} \qquad (\ n>0\ )$$

$$\frac{E \vdash exp \Longrightarrow con \qquad E \vdash exp' \Longrightarrow val'}{E \vdash exp\ exp' \Longrightarrow con(val')}$$

$$\frac{E \vdash \begin{array}{c} exp \\ \hline mem,excs \end{array} \Longrightarrow \begin{array}{c} ASS \\ \hline mem',excs' \end{array} \qquad E \vdash \begin{array}{c} exp' \\ \hline mem',excs' \end{array} \Longrightarrow \begin{array}{c} (addr,val) \\ \hline mem'',excs'' \end{array}}{E \vdash \begin{array}{c} exp\ exp' \\ \hline mem,excs \end{array} \Longrightarrow \begin{array}{c} () \\ \hline mem''+\{(addr,val)\},\ excs'' \end{array}}$$

$$\frac{E \vdash \begin{array}{c} exp \\ \hline mem,excs \end{array} \Longrightarrow \begin{array}{c} REF \\ \hline mem',excs' \end{array} \qquad E \vdash \begin{array}{c} exp' \\ \hline mem',excs' \end{array} \Longrightarrow \begin{array}{c} val \\ \hline mem'',excs'' \end{array}}{E \vdash \begin{array}{c} exp\ exp' \\ \hline mem,excs \end{array} \Longrightarrow \begin{array}{c} addr \\ \hline mem''+\{(addr,val)\},\ excs'' \end{array}} \qquad (addr\ not\ bound\ in\ mem'')$$

$$\frac{E \vdash exp \Longrightarrow F \qquad E \vdash exp' \Longrightarrow val}{E \vdash exp\ exp' \Longrightarrow APPLY(F,val)} \qquad (F\ in\ Basfun)$$

$$\frac{E \vdash exp \Longrightarrow [match,E',VE'] \qquad E \vdash exp' \Longrightarrow val' \qquad E'+REC(VE') \vdash match\ ,\ val' \Longrightarrow val}{E \vdash exp\ exp' \Longrightarrow val}$$

$$\frac{VE,EE \vdash exp \Longrightarrow val}{VE,EE \vdash \underline{raise}\ exn\ \underline{with}\ exp \Longrightarrow \langle exc, val\rangle} \quad (\ EE(exn)=exc\ )$$

$$\frac{E \vdash dec \Longrightarrow E' \qquad E+E' \vdash exp \Longrightarrow val}{E \vdash \underline{let}\ dec\ \underline{in}\ exp\ \underline{end} \Longrightarrow val}$$

$$E \vdash \underline{fn}\ match \Longrightarrow [match, E, \{\}]$$

** $$\frac{E \vdash exp \Longrightarrow val}{E \vdash exp\ \underline{handle}\ handler \Longrightarrow val}$$

** $$\frac{E \vdash exp \Longrightarrow pack \qquad E \vdash handler\ ,\ pack \Longrightarrow val' \,//\, pack'}{E \vdash exp\ \underline{handle}\ handler \Longrightarrow val' \,//\, pack'}$$

** The last two rules, concerning exception handling, are the only rules for which the auxiliary rules for exception transmission are not added.

3.5 Evaluating a Value Binding

A value binding evaluates either to a value environment, or to a packet.

$$E \vdash \text{valbind} \Longrightarrow VE \;//\; pack$$

The rules are as follows:

$$\frac{E \vdash exp \Longrightarrow val \qquad \vdash pat\ ,\ val \Longrightarrow VE \;//\; FAIL}{E \vdash pat = exp \Longrightarrow VE \;//\; \langle ebind, () \rangle}$$

$$\frac{E \vdash valbind1 \Longrightarrow VE1 \quad .. \quad E \vdash valbindn \Longrightarrow VEn}{E \vdash valbind1\ \underline{and}\ \_\_\ \underline{and}\ valbindn \Longrightarrow VE1+\_\_+VEn} \quad (n>2)$$

$$\frac{E \vdash valbind \Longrightarrow VE}{E \vdash \underline{rec}\ valbind \Longrightarrow REC(VE)}$$

3.6 Evaluating an Exception Binding

An exception binding evaluates to an exception environment.

$$E \vdash excbind \Longrightarrow EE$$

The rules are as follows:

$$** \qquad E \vdash \frac{exn}{(mem, excs)} \Longrightarrow \frac{\{(exn, exc)\}}{(mem, excs+\{(exc, exn)\})} \cdot (\text{exc not in excs})$$

$$VE, EE \vdash exn = exn' \Longrightarrow \{(exn, exc)\} \qquad (exc = EE(exn'))$$

$$\frac{E \vdash excbind1 \Longrightarrow EE1 \quad .. \quad E \vdash excbindn \Longrightarrow EEn}{E \vdash excbind1\ \underline{and}\ \_\_\ \underline{and}\ excbindn \Longrightarrow EE1+\_\_+EEn} \quad (n>2)$$

** In this rule the store is made explicit since the evaluation both depends upon the store and changes it.

3.7 Evaluating a Declaration

A declaration evaluates either to an environment, or to a packet.

$$E \vdash dec \Longrightarrow E' \;//\; pack$$

The rules are as follows:

$$\frac{E \vdash valbind \Longrightarrow VE}{E \vdash \underline{val}\ valbind \Longrightarrow VE,\{\}}$$

$$\frac{E \vdash excbind \Longrightarrow EE}{E \vdash \underline{exception}\ excbind \Longrightarrow \{\},EE}$$

$$E \vdash \underline{type}\ typbind \Longrightarrow \{\},\{\}$$

$$E \vdash \underline{datatype}\ databind \Longrightarrow \{\},\{\}$$

$$\frac{E \vdash dec \Longrightarrow E'}{E \vdash \underline{abstype}\ databind\ \underline{with}\ dec\ \underline{end} \Longrightarrow E'}$$

$$\frac{E \vdash dec1 \Longrightarrow E1 \qquad E+E1 \vdash dec2 \Longrightarrow E2}{E \vdash \underline{local}\ dec1\ \underline{in}\ dec2\ \underline{end} \Longrightarrow E2}$$

$$\frac{E \vdash dec1 \Longrightarrow E1 \qquad E+E1 \vdash dec2 \Longrightarrow E2 \qquad \cdots \qquad E+E1+\cdots+E(n-1) \vdash decn \Longrightarrow En}{E \vdash dec1\ \cdots\ decn \Longrightarrow E1+\cdots+En} \quad (n>0)$$

Note: there are no rules for evaluating type bindings and datatype bindings, since they have no effect at run-time.

APPENDIX: The treatment of Recursion

The treatment of recursion in a semantic definition is peculiarly central, since it is only through recursion that a program can perform computations whose length is not bounded by some simple function of the size of the program.

In this Appendix we first examine the syntactic restriction which should be placed upon recursive value bindings, and then we discuss the treatment of function values which we have adopted, and the method by which recursive bindings are evaluated.

Consider a value binding

f = exp

in which exp is of functional type. The binding is evaluated in a given environment E by first evaluating exp to a value v (in this case a function value), and then returning the value environment VE={(f,v)}. Now it is natural to expect that the value environment returned by the evaluation of the recursive binding

rec f = exp

is obtained in a simple manner from VE, rather than by other means.

With this prescription, it is confusing (at least) if the evaluation of exp entails the evaluation of any value variable. For if this variable is f itself (or something being declared simultaneously with f by mutual recursion) then the prescription implies that it will - in this evaluation - be interpreted in the outer environment E; this conflicts with our normal understanding of recursive bindings, which dictates in this case that f is interpreted in the environment which results from evaluating the recursive binding itself.

This argument led to the restriction in CoreML that the right side of any binding within rec (e.g. exp in the case above) should always be a function expression "$_{fn}$ match". This is more restrictive than we may wish, as the following example shows.

Consider the following data type of "lazy" lists:

```
datatype 'a lazylist = empty
                     | prefix of 'a * (unit -> 'a lazylist)
```

Intuitively, a non-empty lazy list L consists of an element, of some type 'a, paired with a function (of no arguments) returning a lazy list. Then, to define a lazy list ONES which is in effect the infinite sequence of 1's, we expect to be allowed to write

```
val rec ONES : int lazylist = prefix(1, fn ()=>ONES )          (*)
```

But the strong restriction above forbids this, and we must write instead

```
val rec ONES' : unit -> int lazylist = fn() => prefix(1,ONES')
val ONES = ONES'()
```

Of course the first line would naturally be sugared thus : "$_{fun}$ ONES'() = prefix(1,ONES')" .

Now there is a reasonable relaxed restriction which allows (*), for which the rules in this paper are sound. It is as follows:

For each value binding "pat = exp" within rec, the expression exp must be built from function expressions using only constructors and record formation.

The semantic treatment of recursive bindings described below is a minor modification of what Plotkin proposes in his Aarhus paper, and has been proved equivalent to it under the above relaxed restriction. Note that the restriction indeed ensures that no variable is evaluated during the evaluation of exp.

The representation for a function value in ML, following Landin, would be as a pair

[match,E]

where E is an environment. This would be the value of the function expression "fun match" evaluate in E. With this representation, the environment returned by evaluating the recursive binding

rec f = fn match

should be a value environment VE which satisfies the equation

VE = { (f, [match,E+VE]) }

While this equation can be shown to determine a unique - but infinitely deep - value environment VE, it is important in operational semantics to restrict our structures to be finite. This can be simply achieved as follows, and we thus justify the choice of the semantic structures used in this paper:

(1) We adopt the form [match,E,VE] for a function value. The third component records those value bindings to be interpreted recursively when the function is applied to a value.

(2) We define an operation REC upon value environments as follows:

if VE = { (var1,val1), .., (varn,valn) }
then REC(VE) = { (var1,val1'), .., (varn,valn') }

where, whenever vali contains a function value [match,E',VE'] not contained in another function value within vali, the value vali' contains instead the function value [match,E',VE] . This ensures that, when the function is applied to a value, the variables vari will be interpreted (recursively) in VE.

(3) When the evaluation of a value binding - valbind say - returns VE, then the evaluation of "$_{rec}$ valbind" returns REC(VE) .

(4) The value of "$_{fn}$ match" evaluated in E will be [match,E,{}] .

(5) When a function value [match,E,VE] is applied to a value v, then match will be applied to v in the environment E+REC(VE) .

In effect, the operation REC "unrolls" VE once, in preparation for any application; at each application of a function value its "recursive" component is "unrolled" once more in preparation for further applications.