# Implementing Operational Semantics (Preliminary Report)

Roy L. Crole

email: R.Crole@mcs.le.ac.uk
http://www.mcs.le.ac.uk/~rcrole

December 1997

**Abstract**

This paper describes a high level operational semantics for a simple programming language, called $\mathbb{KOREL}$, together with a parser, interpreter and pretty printer which are implemented in the (pure) functional programming language Haskell. The syntax of $\mathbb{KOREL}$ is presented via BNF grammars, and the operational semantics is specified via structured, inductive rules. The paper outlines the broad ideas behind the Haskell implementation, with a more detailed explication of the key techniques. A code listing can be found in [Cro97].

# 1 Introduction

This paper presents a high level operational semantics for a simple programming language, together with a parser, interpreter and pretty printer which are implemented in the (pure) functional programming language Haskell. Our programming language is called $\mathbb{KOREL}$, so called because it can be considered to represent the core programming features of a language combining imperative and functional constructs, and has both an eager and a lazy operational semantics. This paper proceeds as follows:

- in Section 2 we give the $\mathbb{KOREL}$ program syntax, together with brief explanations of some non-standard constructs;
- in Section 3 we outline an operational semantics;
- in Section 4 and Section 5 we give a broad specification of $\mathbb{KOREL}$, plus implementation details; and
- in Section 6 we give a full listing of the $\mathbb{KOREL}$ typing and operational semantics rules.

# 2 The Syntax Of $\mathbb{KOREL}$

We specify the syntax of $\mathbb{KOREL}$ using the BNF grammar given in Figure 1. Most of the constructs will be familiar to anyone who has programmed in imperative and functional languages; indeed, $\mathbb{KOREL}$ closely resembles ML [MTH90, MT91]. A few of the constructs may appear unfamiliar at first sight, but the intended interpretations are indicated in the BNF grammar. We make the following additional comments:

- `skip` is the command which "does nothing" when executed;
- sequencing terms `t1 ; t2` execute `t1` and then `t2`, and the terms can be of any type;
- references [MTH90, Pit97b] are at all types;
- functions, lists and pairs have both lazy and eager versions;
- pairs can be accessed through standard projections, or through the use of a splitting term.

For further details about lazy and eager program semantics, please see any of [Cro95a, Cro95b, Pit97a]. The paper [Plo75] contains one of the original accounts of notions of eagerness and laziness.

A few explanatory comments are in order. Notice that $\mathbb{KOREL}$ is a mixture of traditional imperative and functional constructs; in particular, the grammar defining the syntax of the language does not distinguish between syntactic classes of (for example) Boolean expressions, commands, and so on. The language as it stands is untyped, in the sense

| | |
|---|---|
| `alphachar ::= A \| a \| B \| b \| ... \| Z \| z` | |
| `var ::= alphachar`$^+$ | variables |
| `loc ::= L1 \| L2 \| L3 ...` | locations |
| `b ::= true \| false` | Booleans |
| `z ::= ... -2 \| -1 \| 0 \| 1 \| 2 ...` | integers |
| `iop ::= + \| - \| *` | integer operator |
| `bop ::= = \| <= \| >= \| < \| >` | Boolean operator |
| `op ::= iop \| bop` | |
| `t ::= var \| loc \| b \| z` | atomic datatype terms |
| `\| t iop t` | integer arithmetic |
| `\| t bop t` | Boolean arithmetic |
| `\| if t then t else t` | conditional |
| `\| skip` | null command |
| `\| t ; t` | sequencing |
| `\| while t do t` | while loop |
| `\| ref t` | reference |
| `\| !t` | reference lookup |
| `\| t:=t` | assignment |
| `\| local var := t in t` | local assignment |
| `\| %var -> t` | eager function |
| `\| #var -> t` | lazy function |
| `\| t t .. t` | function application |
| `\| []` | empty list |
| `\| t:t` | eager cons |
| `\| [t, .. ,t]` | non-empty eager list |
| `\| t::t` | lazy cons |
| `\| {t, .. ,t}` | non-empty lazy list |
| `\| hd t \| tl t` | head and tail |
| `\| el t` | test for empty list |
| `\| ()` | unit element |
| `\| <t,t>` | eager pair |
| `\| <<t,t>>` | lazy pair |
| `\| split t as var,var in t` | pair splitting |
| `\| fst t \| snd t` | first and second projections |
| `\| inl t \| inr t` | left and right inclusions |
| `\| case t of var.t or var.t` | case expression |
| `\| rec var.t` | recursion |
| `\| let var = t in t` | let expression |

Figure 1: The Syntax of $\mathbb{KOREL}$ Terms

that no type checker is actually implemented. However, it is quite simple to describe a type assignment system for $\mathbb{KOREL}$, in which the primary form of judgement would be

$$\underbrace{\texttt{var1} :: \sigma_1 \ldots \texttt{var2} :: \sigma_n}_{E} \vdash \texttt{t}::\sigma \qquad \textit{typing judgement}$$

asserting that the term `t` has type $\sigma$ in the given typing environment $E$. For more information about type assignment systems in general, see [HS88], [Pit97a], [Sch94].

In fact we give this type assignment system in Section 6, where for $\mathbb{KOREL}$, the BNF grammar of types would be

| | | | |
|---|---|---|---|
| $\sigma$ | ::= | $Cmd$ | commands |
| | \| | $Bool$ | Booleans |
| | \| | $Int$ | integers |
| | \| | $Ref(\sigma)$ | references |
| | \| | $\sigma \times \sigma$ | products |
| | \| | $\sigma + \sigma$ | (disjoint) sums |
| | \| | $Lists(\sigma)$ | lists |
| | \| | $\sigma \to \sigma$ | functions |

Note also that the usual variable binding operations hold for $\mathbb{KOREL}$ terms; for example, occurrences of the variables `var1` and `var2` in the term `t2` are *bound* in the term

```
split t1 as var1,var2 in t2.
```

We will refer to the occurrences immediately after the "`as`" and before the "`in`" as *binding* occurrences. Occurrences of variables which are not bound are *free*. Note that the type assignment system given in Section 6 implicitly indicates the formation of bound variables in terms. We shall use the notation `t[t'/var]` to denote the term `t` with all free ocurrences of `var` replaced by `t'`, renaming bound variables to avoid capture. We shall not say any more about free and bound variables, and substitution, as we assume familiarity. For a detailed account please see [HS88] or [Cro95a].

We aim to give an operational semantics to $\mathbb{KOREL}$ in both transition and evaluation styles. We assume the reader is familiar with structured operational semantics as introduced by Plotkin in [Plo81]. Evaluation (or natural) semantics was introduced by Kahn [Kah88]. For general accounts of such semantics, see for example [Win93], [Gun92]. For material which matches closely the presentation in *loc cit*, see [Cro95a], [Cro95b] or [Pit97a]. In order to give such a semantics to $\mathbb{KOREL}$, we shall need to define the *values* of the language; and in order to do this, we first need to define a notion of function *declaraton*. An example of a function declaration is

```
f g x y = (g x) + (g y) | g z = z + 1 | k = 3
```

which declares the definitions of the three functions `f`, `g` and `k`. Thus a function declaration acts like a simple Haskell [Tho19] script which gives the definitions of various functions. Note that `k` would usually be considered as "a constant". We shall regard `k` as a function with constant output 3, which will help to keep the exposition in *loc cit* uniform. In general, a $\mathbb{KOREL}$ function declaration is a finite list of function definitions, given in the form

Given a function declaration `d`, the corresponding values are

```
v ::= loc | b | z | skip
| %var -> t | #var -> t | var var*
| [] | v:v | t::t | [v, ..  ,v] | {t, ..  ,t}
| () | <v,v> | <<t,t>>
| inl v | inr v
```

where $0 \leq$ *length*(`var var`$^*$) $\leq$ *arity* `var`, and the (function) variable `var` is defined in `d`.

Figure 2: The Syntax of $\mathbb{KOREL}$ Values

| `d = fundef1 \| fundef2 \| ...  \| fundefm` | *function declaration* |
|---|---|

where each function definition takes the form

| `var var* = t` | *function definition* |
|---|---|

where `var`$^*$ indicates a finite (possibly zero) number of variables, and `t` is a term. We define the *arity* of the first `var` to be the length of `var`$^*$, and refer to it as a *function variable* (so for example the arity of `f` above is 3, with `var` being `f` and `var`$^*$ being `g x y`). We say that the function variables are *defined* in the declaration.

Given a particular declaration (say `d`), then the *values* of $\mathbb{KOREL}$ are given by the grammar in Figure 2. Note that in the example above, each of `f`, `f g`, `f g x` and `g` are values. The basic idea is that a function variable without its full quota of arguments is a value because there is insufficient data to actually evaluate the function. Once the number of arguments is equal to the arity of the function variable, then the term is no longer a value—we can use the function definition to compute the term: `f g 2 3 = (g 2) + (g 3) = ....`

Note that a consequence of the definition of values is that a variable `var` is a value if it is not defined in the given `d`. Unlike most standard programming languages, we will define and implement a semantics over general terms [Pit90], and not just closed ones (that is, those with no free variables).

Finally, we need to define a notion of *state*. This will be a finite list of pairs of locations and terms:

| `s = [(L1,v1),(L2,v2),...,(Ln,vn)]` | *state* |
|---|---|

for example

`s = [(L1, 3), (L2, <5,6>)]`  or  `s = []`.

For the first example, we say that the second location `L2` has a look-up value of `<5,6>`, and we write `s(L1) = <5,6>` to indicate this. In general `s(Li)` is the look-up value of

location `Li`, which is undefined if `Li` does not appear in `s`. (As a passing remark, note that in KOREL syntax the term `!l` yields the look-up of the location `l`. Thus `!L2` will evaluate to the value `<5,6>`.) For a state `s`, and location `Lk`, we write `s{Lk -> t}` for the state which is identical to `s`, except that the look-up of `Lk` is updated to be `t`:

$$\texttt{s\{Lk -> t\} (Li)} \;=\; \begin{cases} \texttt{t} & \text{if } i = k \\ \texttt{s(Li)} & \text{otherwise} \end{cases}$$

We also write $\texttt{s\{Lk\}}^{-1}$ for the state which is identical to `s`, except that the look-up of `Lk` is always undefined (and thus if `Lk` is defined in `s`, the location has been *deallocated*):

$$\texttt{s\{Lk\}}^{-1}\ \texttt{(Li)} \;=\; \begin{cases} \text{undefined} & \text{if } i = k \\ \texttt{s(Li)} & \text{otherwise} \end{cases}$$

We shall refer to a pair `(s,t)` consisting of a state and a term as a *configuration*, and a pair of the form `(d,(s,t))` where `d` is a declaration as a *program*.

# 3 The Semantics of KOREL

The meaning of the language is given by defining both a transition style and evaluation style operational semantics. The semantics is specified by giving inductive definitions of judgements [MTH90] of the forms given below

(1) `d |- <s, t> --> <s', t'>` transition semantics

(2) `d |- <s, t> ==> <s', v>` evaluation semantics

Thus, formally, `-->` is a ternary relation between function declarations, configurations and configurations, and if the triple

`(d, <s, t>, <s', t'>)`

is an element of `-->`, then we write (1) above to indicate this. The inductive definitions of `-->` and `==>` are quite standard, but we do give the derivation rules in Section 6. Note that as usual, the two semantics are connected by the fact that `==>` is "essentially" the transitive closure of `-->`:

`d |- <s, t> = <s1, t1> --> <s2, t2> ... --> <sn, vn>`

iff

`d |- <s, t> ==> <s', v>` where `<s',v> = <sn, vn>`

We call the family of configurations (`<si, vi>` | $1 \leq i \leq n$) the *full transition sequence* of the program `d |- <s, t>`, and `<s', v>` the *final* configuration.

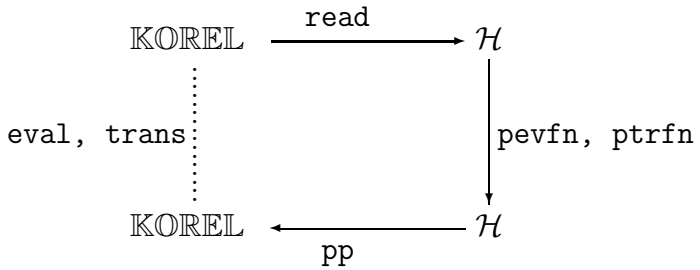
# 4 The Specification of KOREL

The programming language KOREL provides an implementation of the transition and evaluation semantics. The key point is that both of these relations are deterministic, that is, they determine partial functions:

$\forall$ `d |- <s, t> --> <s', t'>.` $\forall$ `<s'',t''>.`

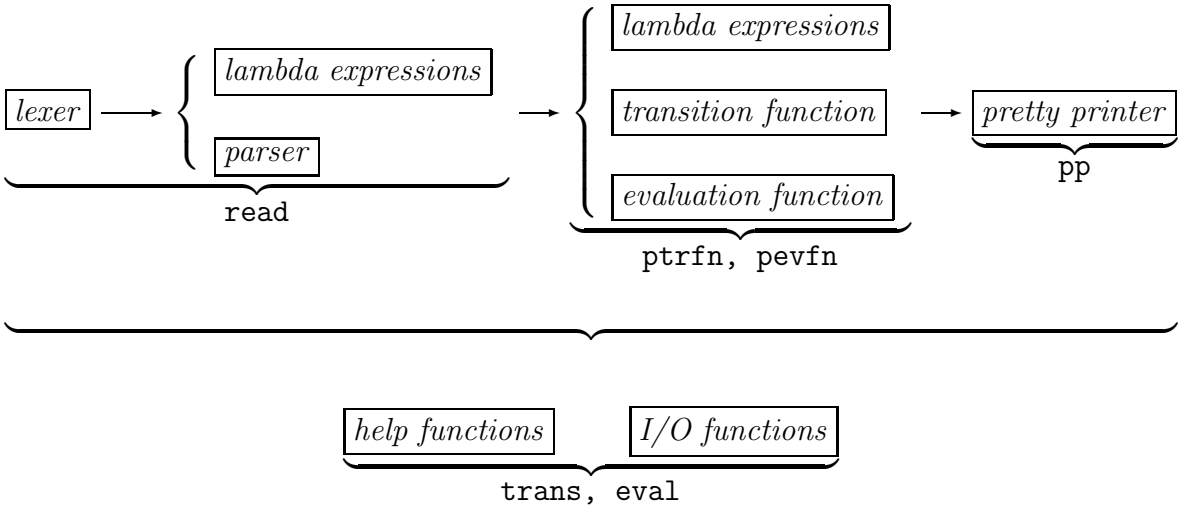
`d |- <s, t> --> <s'', t''>` $\implies$ `<s',t'>` $=$ `<s'',t''>`

The function `trans` implements `-->` with a corresponding function `eval` implementing `==>`. KOREL provides a user interface into which the user may type either the transition function or evaluation function, together with a program, and KOREL responds with either the resulting full sequence of transitions, or the final configuration. The user input is given in the form of a string of characters. However, this input must be represented internally in an appropriate fashion; this is achieved through the implementation of lexing and parsing functions, and pretty printing functions, which map KOREL programs (given as strings) into an internal representation, and vice versa:




For a general reference to functional lexing, parsing and pretty printing, see [Pau97]. Here, $\mathcal{H}$ denotes Haskell code which provides the internal machine representation. The functions `read` and `pp` are mutual inverses, which provide an interface for I/O between KOREL input and $\mathcal{H}$.

KOREL consists of the following components:




The lexer, parser (and lambda expressions) implement the `read` function; the lambda expressions, transition and evaluation functions implement `ptrfn` and `pevfn`; and the pretty

printer implements pp. These components are coded through the following modules: Basic.hs; Lambda.hs; Parse.hs; Scanner.hs; Pare.hs; Transem.hs; Evsem.hs; Pretty.hs; Help.hs; and Main.hs. We shall now describe, in outline, the implementation of KOREL via these modules.

# 5 Implementation

## 5.1 Main.hs and Help.hs

The file `Help.hs` provides basic help to the user; here we print out the help pages:

```
                          K     K      OOOOOO      RRRR        EEEEEEE     L
                          K    K      O      O     R    R      E           L
                          K   K       O      O     R     R     E           L
                          K  K        O      O     R     R     E           L
                          K K         O      O     R    R      E           L
 Welcome to the world of  KK          O      O     RRRR       EEEEEEE     L
                          K K         O      O     R  R       E           L
                          K  K        O      O     R   R      E           L
                          K   K       O      O     R    R     E           L
                          K    K      O      O     R     R    E           L
                          K     K      OOOOOO      R      R   EEEEEEE     LLLLLLL

 Copyright (c) Leicester University, 1997

 Please type help for help    ...    recursion is everywhere  ...


 >>help
help

 please type 'help 1' for a description of KOREL

 please type 'help 2' for the BNF grammars constituting KOREL programs

 please type 'help 3' to see some example programs

 please type 'help 4' to see how to use KOREL

 That's it folks! You're on yer own now ....
 >>help1
help1

 This Haskell program parses and interprets a language KOREL.
 The parser is combinatory. The interpreter is a coding of both evaluation
 and transition style operational semantics of the form:

 (eval) function declarations |- <state, program> ==> <state', value>

 (trans) function declarations |- <state, program> --> <state', program'>

 where ==> is -->^*
```

```
 A KOREL program is input as a string which can take the form:

 (i)        progexpn

 (ii)       declaration (| declaration)^* |- progexpn

 (iii)      < state , progexpn >

 (iv)       declaration (| declaration)^* |- < state , progexpn >

 see 'help 2' for BNF grammars for declarations, states and programs


 >>help2
help2

 The KOREL grammar for progexpn t is

 t ::= var | loc | true | false | z
     | t+t | t-t | t*t | t=t | t<=t | t>=t | t<t | t>t
     | %var -> t | #var -> t | t t^*
     | [] | t:t | t::t | [t, .. ,t] | {t, .. ,t} | hd t | tl t | el t
     | () | <t,t> | <<t,t>> | split t as var,var in t | fst t | snd t
     | inl t | inr t | case t of var.t or var.t
     | rec var.t
     | let var = t in t
     | if t then t else t
     | local var := t in t
     | skip | t ; t | while t do t | ref t | !t | t:=t

 where var :: AlphaString; loc=Li for i>=1; z :: int -- what a lot of choices!!

 The KOREL grammar for declaration d is

 d ::= var var^* = t

 The KOREL grammar for states s is

 s ::= [(Li, t)^*]      where i is a positive integer

 >>help3
help3

 type (i): [(% x -> x*3) (5*(6+9)) , (# x -> x*3) (5*(6+9))]

 type (ii): f x = if (x=1) then 1 else x*(f(x-1)) | g z = (z+8) |- (f4)*(g1)

 type (iii): < [(L1,2),(L3,<2,3>)] , ((L1:=(!L3)) ; (L3:= 7)) >

 type (iv): swap x y = (block L3:=(!x) in ((x:=(!y)) ; (y:=(!L3)))) |-
                                                  <[(L1,2),(L2,4)],swap L1 L2>


 >>help4
help4
```

```
PLEASE TYPE IN EITHER

'eval progam' OR 'trans program'

FOLLOWED BY 'return', where 'program' is your choice of (i) --> (iv) from help 2

>>
```

Let us consider `Main.hs`. As described above, this module provides a function called `trans` for which given a declaration, state and term, say `d, s, t`, (that is, a program `d |- <s, t>`), will produce each of the transition steps which result from the evaluation of the term to a value. The data are entered onto the command line in the form

```
trans d |- <s, t>
```

and the machine will print up the intermediate states and terms. The following example illustrates this:

```
trans f x y = x+y |- <[(L1,4)],(%z -> z+1)((f 1 2)+(!L1))>

[("L1","4")]    ((% z -> (z+1)) (((f 1) 2)+(!L1)))

[("L1","4")]    ((% z -> (z+1)) ((1+2)+(!L1)))

[("L1","4")]    ((% z -> (z+1)) (3+(!L1)))

[("L1","4")]    ((% z -> (z+1)) (3+4))

[("L1","4")]    ((% z -> (z+1)) 7)

[("L1","4")]    (7+1)

[("L1","4")]    8
```

`eval` is used in the same way as `trans`, except that it will simply return the final state and value, which are computed according to the rules defining the evaluation relation `==>`.

## 5.2 Lambda.hs and Syntax.hs

These modules contains the following type declarations:

```
-- ground types
type C = String         -- constants
type V = String         -- variables
type B = Bool           -- Booleans
type Z = Int            -- Integers
```

```
-- Terms (represented as expressions of an untyped meta-lambda calculus
--        with bound variables represented through de-Bruijn notation)
data Term = Const C | Free V | Bound Int | Abs (V,Term) | Apply (Term,Term)

type Loc = String               -- locations
type State = [(Loc,Term)]       -- state maps locations to terms
type Dec = [(V,[V],Term)]       -- declaration is d = (var, [var1 .. vara], t)
type Conf = (State,Term)        -- configuration is (s, t)
type Prog = (Dec,Conf)          -- program is (d, (s, t))
```

The types given above constitute the types of the language $\mathcal{H}$ into which $\mathbb{KOREL}$ is translated via the `read` function. Notice that $\mathcal{H}$ consists of some ground types, a datatype `Term`, and some further types which are essentially either pair or list types built from the ground types and `Term`. In fact there are a variety of read functions, each of which will map a particular feature (e.g. states, declarations etc) of $\mathbb{KOREL}$ into $\mathcal{H}$, and these are represented below, where $\xi$ denotes an arbitrary fragment of $\mathbb{KOREL}$.

$$\mathbb{KOREL} \xrightarrow{\texttt{read}} \mathcal{H}$$

$$\xi \longmapsto \xi^r$$

$$\texttt{s :: String} \xmapsto{\texttt{readstate}} \texttt{s}^r \texttt{ :: State}$$

$$\texttt{t :: String} \xmapsto{\texttt{readterm}} \texttt{t}^r \texttt{ :: Term}$$

$$\texttt{d :: String} \xmapsto{\texttt{readdec}} \texttt{d}^r \texttt{ :: Dec}$$

$$\texttt{d |- <s, t> :: String} \xmapsto{\texttt{readprog}} (\texttt{d}^r,(\texttt{s}^r,\texttt{t}^r)) \texttt{ :: Prog}$$

Before we give some concrete examples, the key feature to understand is how $\mathbb{KOREL}$ terms are translated into $\mathcal{H}$ terms. Once this is understood, the other translations are routine.

The type `Term` consists of terms of an un-typed lambda calculus into which $\mathbb{KOREL}$ terms are translated via `readterm`. In fact the translation is exactly that of the theory of arities and expressions, and we shall assume that the reader is familiar with this technique. If not, please see [NPS90], which provides a detailed explanation. However, we recall the salient points. In order to clarify the exposition, we shall define a small untyped lambda calculus $\mathcal{L}$ which provides a model of the datatype `Term`. The expressions of $\mathcal{L}$ are given by the grammar

$$t \quad ::= \quad c \quad | \quad u \quad | \quad (u.t) \quad | \quad (t\ t) \quad |$$

where $c$ and $u$ ranges over fixed sets of constants and variables. We shall use brackets in an informal fashion to delineate expression struture, and we shall write $t_1\ t_2\ \ldots t_k$ to denote multiple application (which of course associates to the left). The theory of arities and expressions uses such an untyped lambda calculus to represent the term trees corresponding to program syntax. Any particular term has a key name (for example the

term `while b do c` could be given the key name "while") which becomes a constant in $\mathcal{L}$ (eg *while*). We call *while* a *term constructor constant*. There are two sub terms, and so *while* has arity two. Then if `b` and `c` are represented by the $\mathcal{L}$ terms $b$ and $c$, the while loop is represented by $((while\ b)\ c) = while\ b\ c$. Variable binding in program syntax is represented uniformly by binding in $\mathcal{L}$. For example, `rec var.t` is represented by $rec\ (x.t)$ where $x$ represents `var`. Before giving some more examples, we comment that the datatype `Term` in $\mathcal{H}$ represents lambda expressions using de-Bruijn notation [Pau97]. Let us write the following abbreviations for certain term constructor constants:

| | |
|---|---|
| *plus* | for addition |
| *split* | for splitting of pairs |
| *pair* | for pairing |
| *ap* | for function application |
| *lam* | for lambda abstraction |

The $\mathbb{KOREL}$ term `2+x` would be denoted by the $\mathcal{L}$ expression $plus\,2\,x$, that is, $((plus\,2)\,x)$, which in $\mathcal{H}$ would be represented by

```
Apply (Apply (Const "plus",Const "2"),Free "x")
```

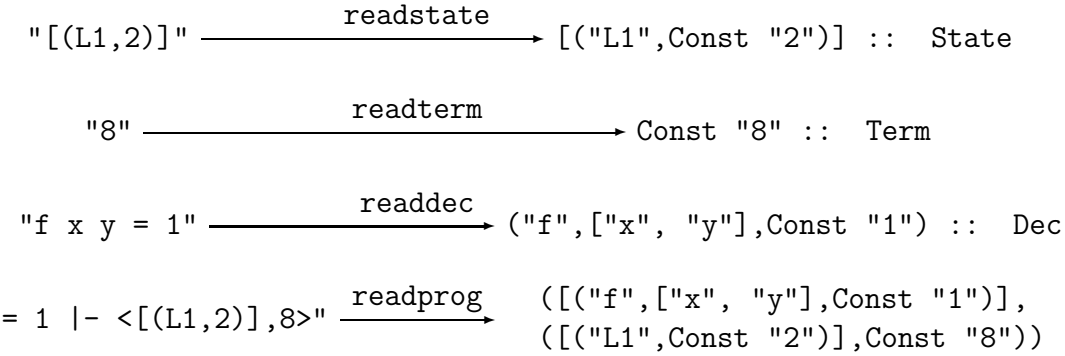
The term `(%z -> z+1)3` would be denoted by the expression $ap\ (lam\ (z.plus\ z\ 1))\ 3$ which in $\mathcal{H}$ would be represented by

```
Apply (Apply (Const "ap",
  Apply (Const "lam",Abs ("z",Apply (Apply (Const "plus",Bound 0),Const "1")))),
    Const "3")
```

Note the occurrence of `Bound 0` giving the de-Bruijn representation of the bound $z$. The $\mathbb{KOREL}$ term `split <2,3> as x,y in x` would be denoted by the $\mathcal{L}$ expression $split\ (pair\ 2\ 3)\ (x.y.x)$ which in $\mathcal{H}$ would be represented by

```
Apply (Apply (Const "split", Apply (Apply (Const "pair",Const "2"),Const "3")),
                                      Abs ("x",Abs ("y",Bound 1)))
```

Now that the translation of $\mathbb{KOREL}$ terms has been explained, it is quite easy to understand the translation of states, function declarations, and programs; here are some simple examples:




We give one final larger example. The $\mathbb{KOREL}$ program

```
f x y = x+y | g z = 1 |- <[(L1,4)],(%z -> z+1)((f 1 2)+(!L1))>
```

would be represented in $\mathcal{H}$ by `(d,(s,t)) :: Prog` where

```
d = [("f",["x", "y"],Apply (Apply (Const "plus",Free "x"),Free "y")), ("g",["z"],Const "1")]

s = [("L1",Const "4")]

t = Apply (Apply (Const "ap",
Apply (Const "lam",Abs ("z",Apply (Apply (Const "plus",Bound 0),Const "1")))),
Apply (Apply (Const "plus",
Apply (Apply (Const "ap",Apply (Apply (Const "ap",Free "f"),Const "1")),Const "2")),
Apply (Const "!",Const "L1"))
```

## 5.3 Scanner.hs, Parse.hs and Pare.hs

These three modules provide a lexer, and combinatory parser which implements the `read` functions. The methods employed are quite standard, and we make refer the reader to the literature on combinatory parsing, for example [Pau97]. We give just one example of a parsing function, to illustrate the general ideas. Consider the general form `var var1 .. vara = t` of a function definition, for example `f g x y = (g x) + (g y)`. The parser code is

```
dec = idr `next` many idr `next` key"=" `next` term `build`  makeDec
      where
      makeDec (f,(a,("=",e))) = (f,a,e)
```

which for the example would give output

```
("f",
["g", "x", "y"],
Apply (Apply (Const "plus",Apply (Apply (Const "ap",Free "g"),Free "x")),
Apply (Apply (Const "ap",Free "g"),Free "y")))
```

## 5.4 Transem.hs and Evalsem.hs

We explain the code in the file `Transem.hs`; that of `Evalsem.hs` is similar. The key function provided by `Transem.hs` is

```
ptrfn ::  Prog -> Conf
```

which mirrors the action of the function `trans`. It satisfies the following condition, that for all `d,s,t,s',t'`, then `d |- <s, t> --> <s', t'>` in $\mathbb{KOREL}$ if and only if `ptrfn` $(d^r,(s^r,t^r)) = (s'^r,t'^r)$ in $\mathcal{H}$. The function `ptrfn` is implemented in a structured fashion, centered around the fact that all $\mathbb{KOREL}$ terms are translated into $\mathcal{H}$ terms. In order to simplify our exposition, we shall often represent $\mathcal{H}$ terms using $\mathcal{L}$ notation, *without explanation*. First, we define

```
ptrfn (dec, (state, term)) = trfn (state,term)
              where
              trfn (s, t) = if isval (dec,t) then (s,t) else trfn' (s,t)
```

`ptrfn` calls `trfn` which first tests to see if the given `term` is a value in the declaration `dec`. If it is, computation stops. If not, `trfn'` is called which then computes the next configuration arising from the transition relation. Let us see how this is done. In general, the elements of type `Term` in which we have most interest will have the form

$$t \stackrel{\text{def}}{=} constr\ t_1\ t_2\ \ldots\ t_k$$

where *constr* is the name of a term constructor. If one examines the definition of the transition semantics, one sees that for a term t, it specifies that in any transition step, either

- one of the $t_i$ must be computed; or
- the constructor itself should be evaluated.

Which of these applies is determined by which of the `ti` are values. Thus `Transem.hs` provides functions of the form

$$trfnarityk\langle which\rangle\ \texttt{s cst t1 t2 ... tk z = <code>}$$

which will compute some of the `ti` to values (`<code>` determining exactly which, with $\langle which\rangle$ naming them) and then, once this is complete, will execute `z` (which corresponds to the constructor being evaluated). These details will now be clarified via an example, when $k = 2$

$$t \stackrel{\text{def}}{=} ap\ \underbrace{((lam\ (x.x))\ (lam\ (z.plus\ z\ 1)))}_{t_1}\ \underbrace{(plus\ 3\ 2)}_{t_2}$$

For example, consider $t$ under an eager semantics. First, $t_1$ must be computed, yielding

$$v_1 \stackrel{\text{def}}{=} lam\ (z.plus\ z\ 1)$$

Hence $t = ap\ v_1\ t_2$. Next $t_2$ is evaluated yielding $v_2 = 5$, with $t = ap\ v_1\ v_2$. Finally, with both $t_1$ and $t_2$ fully evaluated, we can compute the term constructor *ap*:

$$t = ap\ v_1\ v_2 = (plus\ z\ 1)[5/z] \stackrel{\text{def}}{=} plus\ 5\ 1 = 6.$$

For this example, we have

```
trfnarity2both s cst  t1  t2  z  =  if not (isval (dec,t1 ))
                                     then
                                          let (s',t1' ) = trfn (s,t1 )
                                          in (s', ap  t1'  t2 )
                                     else
                                          if not (isval (dec,t2 ))
                                          then
                                               let (s',t2' ) = trfn (s,t2 )
                                                in (s', ap  t1  t2' )
                                          else z
```

which will check whether $t1$ is a value, evaluating $t1$ if it is not, then repeat the process for $t2$ (thus *both* terms are evaluated to values), and finally execute `z`. We then have

```
trfn' (s, ap  t1  t2 )  =  trfnarity2both s ap  t1  t2  z          (*)
                           where
                           z = case t1  of lam x.t1'  -> t2[t1/x]
```

where the code `z` evaluates the constructor *ap* by performing a substitution. In the case when the function application is lazy, the above code at (*) would call `trfnarity2fst`, instead of `trfnarity2both`, thus evaluating just *t1* before performing the substitution, where

```
trfnarity2fst s cst  t1  t2  z  =  if not (isval (dec,t1 ))
                                   then
                                        let (s',t1' ) = trfn (s,t1 )
                                        in (s', cst  t1'  t2 )
                                   else z
```

# 6 The Type System and Operational Semantics

The type system, transition and evaluation style operational semantics of KOREL are presented on the following pages

$$\frac{}{E \vdash \texttt{var} :: \sigma}\ (\text{where } \texttt{var} :: \sigma \in E) \qquad \frac{}{E \vdash \texttt{z} :: \mathit{Int}} \qquad \frac{}{E \vdash \texttt{b} :: \mathit{Bool}}$$

$$\frac{E \vdash \texttt{t1} :: \mathit{Int} \qquad E \vdash \texttt{t2} :: \mathit{Int}}{E \vdash \texttt{t1 iop t2} :: \mathit{Int}} \qquad \frac{E \vdash \texttt{t1} :: \mathit{Int} \qquad E \vdash \texttt{t2} :: \mathit{Int}}{E \vdash \texttt{t1 bop t2} :: \mathit{Bool}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1> --> <s', t1'>}}{\texttt{d} \vdash \texttt{<s, t1 op t2> --> <s', t1' op t2>}} \qquad \frac{\texttt{d} \vdash \texttt{<s, t2> --> <s', t2'>}}{\texttt{d} \vdash \texttt{<s, z1 op t2> --> <s', z1 op t2'>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, z1 op z2> --> <s', z>}}\ (\texttt{z} \text{ is the result of } \texttt{z1 op z2})$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1> ==> <s', z1>} \qquad \texttt{d} \vdash \texttt{<s', t2> ==> <s'', z2>}}{\texttt{d} \vdash \texttt{<s, t1 op t2> ==> <s'', z>}}\ (\texttt{z} \text{ is the result of } \texttt{z1 op z2})$$

Table 1: The Semantics of *Int* and *Bool* Terms

$$\frac{E \vdash \texttt{t} :: \sigma_1}{E \vdash \texttt{inl t} :: \sigma_1 + \sigma_2} \qquad \frac{E \vdash \texttt{t} :: \sigma_2}{E \vdash \texttt{inr t} :: \sigma_1 + \sigma_2}$$

$$\frac{E \vdash \texttt{t} :: \sigma_1 + \sigma_2 \quad E\texttt{, var1} :: \sigma_1 \vdash \texttt{t1} :: \sigma \quad E\texttt{, var2} :: \sigma_2 \vdash \texttt{t2} :: \sigma}{E \vdash \texttt{case t of var1.t1 or var2.t2} :: \sigma}$$

---

$$\frac{\texttt{d} \vdash \texttt{<s, t> --> <s', t'>}}{\texttt{d} \vdash \texttt{<s, inl t> --> <s', inl t'>}} \qquad \frac{\texttt{d} \vdash \texttt{<s, t> --> <s', t'>}}{\texttt{d} \vdash \texttt{<s, inr t> --> <s', inr t'>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> --> <s', t'>}}{\texttt{d} \vdash \texttt{<s, case t of var1 t1 or var2 t2> --> <s', case t' of var1 t1 or var2 t2>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{case inl v of var1 t1 or var2 t2 --> t1[v/var]}}$$

$$\frac{}{\texttt{d} \vdash \texttt{case inr v of var1 t1 or var2 t2 --> t2[v/var]}}$$

---

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', v>}}{\texttt{d} \vdash \texttt{<s, inl t> ==> <s', inl v>}} \qquad \frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', v>}}{\texttt{d} \vdash \texttt{<s, inr t> ==> <s', inr v>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', inl v>} \quad \texttt{d} \vdash \texttt{<s', t1[v/var1]> ==> <s'', v'>}}{\texttt{d} \vdash \texttt{<s, case t of var1 t1 or var2 t2> ==> <s'', v'>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', inr v>} \quad \texttt{d} \vdash \texttt{<s', t2[v/var2]> ==> <s'', v'>}}{\texttt{d} \vdash \texttt{<s, case t of var1 t1 or var2 t2> ==> <s'', v'>}}$$

Table 2: The Semantics of $\sigma + \sigma'$ Terms

$$\frac{}{E \vdash \texttt{()} :: \mathit{unit}}$$

$$\frac{E \vdash \texttt{t1} :: \sigma_1 \quad E \vdash \texttt{t2} :: \sigma_2}{E \vdash \texttt{<t1, t2>} :: \sigma_1 \times \sigma_2} \qquad \frac{E \vdash \texttt{t1} :: \sigma_1 \quad E \vdash \texttt{t2} :: \sigma_2}{E \vdash \texttt{<<t1, t2>>} :: \sigma_1 \times \sigma_2}$$

$$\frac{E \vdash \texttt{t} :: \sigma_1 \times \sigma_2 \quad E, \texttt{ var1} :: \sigma_1, \texttt{ var2} :: \sigma_2 \vdash \texttt{t'} :: \sigma}{E \vdash \texttt{split t as var1,var2 in t'} :: \sigma}$$

---

$$\frac{\texttt{d} \vdash \texttt{<s, t1> --> <s', t1'>}}{\texttt{d} \vdash \texttt{<s, <t1,t2>> --> <s', <t1',t2>>}} \qquad \frac{\texttt{d} \vdash \texttt{<s, t2> --> <s', t2'>}}{\texttt{d} \vdash \texttt{<s, <v1,t2>> --> <s', <v1,t2'>>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1> --> <s', t1'>}}{\texttt{d} \vdash \texttt{<s, split t1 as var1, var2 in t2> --> <s', split t1' as var1, var2 in t2>}}$$

---

$$\frac{\texttt{d} \vdash \texttt{<s, split <v, v'> as var1, var2 in t2> --> <s, t2[v/var1,v'/var2]>}}{\texttt{d} \vdash \texttt{<s, split <<t, t'>> as var1, var2 in t2> --> <s, t2[t/var1,t'/var2]>}}$$

---

$$\frac{\texttt{d} \vdash \texttt{<s, t1> ==> <s', v1>} \quad \texttt{d} \vdash \texttt{<s', t2> ==> <s'', v2>}}{\texttt{d} \vdash \texttt{<s, <t1,t2>> ==> <s'', <v1,v2>>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1> ==> <s', <v1,v1'>>} \quad \texttt{d} \vdash \texttt{<s', t2[v1/var1, v1'/var2] > ==> <s'', v>}}{\texttt{d} \vdash \texttt{<s, split t1 as var1, var2 in t2> ==> <s'', v>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1> ==> <s', <<t1',t1''>>>} \quad \texttt{d} \vdash \texttt{<s', t2[t1'/var1, t1''/var2] > ==> <s'', v>}}{\texttt{d} \vdash \texttt{<s, split t1 as var1, var2 in t2> ==> <s'', v>}}$$

Table 3: The Semantics of $\sigma \times \sigma'$ Terms

$$\frac{}{E \vdash \texttt{[]} :: \mathit{Lists}(\sigma)}$$

$$\frac{E \vdash \texttt{t1} :: \sigma \quad E \vdash \texttt{t2} :: \mathit{Lists}(\sigma)}{E \vdash \texttt{t1:t2} :: \mathit{Lists}(\sigma)} \qquad \frac{E \vdash \texttt{t1} :: \sigma \quad E \vdash \texttt{t2} :: \mathit{Lists}(\sigma)}{E \vdash \texttt{t1::t2} :: \mathit{Lists}(\sigma)}$$

$$\frac{E \vdash \texttt{ti} :: \sigma \quad (1 \leq i \leq n)}{E \vdash \texttt{[t1,t2, .. ,tn]} :: \mathit{Lists}(\sigma)} \qquad \frac{E \vdash \texttt{ti} :: \sigma \quad (1 \leq i \leq n)}{E \vdash \{\texttt{t1,t2, .. ,tn}\} :: \mathit{Lists}(\sigma)}$$

$$\frac{E \vdash \texttt{t} :: \mathit{Lists}(\sigma)}{E \vdash \texttt{hd t} :: \sigma} \qquad \frac{E \vdash \texttt{t} :: \mathit{Lists}(\sigma)}{E \vdash \texttt{tl t} :: \mathit{Lists}(\sigma)} \qquad \frac{E \vdash \texttt{t} :: \mathit{Lists}(\sigma)}{E \vdash \texttt{el t} :: \mathit{Bool}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1> --> <s', t1'>}}{\texttt{d} \vdash \texttt{<s, t1:t2> --> <s', t1':t2>}} \qquad \frac{\texttt{d} \vdash \texttt{<s, t2> --> <s', t2'>}}{\texttt{d} \vdash \texttt{<s, v1:t2> --> <s', v1:t2'>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, ti> --> <s', ti'>} \quad (j+1 \leq i \leq a)}{\texttt{d} \vdash \texttt{<s, [v1,v2, .. ,vj, t(j+1), .. ,ta]> --> <s', [v1,v2, .. ,vj, t(j+1)', .. ,ta]>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> --> <s', t'>}}{\texttt{d} \vdash \texttt{<s, hd t> --> <s', hd t'>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, hd v:v'> --> <s, v>}} \qquad \frac{}{\texttt{d} \vdash \texttt{<s, hd t::t'> --> <s, t>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> --> <s', t'>}}{\texttt{d} \vdash \texttt{<s, tl t> --> <s', tl t'>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, tl v:v'> --> <s, v'>}} \qquad \frac{}{\texttt{d} \vdash \texttt{<s, tl t::t'> --> <s, t'>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> --> <s', t'>}}{\texttt{d} \vdash \texttt{<s, el t> --> <s', el t'>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, el t:t'> --> <s, false>}} \qquad \frac{}{\texttt{d} \vdash \texttt{<s, el t::t'> --> <s, false>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, el []> --> <s, true>}}$$

Table 4: The Semantics of $\mathit{Lists}(\sigma)$ Terms

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', v>} \quad \texttt{d} \vdash \texttt{<s', t'> ==> <s'', v'>}}{\texttt{d} \vdash \texttt{<s, t:t'> ==> <s'', v:v'>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', v:v'>}}{\texttt{d} \vdash \texttt{<s, hd t> ==> <s', v>}} \qquad \frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', v:v'>}}{\texttt{d} \vdash \texttt{<s, tl t> ==> <s', v'>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', []>}}{\texttt{d} \vdash \texttt{<s, el t> ==> <s', true>}} \qquad \frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', v:v'>}}{\texttt{d} \vdash \texttt{<s, el t> ==> <s', false>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', t'::t''>}}{\texttt{d} \vdash \texttt{<s, el t> ==> <s', false>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', t'::t''>} \quad \texttt{d} \vdash \texttt{<s', t'> ==> <s'', v>}}{\texttt{d} \vdash \texttt{<s, hd t> ==> <s'', v>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', t'::t''>} \quad \texttt{d} \vdash \texttt{<s', t''> ==> <s'', v>}}{\texttt{d} \vdash \texttt{<s, tl t> ==> <s'', v'>}}$$

Table 5: The Semantics of $Lists(\sigma)$ Terms, Continued

$$\frac{}{E \vdash \texttt{<s,Lk>} :: Ref(\sigma)}\ (\texttt{s(Lk)} \text{ defined } \&\ E \vdash \texttt{s(Lk)}::\sigma) \qquad \frac{E \vdash \texttt{t} :: \sigma}{E \vdash \texttt{ref t} :: Ref(\sigma)} \qquad \frac{E \vdash \texttt{t} :: Ref(\sigma)}{E \vdash \texttt{!t} :: \sigma}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> --> <s', t'>}}{\texttt{d} \vdash \texttt{<s, ref t> --> <s', ref t'>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, ref v> --> <s\{Lk -> v\}, Lk>}}\ (\text{where the look-up of } \texttt{Lk} \text{ is undefined in } \texttt{s})$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> --> <s', t'>}}{\texttt{d} \vdash \texttt{<s, !t> --> <s', !t'>}} \qquad \frac{}{\texttt{d} \vdash \texttt{<s, !Lk> --> <s, s(Lk)>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', v>}}{\texttt{d} \vdash \texttt{<s, ref t> ==> <s'\{Lk -> v\}, Lk>}}\ (\text{where the look-up of } \texttt{Lk} \text{ is undefined in } \texttt{s})$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', Lk>}}{\texttt{d} \vdash \texttt{<s, !t> ==> <s', s'(Lk)>}}\ (\text{where the look-up of } \texttt{Lk} \text{ is defined in } \texttt{s'})$$

Table 6: The Semantics of $Ref(\sigma)$ Terms

$$\frac{E,\ \texttt{var} :: \sigma_1 \vdash \texttt{t} :: \sigma_2}{E \vdash \texttt{\%var -> t} :: \sigma_1 \to \sigma_2} \qquad \frac{E,\ \texttt{var} :: \sigma_1 \vdash \texttt{t} :: \sigma_2}{E \vdash \texttt{\#var -> t} :: \sigma_1 \to \sigma_2} \qquad \frac{E \vdash \texttt{t2} :: \sigma_1 \to \sigma_2 \quad E \vdash \texttt{t1} :: \sigma_1}{E \vdash \texttt{t1 t2} :: \sigma_2}$$

---

$$\frac{\texttt{d} \vdash \texttt{<s, t1> --> <s', t1'>}}{\texttt{d} \vdash \texttt{<s, t1 t2> --> <s', t1' t2>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t2> --> <s', t2'>}}{\texttt{d} \vdash \texttt{<s, (\%var -> t1) t2> --> <s', (\%var -> t1) t2'>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, (\%var -> t) v> --> <s', t[v/var]>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, (\#var -> t1) t2> --> <s', t2[t1/var]>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t(j+1)> --> <s', t(j+1)'>} \qquad (j+1 \le i \le a)}{\texttt{d} \vdash \texttt{<s, f v1 v2 .. vj t(j+1) .. ta> --> <s', f v1 v2 vj t(j+1)' .. ta>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, f v1 v2 .. va> --> <s, t[v1/var1,v2/var2, .. ,va/vara]>}} \text{ (where } \texttt{f v1 v2 .. va = t} \text{ in } \texttt{d})$$

---

$$\frac{\texttt{d} \vdash \texttt{<s, t1> ==> <s', \%var -> t1'>} \quad \texttt{d} \vdash \texttt{<s', t2> ==> <s'', v2>}}{\texttt{d} \vdash \texttt{<s, t1 t2> ==> <s'', t1'[v2/var]>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1> ==> <s', \#var -> t1'>}}{\texttt{d} \vdash \texttt{<s, t1 t2> ==> <s', t1'[t2/var]>}}$$

Table 7: The Semantics of $\sigma \to \sigma'$ Terms

$$\frac{}{E \vdash \texttt{skip} :: Cmd} \qquad \frac{E \vdash \texttt{t1} :: Bool \quad E \vdash \texttt{t2} :: Cmd}{E \vdash \texttt{while t1 do t2} :: Cmd}$$

$$\frac{E \vdash \texttt{t1} :: Ref(\sigma_1) \quad E \vdash \texttt{t2} :: \sigma_1}{E \vdash \texttt{t1:=t2} :: Cmd} \qquad \frac{E \vdash \texttt{t1} :: \sigma_1 \quad E \vdash \texttt{t2} :: Cmd}{E \vdash \texttt{local var:=t1 in t2} :: Cmd}$$

---

$$\frac{}{\texttt{d} \vdash \texttt{<s, while t1 do t2>} \texttt{ --> } \texttt{<s, if t1 then (t2 ; while t1 do t2) else skip>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1>} \texttt{ --> } \texttt{<s', t1'>}}{\texttt{d} \vdash \texttt{<s, t1:=t2>} \texttt{ --> } \texttt{<s', t1':=t2>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t2>} \texttt{ --> } \texttt{<s', t2'>}}{\texttt{d} \vdash \texttt{<s, Lk:=t2>} \texttt{ --> } \texttt{<s', Lk:=t2'>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, Lk:=v>} \texttt{ --> } \texttt{<s\{Lk -> v\}, skip>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1>} \texttt{ --> } \texttt{<s', t1'>}}{\texttt{d} \vdash \texttt{<s, local var:=t1 in t2>} \texttt{ --> } \texttt{<s', local var:=t1' in t2>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, local var:=v1 in t2>} \texttt{ --> } \texttt{<s\{Lk->v1\}, local* Lk t2[Lk/var]>}} \ (\texttt{Lk} \notin \texttt{s or t2})$$

$$\frac{\texttt{d} \vdash \texttt{<s, t2>} \texttt{ --> } \texttt{<s', t2'>}}{\texttt{d} \vdash \texttt{<s, local* Lk t2>} \texttt{ --> } \texttt{<s', local* Lk t2'>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, local* Lk v2>} \texttt{ --> } \texttt{<s}\{\texttt{Lk}\}^{-1}\texttt{, v2>}}$$

---

$$\frac{\begin{cases} \texttt{d} \vdash \texttt{<s, t1>} \texttt{ ==> } \texttt{<s', true>} \\ \texttt{d} \vdash \texttt{<s', t2>} \texttt{ ==> } \texttt{<s'', skip>} \\ \texttt{d} \vdash \texttt{<s'', while t1 do t2>} \texttt{ ==> } \texttt{<s''', skip>} \end{cases}}{\texttt{d} \vdash \texttt{<s, while t1 do t2>} \texttt{ ==> } \texttt{<s''', skip>}}$$

$$\frac{\begin{cases} \texttt{d} \vdash \texttt{<s, t1>} \texttt{ ==> } \texttt{<s', v1>} \\ \texttt{d} \vdash \texttt{<s'\{Lk -> v1\}, t2[Lk/var]>} \texttt{ ==> } \texttt{<s'', skip>} \end{cases}}{\texttt{d} \vdash \texttt{<s, local var:=t1 in t2>} \texttt{ ==> } \texttt{<s''}\{\texttt{Lk}\}^{-1}\texttt{, skip>}} \ (\texttt{Lk} \notin \texttt{s or t2})$$

Table 8: The Semantics of *Cmd* Terms

$$\frac{E \vdash \texttt{t1} :: \sigma_1 \quad E \vdash \texttt{t2} :: \sigma_2}{E \vdash \texttt{t1 ; t2} :: \sigma_2}$$

$$\frac{E \vdash \texttt{t} :: \mathit{Bool} \quad E \vdash \texttt{t1} :: \sigma \quad E \vdash \texttt{t2} :: \sigma}{E \vdash \texttt{if t then t1 else t2} :: \sigma} \qquad \frac{E \vdash \texttt{t1} :: \sigma_1 \quad E, \texttt{ var} ::\sigma_1 \vdash \texttt{t2} :: \sigma_2}{E \vdash \texttt{let var=t1 in t2} :: \sigma_2}$$

---

$$\frac{\texttt{d} \vdash \texttt{<s, t1> --> <s', t1'>}}{\texttt{d} \vdash \texttt{<s, t1 ; t2> --> <s', t1' ; t2>}} \qquad \frac{}{\texttt{d} \vdash \texttt{<s, v1 ; t2> --> <s, t2>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> --> <s', t'>}}{\texttt{d} \vdash \texttt{<s, if t then t1 else t2> --> <s', if t' then t1 else t2>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, if true then t1 else t2> --> <s, t1>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, if false then t1 else t2> --> <s, t2>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, rec var.t> --> <s, t[rec var.t/var]>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1> --> <s', t1'>}}{\texttt{d} \vdash \texttt{<s, let var = t1 in t2> --> <s', let var = t1' in t2>}}$$

$$\frac{}{\texttt{d} \vdash \texttt{<s, let var = v1 in t2> --> <s, t2[v1/var]>}}$$

---

$$\frac{\texttt{d} \vdash \texttt{<s, t1> ==> <s', v1>} \quad \texttt{d} \vdash \texttt{<s', t2> ==> <s'', v2>}}{\texttt{d} \vdash \texttt{<s, t1 ; t2> ==> <s'', v2>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', true>} \quad \texttt{d} \vdash \texttt{<s', t1> ==> <s'', v>}}{\texttt{d} \vdash \texttt{<s, if t then t1 else t2> ==> <s'', v>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t> ==> <s', false>} \quad \texttt{d} \vdash \texttt{<s', t2> ==> <s'', v>}}{\texttt{d} \vdash \texttt{<s, if t then t1 else t2> ==> <s'', v>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t[rec var.t/var]> ==> <s', v>}}{\texttt{d} \vdash \texttt{<s, rec var.t> ==> <s', v>}}$$

$$\frac{\texttt{d} \vdash \texttt{<s, t1> ==> <s', v1>} \quad \texttt{d} \vdash \texttt{<s', t2[v1/var]> ==> <s'', v2>}}{\texttt{d} \vdash \texttt{<s, let var = t1 in t2> --> <s'', v2>}}$$

Table 9: The Semantics of Remaining Terms

# References


[Cro95a] R. L. Crole. Functional Programming Theory, 1995. Department of Mathematics and Computer Science Lecture Notes, LaTeX format iv+68 pages with index.

[Cro95b] R. L. Crole. Semantics of Programming Languages, 1995. Department of Mathematics and Computer Science Lecture Notes, LaTeX format iii+97 pages with subject and notation index.

[Cro97] R. L. Crole. The KOREL Programming Language (Preliminary Report). Technical Report 1997/43, Department of Mathematics and Computer Science, University of Leicester, 1997.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[HS88] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and the Lambda Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1988.

[Kah88] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. Elsevier Science Publishers B.V. North Holland, 1988.

[MT91] R. Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Mass., 1991.

[MTH90] R. Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *Monographs on Computer Science*. Oxford University Press, 1990.

[Pau97] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1997. 2nd Edition.

[Pit90] A. M. Pitts. Notes on the translation of simply typed lambda calculus into the computational lambda calculus. Cambridge Computer Laboratory Notes, 1990.

[Pit97a] A. M. Pitts. Lecture notes on semantics of programming languages. Undergraduate Lecture Course, Cambridge University Computer Laboratory, 1997.

[Pit97b] A. M. Pitts. Lecture notes on types. Undergraduate Lecture Course, Cambridge University Computer Laboratory, 1997.

[Plo75] G.D. Plotkin. Call by name, call by value and the $\lambda$ calculus. *Theoretical Computer Science*, 1:125–129, 1975.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI–FN 19, Department of Computer Science, University of Aarhus, Denmark, 1981.

[Sch94] D. A. Schmidt. *The Structure of Typed Programming Languages*. Foundations of Computing Series. MIT Press, Cambridge, Mass., 1994.

[Tho19] S. Thompson. *The Craft of Functional Programming*. ??, 19??

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1993.