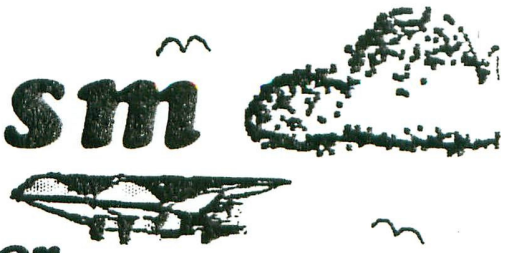
Volume I, Number 1 January, 1983

Polymorphism



The ML/LCF/Hope Newsletter

Contents

Letter from the Editors

Welcome to the second issue of Polymorphism, which is the first to contain technical contributions. In this issue, Robin Milner reports on the history of the development of LCF and ML, Ravi Sethi provides an analysis of the syntax of ML as given in "Edinburgh LCF" (from the standpoint of an LR/1 parser generator like YACC), Luca Cardelli defines his functional abstract machine, which is the basis of his ML compiler, and Chris Wadsworth provides a report on the November SERC Software Initiative Meeting on LCF/ML/Hope with an attached position paper by Mike Gordon and Larry Paulson, who were unable to attend the meeting.

These articles provide a good start, but they also more or less empty the queue of contributions. Future issues will contain installments of a revised manual for VAX ML and a Hope manual (for Franz Hope?), but we look forward to receiving further "external" contributions.

By the way, the convention in this newsletter is to use "VAX ML" to refer to Luca Cardelli's Pascal implementation for the VAX, rather than "Cardelli ML" as in Wadsworth's report or "Luca ML" as in Gordon and Paulson's letter. However, we still face the problem of distinguishing between the various Lisp based LCF/ML variants from Edinburgh, Gothenburg, and INRIA/Cambridge. Any suggestions for a naming scheme?

One suggestion to our European contributors: when using A4 paper, please try to leave ample top and bottom margins so that the text will fit on the American standard 8.5 x 11 inch (29.6 x 28 cm) sheets when photocopied.

A correction to the list of sites running VAX ML: Chris Wadsworth is not going to be porting VAX ML to the Perq, but there are rumors that it might be done at Edinburgh.

So long till the next issue, and keep those cards and letters coming!

Luca Cardelli
David MacQueen

Bell Laboratories
Murray Hill, NJ 07974
USA

How ML evolved

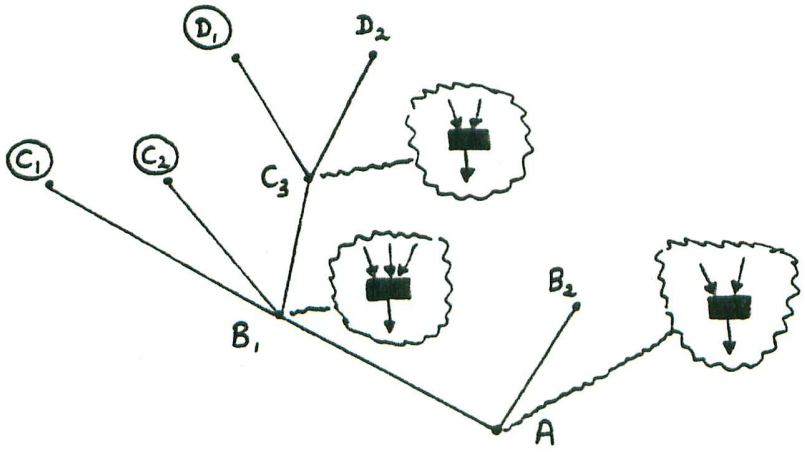
Robin Milner, Edinburgh University, November 1982.

ML is one among many functional programming languages. But not many were designed, as ML was, for a more-or-less specific task. The point of this note is to summarise the process by which we were guided to ML, as it now is, by the demands of the task. We (at least I) feel that to find a good metalanguage for machine assisted proof, which was the task, we could hardly have gone in an essentially different direction; the task seemed to determine the language - and even made it turn out to be a general purpose language!

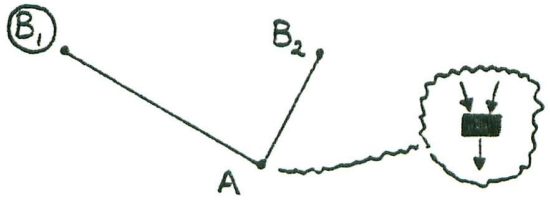
The context in 1974 (when the Edinburgh LCF project began) was our experience with the Stanford LCF proof assistant, developed there in 1971-72 by Richard Weyrauch, Malcolm Newey and myself. The development of ML as a metalanguage for interactive proof was the work of several people; in chronological order they were (besides me) Malcolm Newey, Lockwood Morris, Michael Gordon, and Christopher Wadsworth. Other people, who joined the LCF project after the language was more-or-less fixed, and used it for proofs, are Avra Cohn, Jacek Leszczylowski, David Schmidt, Larry Paulson and Brian Monahan. In the following summary of the development process, the actual logic involved (PPLAMBDA) is rather irrelevant, and it now seems that the same principles apply to any formal deductive system.

Consider the activity of goal-directed proof. If we have a goal A, a logical formula to be proved, then it is sound to replace it by subgoals $B_1,\ldots,B_n$, provided we know how to construct, from achievements of all the B_i, an achievement of A. We may call any subgoaling method a tactic ; it is a valid tactic, then, if it also provides - for each set $\{B_i\}$ of subgoals produced from a goal A - a way of extending achievements of the B_i into an achievement of A, and this "way" we call a validation. Of course, only valid tactics are useful!

Now, given a fixed repertoire of tactics, a natural proof assistant would be one which maintains a goal tree , with the user always working at a leaf. The Stanford LCF system was like this. After some subgoaling, the tree might look thus:

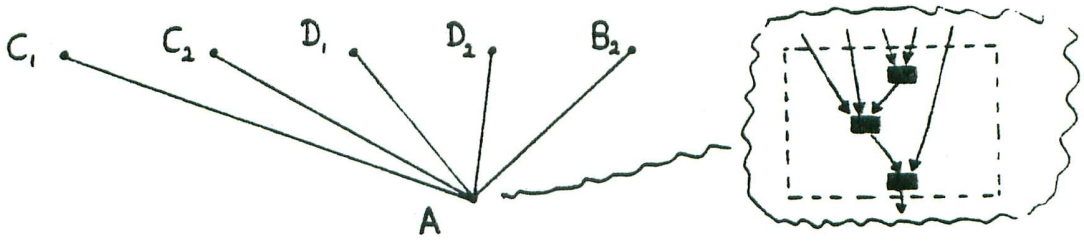


Here, we suppose that the ringed subgoals have (somehow) been achieved; the little black boxes sitting at the non-leaf nodes are the validations waiting to be applied to the (achieved) sons to achieve their father. So, after achieving D_2 the proof assistant would collapse the tree to:



Clearly, under this rigid discipline of tactical tree walking, the only way of proving the main goal is to return to the root, and the only doubt that the theorem has been correctly proved is in the correctness of the built-in programs for the basic repertoire of tactics and for treewalking.

But in a more flexible system, a user should at least be allowed to compose tactics into more powerful ones. For example, he should be allowed to compose the three tactics which were applied to goals A, B_1 and C_3 to produce the tree of our first diagram; this composite tactic applied to goal A would yield a flatter tree



Note that the validation produced by the composite tactic is a particular combination of those produced by the separate tactics. (Of course, as in our first diagrams, some of the subgoals may then be somenow achieved; we have omitted the rings here).

Well, to program this composition the user must be allowed to hold in his hand as objects (or: be allowed to bind as values to metavariables) both goals and validations; moreover, to put them together properly already suggests the need for structure-processing power of the kind found in LISP and other functional programming languages.

What kind of object is a validation? It is a "way of extending the achievements of subgoals into an achievement of the goal". But an achievement is (in our case) a proof, or the theorem which is the last step of a proof; so the validation for a tactic which produces subgoals $\{B_i\}$ from goal A could perhaps be represented by the theorem $\vdash B_1 \supset \ldots \supset B_n \supset A$, and to apply the validation is perhaps just to apply Modus Ponens repeatedly to this theorem and the achievements $\vdash B_1, \ldots, \vdash B_n$, to produce $\vdash A$. So perhaps validations are just theorems, proved somehow at the time that the tactic is applied?

To see that this is wrong, consider the tactic which converts a goal formula $\forall x B(x)$ into a single subgoal formula, $B(x)$ (this is the common method of proving that something holds for all x by proving that it holds for arbitrary x). According to the above suggestions then, the validation should be the theorem $\vdash B(x) \supset \forall x\ B(x)$, and there is no such theorem ! In fact the validation should not be a theorem, but a function from theorems to theorems, i.e. a (primitive or derived) inference rule; in our case, it is the rule of generalisation

$$\text{GEN}\quad \frac{\vdash B(x)}{\vdash \forall x B(x)}$$

(This is why we called the tactic GENTAC)

We immediately see that a tactic is a function producing function; when applied to a goal it produces, as well as a subgoal list, a function which is the validation. So our metalanguage must express second order functions.

Further:

(1) when the validation function depends (as it may in general) upon the properties of the goal attacked, these properties will be bound into the validation at the time of tactic application, and the natural way of doing this critically requires the static binding convention, now normally accepted in preference to the dynamic binding of LISP. (In Landin's terms, the validation is a closure, i.e. an expression paired with an environment).

(2) Since the user is to be allowed to compose tactics (second order functions), his compositions will be third order functions; clearly a metalanguage which expresses functions of arbitrarily high order is the only natural choice.

(3) Since the user is to be allowed to hold validations (and, in general, any primitive or derived inference rule) in his hand, it is critically important that he is only allowed to apply them to theorems, not to other objects (such as formulae) which look so like theorems that in a moment of misguided inspiration he may mistake one for the other! So the metalanguage must be rigorously typed, in a way which at least distinguishes theorems from other things.

(4) For a given tactic T, there are usually goals for which it makes no sense to apply T (Example: it makes no sense to apply GENTAC to a goal formula which is not universally quantified). It would be vastly inconvenient to test a goal by some separate predicate before applying a tactic, so the tactic itself must assume the task of detecting inapplicability, and respond in some suitable way. But in a typed language, every result of applying a tactic must be a goal list paired with a validation, and it is irksome to have to construct a correctly typed but spurious 'result' when the application makes no sense; so the only alternative in this case is to have no result at all. Hence it is natural to have an escape or failure mechanism, under which senseless applications may avoid producing a result, but instead be detected for alternative action. In LISP this response could be to return the result NIL, for example.

(5) In a conversational typed functional language, it soon appeared intolerable to have to declare - for example - a new maplist function for mapping a function over a list, every time a new type of list is to be treated. Even if the maplist function could possess what Strachey called "parametric polymorphism" in an early paper, it also appeared intolerable to have to supply an appropriate type explicitly as a parameter, for each use of this function. (Perhaps this latter is rendered more acceptable if types can be suitably abbreviated by names, but note that even simple list constructors and destructors - CONS, CAR, CDR or whatever - would need explicit type parameters!) So a polymorphic type discipline, with rigorous type checking, emerges as the most natural solution. Note that it emerges as such on purely practical considerations; it is a gift from the Gods that this discipline happens to have a simple semantic theory, and that the type checking has an elegant implementation based upon unification (Robinson). We only discovered afterwards that the proper lineage for this type checking is from Curry's functionality, through Roger Hindley's principal type schemes.

This discussion was somewhat simplified w.r.t. LCF (for example, LCF goals are not simply logical formulae to be proved). But it is, in essence, the process by which we arrived almost unavoidably at the metalanguage ML as it now exists. It shows why ML is a higher order functional programming language with rigorous polymorphic type discipline and an escape mechanism (and, of course, static binding).

Perhaps the development of ML has been made to seem too clean and trouble-free, from the above discussion. There still remain, however, some big problems and question marks. It is important to mention some of them; in doing so, we also place ML in context with other languages - both applicative and imperative.

(1) It is natural to recover, within ML, the simple treewalking proof methodology with which we started. But this global tree is a changing structure; as such, it cannot really be implemented without strain in a purely applicative language. (Or so it appears to me; this is a challenge to applicative language devotees who wish to rule out state change completely.) So ML has an assignment statement!

But this does not sit so easily beside the polymorphic type discipline. Recent work by Luis Damas (forthcoming Ph.D.) shows that the somewhat over-rigid treatment of the types of assignable variable in ML (viz. that they may not be polymorphic) can be relaxed; but the purity and obviousness of the discipline is inevitably lost.

(2) Even without assignment, the type discipline forced us to adopt a restrained form of escape mechanism in ML; it is not allowed to escape (or fail) with a value of arbitrary type, but only with a token. This problem has to do with the dynamic nature of the escape-trapping mechanism; the trap for each escape is not textually determined, and it appeared to us most useful that it should not be so. A clean solution to this problem - probably easier than that for assignable variables - would be an important development.

(3) ML does not adopt the clausal form of function definition, which is found so convenient by users of HOPE and PROLOG. How can we get a semantically rigorous form of this clausal definition, in which the constructor-patterns in formal parameters can involve not only primitive constructors, but also the constructors of user-defined abstract types? The problem is to know that these constructors are constructors, in the sense of being uniquely decomposable (or else to admit nondeterminism into the language). If I understand Rod Burstall right, this is partly why HOPE is called HOPE; the answer may eventually become CLEAR.

(4) ML does not use lazy evaluation; it calls by value. This was decided for no other reason than our inability to see the consequences of lazy evaluation for debugging (remember that we wanted a language which we could use rather than research into), and the interaction with the assignment statement, which we kept in the language for reasons already mentioned. In fact, this sharpens the challenge mentioned in (1) above; is there a good language in which lazy evaluation and controllable state-change sit well side-by-side? John Reynolds has for a long time worried about such possible incompatabilities between applicative and imperative languages (cf. his "Syntactic Control of Interference", which exposes the problem with great honesty).

Conclusion. I hope this short essay has shown that machine-assisted proof provided a beautifully appropriate focus for developing functional programming and demonstrating its importance. It would be very useful for others to use this newsletter as a medium for reporting other real exercises in functional programming, so that a balance is kept between the seductive purity of functional languages and the methodology of their use. The remarks above tried to point out the considerable tension that exists between these two aspects of programming, and to show that it is not at all trivial to resolve the tension.

Unambiguous syntax for ML

Ravi Sethi

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

ML is a functional language that was designed as part of the Edinburgh LCF effort. Since several geographically distributed groups are now working with ML, the need for an unambiguous syntax arises. A machine checked syntax based on that of the initial Lisp implementation is given. The syntax differs from that of the Pascal implementation.

November 29, 1982

Unambiguous syntax for ML

Ravi Sethi

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

"ML is a functional language in the tradition of ISWIM and GEDANKEN [3]."

While ML began as the metalanguage for conducting proofs in Edinburgh LCF, it is a programming language in its own right. ML as initially implemented in Lisp is described in [3]. For most practical purposes, the syntax in [3] is perfectly adequate. However, the ambiguities in it have to be identified and resolved afresh by anyone interested in developments based on it. Several clarifications and extensions already appear in the Pascal implementation [2]. It appears that existing syntax analyzers for ML are hand coded, and that grammars characterizing the languages accepted by these analyzers are not available. It is the purpose of this paper to adapt the syntax in [3] so that it can be used to generate parsers. As far as possible, ambiguities will be resolved following [2]. Some comments on the treatment of user defined infix operators appear in Section 6.

The syntax is described in a form suitable for processing by Yacc [4], the parser generator distributed with the UNIX™ operating system. Some of Yacc's conventions will be reviewed here to make it easier to read this paper. A terminal symbol in the grammar is either a single character enclosed between `''`, e.g. `'#'`, or an identifier declared in a `%token` declaration, e.g.

```
%token          ARR BOOL INT
```

Associativity and precedence of operators (in expressions) can be indicated using `%left` and `%right` as in:

```
%right          ARR
%right          '+'
%right          '#'
```

All identifiers on a `%left` or `%right` line have the same precedence (i.e. binding power) and associate to the left or right, respectively. Successive `%left` and `%right` lines indicate increasing precedence; e.g. `'#'` has higher precedence than `'+'` and `ARR`.

The notation for syntactic rules is very similar to that introduced by Backus [1]; e.g.

```
type    : type ARR type         /* R   continuous function     */
        | type '+' type         /* R   disjoint sum            */
        | type '#' type         /* R   product                 */
        | s_type                /*     simple type             */
        ;
```

By convention, rules containing operators will be in order of increasing precedence of the operators. (Regretfully, this convention is the opposite of the one in [3]). Comments will be enclosed between `/*` and `*/` and a leading `L` or `R` in the comment indicates left or right associativity,

respectively.

The machine checked syntax is attached as an appendix. In the body of the paper, we will relax the assumption that a single character appear between ``, e.g. `->` will be allowed.

2. Types

Each object in ML has a type, with the basic types being

```
basic_type
        : '.'                   /*      type of ()              */
        | 'bool'
        | 'int'
        | 'token'               /*      characters between ``   */
        ;
```

The only object of type `.` is the expression `()` called *empty*. (Luca Cardelli points out that there is a lot of confusion about the names of `()`, `[]`, and `.`. In [2], `()` is called the *triv value*, `.` the *triv type*, and `[]` the *nil list*.) Integers and booleans are as expected, with tokens consisting of a sequence of characters enclosed between ``.

Given an object of some type, e.g. `int`, it is possible to apply the type operator `list` to define a list of elements of the type, e.g. `int list`. As suggested by this example, type operators are postfix, and their syntax is given by the following fragment from [3]:

```
s_type    : type_arg ID |  ...

type_arg: s_type          /*      single type argument    */
        | '(' type ',' ... ',' type ')'
```

Since `s_type` can also generate a parenthesized `type`, the above syntax is ambiguous, since both productions for `type_arg` can generate, say `(int)`. The ambiguity can be eliminated simply by insisting that the second production for `type_arg` generate two or more types in parentheses.

3. Declarations

When expressions get large, or when there are common subexpressions, it is convenient to define subsidiary expressions using the `let` construct. For example, the following expressions are equivalent

```
let z = a+b+c in z * z * z
(a+b+c)*(a+b+c)*(a+b+c)
```

Subsidiary expressions may also be used to define functions. Note for example the use of a squaring function in the following expression for b^6

```
let     sq(y) = y*y
in      sq ( b * sq(b) )
```

In ML, the `let` part of the above expressions can appear (without the `in` part) as a *declaration*. Phrases like `z=a+b+c` and `sq(y)=y*y` are called *bindings*. Formal parameters like `y` and `z` are called *varstructs*.

It is convenient to allow varstructs to have some structure, e.g. pairs make it easier to deal with functions of two arguments. In addition types can be associated with varstructs as suggested by `v:type`.

Type constraints using `:type` lead to an ambiguity that can be illustrated using:

```
let f a : type = exp
```

The syntax in [3] admits the two distinct parses suggested by the parenthesizations

```
f (a:type)
(f a):type
```

In the former case, `type` is associated with the argument `a`, while in the latter case it is associated with the result of applying `f` to `a`.

A similar ambiguity involving type constraints arises with curried functions like

```
let g x y = exp
```

Now add a type constraint for `x`:

```
let g x : t y = exp
```

At this point there is an ambiguity, because `(t y)` is a legal type expression – recall the discussion of type arguments in Section 2.

The ambiguity in the above examples can be resolved by requiring that type constraints be parenthesized. However, we go further and insist that a varstruct in a function definition be parenthesized if it is not an identifier. This choice is motivated by examples like

```
let     f a,b = exp
```

and

```
        f a,b = exp
```

According to the syntax in [3], the subexpression `f a,b=exp` is parsed differently in declarations and expressions. The reason is that juxtaposition for function application binds very tightly in expressions, but in declarations juxtaposition is weaker than operators like `','`. By insisting that a varstruct in a function definition be parenthesized if it is not an identifier, we treat expressions and declarations uniformly. Incidentally, following [3], the above strings are parsed as:

```
let     f(a,b) = exp
        ( (f a), (b = exp) )
```

The syntax of varstructs uses three nonterminals `var1`, `var2`, and `var3` to set the precedence properly. As an aside, the suffix in `var3.p`, motivated by "plus", suggests one or more instances of `var3`. The additional `s` in `var1.ps` suggests one or more instances of `var1` separated by `';'`.

```
var1    : var1 ',' var1         /* R    pairing                 */
        ¦ var1 '.' var1         /* R    list cons               */
        ¦ var2
        ;
var2    : var2 ':' type         /*       type constraint        */
        ¦ var3
        ;
var3    : ID
        ¦ '(' ')'               /*       empty varstruct        */
        ¦ '(' var1 ')'
        ¦ '[' ']'               /*       empty list             */
        ¦ '[' var1.ps ']'       /*       var1 ';' ... ';' var1  */
        ;
var1.ps : var1
        ¦ var1.ps ';' var1
        ;
```

Simultaneous bindings are accommodated by the following syntax.

```
bind1   : bind1 'and' bind2
        ¦ bind2
```

```
                ;
bind2           : var1 '=' exp1             /*      simple binding          */
                ¦ ID var3.p '=' exp1        /*      function definition     */
                ¦ ID var3.p ':' type '=' exp1
                ¦ '{' bind1 '}'             /*      new rule, see section 4 */
                ;
var3.p          : var3
                ¦ var3.p var3
                ;
```

Declarations consist of `let` followed by a binding:

```
decl            : 'let' bind1               /*      ordinary variables      */
                ¦ ...
```

4. Abstraction and local declarations

In addition to the `let` construct for subsidiary expressions (mentioned in Section 3), ML has an equivalent `where` construct:

```
let v = e1 in e    ≡    e where v = e1
```

There is no trouble in mixing `let` and `where`. It is easy to see that

```
let v1 = e1 where v2 = e2
```

can only be parenthesized:

```
let v1 = ( e1 where v2 = e2 )
```

Ambiguity arises when simultaneous bindings using `and` are allowed. Since `where` occurs in expressions while `and` occurs in bindings, the syntax in [3] says nothing about their relative precedence. As a result, both of the following expressions are ambiguous:

```
e where v1 = e1 where v2 = e2 and v3 = e3
let v1 = e1 where v2 = e2 and v3 = e3
```

The former case is reminiscent of the dangling-else in Algol 60: attaching the `and` to the nearest `where` leads to

```
e where v1 = ( e1 where v2 = e2 and v3 = e3 )
```

A case can also be made for *not* attaching an `and` to the nearest `where`. For example, treating `where` as a more local declaration than `let` suggests the parenthesization:

```
let v1 = ( e1 where v2 = e2 ) and v3 = e3
```

Since it is difficult to choose between the above possibilities, we restrict the syntax of `where` expressions. We follow [2] and resolve the difficulty by not allowing simultaneous bindings in conjunction with `where` unless they are enclosed between braces `{}`. (As far is parsing is concerned, there is no problem if normal parentheses are used instead of braces.)

More precisely, the following syntax will be used:

```
exp1            : '\' var3.p '.' exp1       /*      lambda abstraction      */
                ¦ decl 'in' exp1            /*      local declaration       */
                ¦ exp2 'where' bind2        /*      let bind2 in exp2       */
                ¦ ...
```

Type bindings are handled similiarly.

5. Expressions

With the exception of the `'+'` operator, all other expression constructs are easy to handle. Besides its use for addition in expressions, `'+'` was used for the disjoint sum of types in the fragment in Section 1. Moreover, the operator is left associative in expressions and right associative in type expressions.

Yacc provides a mechanism for temporarily overriding the declared precedence of an operator. The `%prec` in the following syntax rule

```
exp     : ...
        | exp '+' exp   %prec PLUS
```

forces the precedence of `'+'` in this production to be the same as that of `PLUS`.

The difficulties with `'+'` do not end with the above local precedence declaration. It is not clear which sense of `'+'` is intended in the following expression:

```
x : t + u
```

As in [2], `t+u` will be treated as a domain. This is the reason for the only shift/reduce conflict in the grammar in the appendices.

The precedence and associativity of operators in [3] and [2] is different. The attached grammar follows [3]. Precedence and associativity in [2] is suggested by:

```
%nonassoc       ':'
%right          '.'
%right          '@'
%left           OR
%left           '&'
%left           NOT
%left           '=' '>' '<'
%left           '-' '+'
%left           '/' '*'
```

6. User defined infix operators

ML allows the user to define "any identifier (and certain single characters)" as infix operators. "Such user defined infixes bind tighter than the infix '... => ... | ...' but weaker than `'or'` [3] ."

Since Yacc builds parsing tables when the grammar is presented, it is not convenient to add new infix operators. However, some provisions can be made for user defined infix operators by adding n new operators $\texttt{INFIX}_1, \cdots, \texttt{INFIX}_n$, where n is a large enough constant chosen at parser construction time. The lexical analyzer can keep a table of identifiers that have been defined to be infix operators. When an identifier declared to have the same precedence as $\texttt{INFIX}_i$ is recognized, the lexical analyzer returns the token $\texttt{INFIX}_i$, setting the global variable `yylval` to indicate which identifier was seen (as usual). Similar arrangements can be made for single character infix operators.

Acknowledgements

This paper could not have been written without the help and encouragement of Luca Cardelli and Dave MacQueen.

References

1. J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference," pp. 125-132 in *Proc. Intl. Conf. Information Processing*, Unesco, Paris (1960).

2. L. Cardelli, *VAX-ML Manual*, Bell Laboratories, Murray Hill NJ (to appear 1983).

3. M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science 78, Springer-Verlag, Berlin (1979).

4. S. C. Johnson, "Yacc – yet another compiler compiler," CSTR 32, Bell Laboratories, Murray Hill NJ (July 1975). See the *UNIX Programmer's Manual* 2 Section 19 (January 1979)

```
/*
        a yacc grammar for ml is obtained by including the
        files indicated.  the inclusion can be done using the
        c preprocessor (this file is called plan):

                /lib/cpp -P plan

        the resulting grammar has 1 shift/reduce conflict
*/

#include        "tokens"

%token          constant

%start          exp01

%%

#include        "types"

#include        "declarations"

#include        "exp"
```

```
/*
        token names formed by mapping to upper case, e.g. int => INT
*/
%token          ABS ABSRECTYPE ABSTYPE AND BOOL ELSE FAIL FAILWITH
%token          IF IN INT LET LETREC LETREF LETTYPE LOOP
%token          REC REF THEN TOKEN TYPE WHERE WHEREABSRECTYPE
%token          WHEREABSTYPE WHEREREC WHEREREF WHERETYPE WITH

%token          ASS                     /*      :=                      */
%token          XX                      /*      ||                      */
%token          XBS                     /*      |\                      */
%token          COND                    /*      =>                      */
%token          ID
%token          LAM                     /*      \                       */
%token          QBS                     /*      ?\                      */
%token          QQ                      /*      ??                      */
%token          STAR                    /*      *, **, ***, ...         */
%token          STAR0                   /*      *0, **0, ...            */
%token          STAR1                   /*      *1, **1, ...            */
%token          STARID                  /*      *ID, **ID, ...          */

/*
        precedence in type expressions
*/

%right          ARR
%right          '+'
%right          '#'

/*
        precedence in varstructs and expressions
*/

%right          ','                     /* R    pairing                 */
%nonassoc       DO
%right          OR
%right          '&'
%nonassoc       NOT
%left           '='
%right          '@'
%right          '.'                     /* R    list cons               */
%nonassoc       '>'
%nonassoc       '<'
%left           '-'
%left           PLUS
%left           '/'
%left           '*'
%nonassoc       UMINUS
```

```
/*
        types

        type variables, used to denote the types of
        polymorphic functions, begin with a star
*/

type    : type ARR type                 /* R    continuous function     */
        | type '+' type                 /* R    disjoint sum            */
        | type '#' type                 /* R    product                 */
        | s_type                        /*      simple type             */
        ;

s_type  : basic_type
        | type_var
        | ID                            /*      defined, abstract type  */
        | type_arg ID
        | '(' type ')'
        ;

type_arg: s_type
        | '(' type.2pc ')'              /*      two or more arguments   */
        ;

type.2pc: type ',' type
        | type.2pc ',' type
        ;

type_var: STAR                          /*      *,   **,   ...          */
        | STAR0                         /*      *0,  **0,  ...          */
        | STAR1                         /*      *1,  **1,  ...          */
        | STARID                        /*      *ID, **ID, ...          */
        ;

basic_type
        : '.'                           /*      type of ()              */
        | INT
        | BOOL
        | TOKEN
        ;
```

```
/*
        declarations decl
*/

decl    : LET bind1                 /*      ordinary variables      */
        ¦ LETREF bind1              /*      assignable variables    */
        ¦ LETREC bind1              /*      recursive functions     */
        ¦ LETTYPE type_bind1        /*      defined types           */
        ¦ ABSTYPE atype_bind1       /*      abstract types          */
        ¦ ABSRECTYPE atype_bind1/*      recursive abstract types*/
        ;

/*
        bindings bind1, bind2
*/

bind1   : bind1 AND bind2
        ¦ bind2
        ;

bind2   : var1 '=' exp01            /*      simple binding          */
        ¦ ID var3.p '=' exp01       /*      function definition     */
        ¦ ID var3.p ':' type '=' exp01
        ¦ '{' bind1 '}'             /*      new rule                */
        ;

var3.p  : var3
        ¦ var3.p var3
        ;

/*
        varstructs var1, var2, var3
*/

var1    : var1 ',' var1             /* R    pairing                 */
        ¦ var1 '.' var1             /* R    list cons               */
        ¦ var2
        ;

var2    : var2 ':' type             /*      type constraint         */
        ¦ var3
        ;

var3    : ID
        ¦ '(' ')'                   /*      empty varstruct         */
        ¦ '(' var1 ')'
        ¦ '[' ']'                   /*      empty list              */
        ¦ '[' var1.ps ']'           /*      var1 ';' ... ';' var1   */
        ;

var1.ps : var1
        ¦ var1.ps ';' var1
        ;

/*
        type bindings
```

```
*/

type_bind1
        : type_bind1 AND type_bind2
        | type_bind2
        ;

type_bind2
        : ID '=' type
        ;

atype_bind1
        : dummy_arg ID '=' type AND atype_bind1
        | dummy_arg ID '=' type WITH bind1
        ;

atype_bind2
        : dummy_arg ID '=' type WITH bind2
        ;

dummy_arg
        : /* empty */
        | type_var
        | '(' type_var.pc ')'
        ;

type_var.pc
        : type_var
        | type_var.pc ',' type_var
        ;
```

```
/*
        expressions exp01, exp02, ... , exp11
*/

exp01   : LAM var3.p '.' exp01          /*      LAM = \, abstraction    */
        ! decl IN exp01                 /*      local declaration       */
        ! exp02 WHERE bind2             /* R    let bind2 in exp02      */
        ! exp02 WHEREREF bind2          /* R    bind2 derives a string  */
        ! exp02 WHEREREC bind2          /* R    ending in exp01         */
        ! exp02 WHERETYPE type_bind2
        ! exp02 WHEREABSTYPE atype_bind2
        ! exp02 WHEREABSRECTYPE atype_bind2
        ! exp02
        ;

exp02   : exp03 ';' exp02               /*      sequencing              */
        ! exp03
        ;

exp03   : exp04 catches                 /*      failure trap and loop   */
        ;

catches : catchall                      /*      catch all failures      */
        ! catch catches                 /* R                            */
        ;

catch   : QQ exp11 exp04                /*      QQ = ??, trap           */
        ! XX exp11 exp04                /*      XX = !!, trap'n iterate */
        ;

catchall: /* empty */
        ! '?' exp04
        ! '!' exp04
        ! QBS ID exp04                  /*      QBS = ?\ID              */
        ! XBS ID exp04                  /*      XBS = !\ID              */
        ;

exp04   : conditional                   /*      conditional and loop    */
        ! FAIL                          /*      failwith 'fail'         */
        ! FAILWITH exp05                /*      failure with token      */
        ! var1 ASS exp05                /*      ASS = :=, assignment    */
        ;

conditional
        : IF exp05 THEN exp05
        ! IF exp05 LOOP exp05
        ! IF exp05 THEN exp05 conditional
        ! IF exp05 LOOP exp05 conditional
        ! else
        ;

else    : ELSE exp05
        ! LOOP exp05
        ;

exp05   : exp05 ',' exp05               /* R    pairing                 */
```

```
        | DO exp05                          /*      eval. for side effects  */
        | exp06
        ;

exp06   : exp07 COND exp06 '|' exp06        /* R    COND = =>               */
        | exp07
        ;

exp07   : exp07 OR exp07                    /* R    disjunction             */
        | exp07 '&' exp07                   /* R    conjunction             */
        | NOT exp07                         /*      negation                */
        | exp07 '=' exp07                   /* L    equality                */
        | exp07 '@' exp07                   /* R    list append             */
        | exp07 '.' exp07                   /* R    list cons               */
        | exp07 '>' exp07                   /*      greater than            */
        | exp07 '<' exp07                   /*      less than               */
        | exp07 '-' exp07                   /* L    subtraction             */
        | exp07 '+' exp07  %prec PLUS       /* L    PLUS since + is %right  */
        | exp07 '/' exp07                   /* L    division                */
        | exp07 '*' exp07                   /* L    multiplication          */
        | exp08
        ;

exp08   : '-' exp08                         /*      unary minus             */
        | exp09
        ;

exp09   : exp09 ':' type                    /*      type constraint         */
        | exp10
        ;

exp10   : exp10 exp11                       /* L    function application    */
        | exp11
        ;

exp11   : ID                                /*      variable                */
        | constant
        | '(' exp01 ')'                     /*      equiv. exp01            */
        | '(' ')'                           /*      empty                   */
        | '[' exp01 ']'                     /*      generates a list        */
        | '[' ']'                           /*      empty list              */
        ;
```

The Functional Abstract Machine

Luca Cardelli

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

The Functional Abstract Machine (Fam) is a stack machine designed to support functional languages on large address space computers. It can be considered a SECD machine [1] which has been optimized to allow very fast function application and the use of true stacks (as opposed to linked lists).

The machine qualifies to be called functional because it supports functional objects (closures, which are dynamically allocated and garbage collected), and aims to make function application as fast as, say, taking the head of a list. All the optimization and support techniques which make application slower are strictly avoided, while tail recursion and pattern-matching calls are supported. Restricted side effects and arrays are provided, but they are less efficient than one might expect. Moreover the performance of the proposed garbage collector deteriorates in the presence of large numbers of updatable objects.

The machine is intended to make compilation from high level languages easy and regular, by providing a rich and powerful set of operations and an open-ended collection of data types. This richness of types can also facilitate portability, because every type can be independently implemented in different ways. However the number of machine instructions tends to be high, and in general there is little concern for minimality.

The instructions of the machine are not supposed to be interpreted, but assembled into machine code and then executed. This explains why no optimized special-case operations are provided; special cases can be easily detected at assembly time.

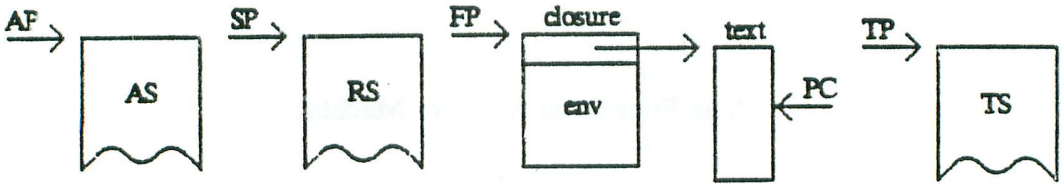
For efficiency considerations, the abstract machine is not supposed to perform run-time type checking (even if a hardware implementation of it might), and hence it is not type-safe. Moreover, as a matter of principle, there is no primitive to test the type of an object; the correct application of machine operations should be guaranteed by typechecking in the source language. Where needed, the effect of run-time typechecking can be achieved by the use of *variant* (i.e. tagged) data types.

2. The State

The state of the abstract machine is determined by six pointers, together with their denotations, and a set of memory locations. The pointers are the Argument Pointer, the Frame Pointer, the Stack Pointer, the Trap Pointer, the Program Counter and the Environment Pointer. They point to three independent stacks and, directly or indirectly, to the data heap. The memory takes care of side-effects in the heap and includes the file system.

The *Argument Pointer* (AP) points to the top of the *Argument Stack* (AS), where arguments are loaded to be passed to functions and results of functions are delivered. This stack is also used to store local and temporary values. In all machine operations which take displacements on AS, the first object on AS is at displacement zero.

The *Frame Pointer* (FP) points to the current closure (or frame) (FR) consisting of the text of the currently executed program, and of an environment for the free variables of the program.



The State

The *Program Counter* (PC) points to the program to be executed (PR) (which is part of the current closure).

The *Stack Pointer* (SP) points to the top of the *Return Stack* (RS), where program counter and closure pointer are saved during function calls.

The *Trap Pointer* (TP) points to the *Trap Stack* (TS), where *trap frames* are stored. A trap frame is a record of the state of the machine, which can be used to resume a previous machine state (side effects in the heap are not reverted).

The *Environment Pointer* (EP) also points to AS, and it defines the top level environment of execution for use in interactive systems. At the beginning of execution, EP is the same as AP, but it normally grows because of top level definitions.

The abstract machine assumes the existence of a memory of cells of different sizes. Typical cell types are: *triv*, containing the value triv; *bool*, containing booleans; *int*, containing unbounded precision integers; *string*, containing character strings; *ref*, containing updatable pointers (the only updatable objects in the machine together with arrays); *pair*, a pair of cells; *nil*, an empty-list cell; *list*, a cell paired with a nil or list cell, used to represent linear lists; *record*, an n-tuple of cells; *variant*, a tagged cell, used to represent disjoint union types; *text*, a cell containing executable code; *closure*, a function cell consisting of a text cell and a set of cells for the global variables of the text; *array*, an efficient representation of lists of refs.

The exact format of these cells is inessential, as long as the primitive operations on them have the expected properties. The abstract machine does not assume these cells to contain any information about the type; however the garbage collector will want to know at least about the *format* of the cells (different types may have the same storage format) This can be efficiently encoded in the address of a cell. The generic equality operations use this information too.

Stacks and cells contain pointers to other cells, and this convention will be strictly used in the pictures which follow. However in practice cells which are not bigger then a pointer (e.g. bool or short integer) are stored directly, instead of storing pointers to them. These are called *unboxed* cells, and it must be possible to distinguish them from pointers for the sake of the garbage collector (e.g. by imposing that every pointer has a value bigger then the value of any unboxed cell). Ref cells must always be *boxed*, to guarantee the sharing of side-effects.

3. Operational Semantics

The semantics of the abstract machine is given operationally by state transitions. A machine state is represented by a tuple:

(AS, RS, FR, PR, TS, ES, M)

some conditions must hold for a tuple to be a valid machine state, and these are mentioned below.

For any stack S (i.e. AS, RS, TS or ES) we write S.x:t for the operation of pushing a cell x of type t on the top of S (t may contain type variables α, β, etc.). The empty stack is <> and S.x:t is a stack iff S is a stack. Moreover S[n]x:t is a stack which is the same as S, except that the n-th cell

from the top contains x of type t; the top cell of S has displacement 0. In case of conflicting substitutions, like S[n]x:t[n]x':t', the rightmost substitution is the valid one.

A tuple (AS, RS, FR, PR, TS, ES, M) is a machine state only if ES (pointed to by EP) is equal to AS minus some of the top cells of AS.

The frame FR (pointed to by FP) has the form:

$$\text{closure}(\text{text}(c,l_0,..,l_n),x_0,..,x_m): \alpha\rightarrow\beta$$

where c is a sequence of machine operations, l_i are *literals* of c and x_j are values for the free variables of (the source program whose translation is) c. The literals of c are "big constants" like strings and inner lambda expressions, which occur in (the program whose translation is) c; they are taken out of the code so that the garbage collector can access them easily. The code c together with its literals is a *text* (and a literal can be a text). A text together with its free variables is a *closure*. Every closure implements a function having some type $\alpha\rightarrow\beta$.

The program PR (pointed to by PC) is a string of abstract machine operations. The empty program is <> and the initial instruction of PR is singled out by writing $op(x_1:\alpha_1,..,x_n:\alpha_n).PR'$, where x_i are the parameters of op.

The *memory* M is a pair of functions:

$$M = L,F: (\text{address} \rightarrow \text{value}) \times (\text{streamname} \rightarrow \text{stream})$$

where L are the *locations* and F is the *file system*. A tuple (AS, RS, FR, PR, TS, ES, M) is a machine state only if M defines all the addresses and file names mentioned by the other elements of the tuple. Stream names are strings, and streams (which represent files) are lists of characters (in terms of abstract machine data structures, characters are 1-character strings). For any stream q and character c, c.q is the result of prefixing c at the head of q, and q.c is the result of appending c at the tail of q (we also use s.q and q.s for strings $s=c_1..c_n$); <> is the empty stream.

Addresses (the formal characterization of ref cells) are, say, integers; values are all the abstract machine data types, including addresses and stream names but excluding streams. Given an address a, M(a) = L(a) is the value contained at that address in L, and M[v/a] = L[v/a] is the memory which is the same as M, except that L(a) is the value v. Given a string s, M(s) = F(s) is the stream with stream name s in F, and M[q/s] = F[q/s] is the memory which is the same as M, except that F(s) is the stream q. The convention about conflicting substitutions mentioned above applies.

Every machine operation op(..) implements a state transition, denote by:

$$(AS, RS, FR, op(x_1:\alpha_1,..,x_n:\alpha_n).PR, TS, ES, M) \Rightarrow (AS', RS', FR', PR', TS', ES', M')$$

In order the make the operation more visible, we normally use the following equivalent notation:

$$op(x_1:\alpha_1,..,x_n:\alpha_n)\ (AS, RS, FR, \hat{}.PR, TS, ES, M) \Rightarrow (AS', RS', FR', PR', TS', ES', M')$$

There may be several state transitions for the same operation, with different starting states. This allows us to express operations which discriminate on some values present on the stacks (e.g. conditional jumps). There may even be several state transitions for the same operation and the same starting state, which expresses nondeterministic behaviour (e.g. a random number generator, or simply selecting a new unused address). Conversely, there may be no state transition for some operation in some state: this means that the machine reached an inconsistent state, and the result of the operation is unpredictable. Finally there may be no operation to execute (i.e. PR=<>), in which case the machine stops.

Some operations may *fail* on some inputs (e.g. taking the head of an empty list). This is a well defined situation, and we use the notation:

```
op(x1:α1,...,xn:αn)
  (AS, RS, FR, ^.PR, TS, ES, M) =>
     (AS', RS', FR', PR', TS', ES', M')
  ?'op' if p
```

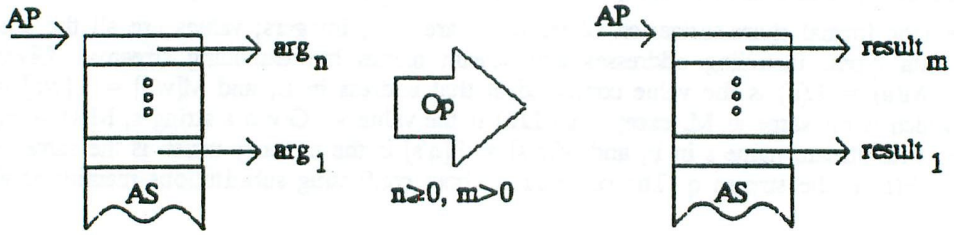
to indicate that we have a failure with reason 'op' (a string) when the predicate p is true of the starting state (AS, RS, FR, ^.PR, TS, ES, M), otherwise the normal state transition will happen. This is only an abbreviation for the following state transitions:

```
op(x1:α1,...,xn:αn)
  (AS, RS, FR, ^.PR, TS, ES, M) =>              (if not p)
     (AS', RS', FR', PR', TS', ES', M')

op(x1:α1,...,xn:αn)
  (AS, RS, FR, ^.PR, TS, ES, M) =>              (if p)
     (AS.'op', RS, FR, FailWith( ).PR, TS, ES, M)
```

where the FailWith operation, which is executed next, is defined in the section on Trap Operations.

4. Data Operations

These are operations which transfer data back and forth between the Argument Stack and data cells. In general they take n arguments (n≥0) from the top of AS popping the stack n times, and push back m results (m≥1).



Data Operations

Data operations may *fail*, and these failures can be trapped. Abstract machine failures are indistinguishable from user defined failures (see sections on failures).

The set of abstract machine data types is open ended, and so is the set of data operations. The rest of this section describes data types and operations which are thought to be commonly useful, but are often meant as simple suggestions. Some data types are used as arguments to basic machine operations, and hence must be present in every implementation. They include triv, bool, (short) int, string, list and closure. Closures are treated in a separate section.

4.1 Triv Operations

Triv is the type containing the single object triv. There is only one operation on this type, which constructs and pushes a triv cell on AS.

```
Triv ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
    (AS.triv, RS, FR, PR, TS, ES, M)
```

4.2 Boolean Operations

Boolean operations construct and manipulate booleans in the usual ways. Conditional branches could also be considered boolean operations, but they are described among the control operations.

```
True ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
    (AS.true, RS, FR, PR, TS, ES, M)

False ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
    (AS.false, RS, FR, PR, TS, ES, M)

Not ( )
 (AS.b:bool, RS, FR, ^.PR, TS, ES, M) =>
    (AS.not(b), RS, FR, PR, TS, ES, M)

And ( )
 (AS.b:bool.b':bool, RS, FR, ^.PR, TS, ES, M) =>
    (AS.and(b,b'), RS, FR, PR, TS, ES, M)

Or ( )
 (AS.b:bool.b':bool, RS, FR, ^.PR, TS, ES, M) =>
    (AS.or(b,b'), RS, FR, PR, TS, ES, M)

BoolEq ( )
 (AS.b:bool.b':bool, RS, FR, ^.PR, TS, ES, M) =>
    (AS.b=b', RS, FR, PR, TS, ES, M)
```

4.3 Integer Operations

Unbounded precision integers are the standard. The operation Divide fails with string 'divide' when the denumerator is zero, and Modulo fails with string 'modulo' when the second argument is zero. All the other operations are always defined (actually a 'collect' failure is generated when the result of an integer operation overflows the available memory). Short integers are acceptable as a partial implementation; overflows should then produce failures with strings 'minus', 'plus', 'diff' and 'times'. Real numbers, if implemented, should be a different data type.

```
Int (n: int)
 (AS, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n, RS, FR, PR, TS, ES, M)

Minus ( )
 (AS.n:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.-n, RS, FR, PR, TS, ES, M)
 ?'minus' on overflow
```

```
Plus ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n+m, RS, FR, PR, TS, ES, M)
 ?'plus' on overflow

Diff ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n−m, RS, FR, PR, TS, ES, M)
 ?'diff' on overflow

Times ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n×m, RS, FR, PR, TS, ES, M)
 ?'times' on overflow

Divide ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n/m, RS, FR, PR, TS, ES, M)
 ?'divide' if m=0

Modulo ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.mod(n,m), RS, FR, PR, TS, ES, M)
 ?'modulo' if m=0

Greater ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n>m, RS, FR, PR, TS, ES, M)

Less ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n<m, RS, FR, PR, TS, ES, M)

GreaterEq ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n≥m, RS, FR, PR, TS, ES, M)

LessEq ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n≤m, RS, FR, PR, TS, ES, M)

IntEq ( )
 (AS.n:int.m:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n=m, RS, FR, PR, TS, ES, M)
```

4.4 String Operations

Strings are ordered sequences of ascii characters. There is no limitation on their length apart from the size of the memory, and the Length operation is assumed to take constant time. *SubString* extracts a substring from a string, given a starting position (where the first char is position 1) and a substring size (may fail with 'substring'). *Explode* converts a string into a list of 1-character strings which are its characters. *Implode* concatenates a list of strings into a single string. *Explode-Ascii* converts a string into a list of numbers which are the Ascii codes for the characters of the string. *ImplodeAscii* converts a list of numbers (Ascii codes) into the corresponding string (may fail

with 'implodeascii'). *IntToString* converts an integer into its string representation ('−' is used for negative numbers). *StringToInt* converts a string representing a number into that number; between digits. *StringEq* compares two strings: they are equal if their length is the same and they contain the same characters.

```
Length ( )
 (AS.s:string, RS, FR, ^.PR, TS, ES, M) =>
    (AS.length(s), RS, FR, PR, TS, ES, M)

SubString ( )
 (AS.s:string.from:int.size:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.substring(s,from,size), RS, FR, PR, TS, ES, M)
 ?'substring' if from<1 or size<0 or from+size-1>length(s)

Explode ( )
 (AS.s:string, RS, FR, ^.PR, TS, ES, M) =>
    (AS.sl:string list, RS, FR, PR, TS, ES, M)

Implode ( )
 (AS.sl:string list, RS, FR, ^.PR, TS, ES, M) =>
    (AS.s:string, RS, FR, PR, TS, ES, M)

ExplodeAscii ( )
 (AS.s:string, RS, FR, ^.PR, TS, ES, M) =>
    (AS.nl:int list, RS, FR, PR, TS, ES, M)

ImplodeAscii ( )
 (AS.nl:int list, RS, FR, ^.PR, TS, ES, M) =>
    (AS.s:string, RS, FR, PR, TS, ES, M)
 ?'implodeascii' if nl contains some n<0 or n>127

IntToString ( )
 (AS.n:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.s:string, RS, FR, PR, TS, ES, M)

StringToInt ( )
 (AS.s:string, RS, FR, ^.PR, TS, ES, M) =>
    (AS.n:it, RS, FR, PR, TS, ES, M)
 ?'stringtoint' if s is not a valid int representation

StringEq ( )
 (AS.s:string.s':string, RS, FR, ^.PR, TS, ES, M) =>
    (AS.s=s', RS, FR, PR, TS, ES, M)
```

4.5 Reference Operations

Reference is the basic side-effectable type, to be used to implement assignable variables, updatable structures, circular data, call-by-reference, call-by-need, etc. Note that data like pairs, lists etc. are *not* assignable: this fact is crucially used by the garbage collector.

A reference is simply an assignable pointer to another cell. *Ref* builds a reference to an existing object. *At* extracts the contents of a reference. *Assign* takes a reference and a value, and assigns the value as the new content of the reference; the result of the assignment operation is triv. *DestRef* is like At, but takes two AS displacements: the first one is where the reference is, and the second one is where the content of the reference is stored.

```
Ref ( )
 (AS.x:α, RS, FR, ^.PR, TS, ES, M) =>
    (AS.l:address, RS, FR, PR, TS, ES, M[x/l])
    where l is a new address.

At ( )
 (AS.l:address, RS, FR, ^.PR, TS, ES, M) =>
    (AS.M(l), RS, FR, PR, TS, ER, M)

Assign ( )
 (AS.l:address.x:α, RS, FR, ^.PR, TS, ES, M) =>
    (AS.triv, RS, FR, PR, TS, ES, M[x/l])

DestRef (n: int≥0, m: int≥0)
 (AS[n]l:address, RS, FR, ^.PS, TS, ES, M) =>
    (AS[n]l[m]M(l), RS, FR, PR, TS, ES, M)
```

Note that, according to the convention on conflicting indexing (see the section on Operational Semantics), if n=m then the address l in AS is overwritten by its contents.

4.6 Pair Operations

A pair cell (x,y) is simply a pair of cells x and y, with a left and a right component.

```
Pair ( )
 (AS.x:α.y:β, RS, FR, ^.PR, TS, ES, M) =>
    (AS.(x,y), RS, FR, PR, TS, ES, M)

Left ( )
 (AS.(x:α,y:β), RS, FR, ^.PR, TS, ES, M) =>
    (AS.x, RS, FR, PR, TS, ES, M)

Right ( )
 (AS.(x:α,y:β), RS, FR, ^.PR, TS, ES, M) =>
    (AS.y, RS, FR, PR, TS, ES, M)

DestPair (n: int≥0, m: int≥0, p: int≥0)
 (AS[n](x:α,y:β), RS, FR, ^.PR, TS, ES, M) =>
    (AS[n](x,y)[m]x[p]y, RS, FR, PR, TS, ES, M)
```

4.7 List Operations

A list cell can be a *Nil* cell or a *Cons* cell containing an arbitrary cell (the head of the list) and another list cell (the tail). *Head* and *Tail* fail with 'head' and 'tail' on null lists. *DestNil* takes a list at some depth on AS and fails with 'destnil' if the list is not null. *DestCons* takes three AS displacements; the first one must contain a non-null list (otherwise fails with 'destcons'), the second one is where the head of that list is copied, and the third one is where the tail of that list is copied. The three displacements may coincide: the first one will be overwritten by the second and third, and the second one will be overwritten by the third.

```
Nil ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
    (AS.nil, RS, FR, PR, TS, ES, M)

Cons ( )
```

```
 (AS.x:α.l:α list, RS, FR, ^.PR, TS, ES, M) =>
    (AS.cons(x,l), RS, FR, PR, TS, ES, M)

Head ( )
 (AS.l:α list, RS, FR, ^.PR, TS, ES, M) =>
    (AS.head(l), RS, FR, PR, TS, ES, M)
 ?'head' if l=nil

Tail ( )
 (AS.l:α list, RS, FR, ^.PR, TS, ES, M) =>
    (AS.tail(l), RS, FR, PR, TS, ES, M)
 ?'tail' if l=nil

Null ( )
 (AS.l:α list, RS, FR, ^.PR, TS, ES, M) =>
    (AS.l=nil, RS, FR, PR, TS, ES, M)

DestNil (n: int≥0)
 (AS[n]l:α list, RS, FR, ^.PR, TS, ES, M) =>
    (AS[n]l:α list, RS, FR, PR, TS, ES, M)
 ?'destnil' if l≠nil

DestCons (n: int≥0, m: int≥0, p: int≥0)
 (AS[n]l:α list, RS, FR, ^.PR, TS, ES, M) =>
    (AS[n]l[m]head(l)[p]tail(l), RS, FR, PR, TS, ES, M)
 ?'destcons' if l=nil
```

4.8 Record Operations

Records are tuples of cells (written $(|x_1:\alpha_1,..,x_n:\alpha_n|)$) with a constant-time field selection operation. *Record* builds a record of n fields taken from AS. *Field* selects a field of a record. *DestRecord* takes an AS displacement and a list of n AS displacements (for records of n fields) and distributes the fields on AS according to the displacements. The rightmost displacements overwrite the previous ones when they coincide.

```
Record (n: int≥0)
 (AS.x1:α1..xn:αn, RS, FR, ^.PR, TS, ES, M) =>
    (AS.(|x1,..,xn|), RS, FR, PR, TS, ES, M)

Field (i: int≥1≤n)
 (AS.(|x1:α1,..,xn:αn|), RS, FR, ^.PR, TS, ES, M) =>
    (AS.xi, RS, FR, PR, TS, ES, M)

DestRecord (n: int≥0, [m1: int≥0; .. ; mp: int≥0])
 (AS[n](|x1:α1,..,xp:αp|), RS, FR, ^.PR, TS, ES, M) =>
    (AS[n](|x1,..,xp|)[m1]x1..[mp]xp, RS, FR, PR, TS, ES, M)
```

The Field operation is undefined if the index i is out of bound; similarly DestRecord is undefined if AS[n] is not a record of lenght p. As we already mentioned, it is assumed that these situations can never arise at run time because of typechecking at the the source program level.

4.9 Variant Operations

Variant cells (written [|a=x:t|]) contain a tag a (an integer) and another cell x; they are used to discriminate among a finite set of possibilities. *Variant* builds a variant with a given tag, taking the contents from the stack. *As* extracts the contents of a variant, provided that the given tag matches the variant tag (may fail with 'as'). *Is* tests whether a given tag is the tag of a variant on the stack. *DestVariant* is like As, but works at an arbitrary displacement on AS and may fail with 'destvariant'. The *Case* operation associates a program with each variant tag, by making a constant-time selection based on the tag of a variant on AS.

```
Variant (a: int≥1)
 (AS.x:α, RS, FR, ^.PR, TS, ES, M) =>
    (AS.[|a=x|], RS, FR, PR, TS, ES, M)

As (a: int≥1)
 (AS.[|a'=x:α|], RS, FR, ^.PR, TS, ES, M) =>
    (AS.x, RS, FR, PR, TS, ES, M)
 ?'as' if a≠a'

Is (a: int≥1)
 (AS.[|a'=x:α|], RS, FR, ^.PR, TS, ES, M) =>
    (AS.a=a', RS, FR, PR, TS, ES, M)

DestVariant (a: int≥1, n: int≥0, m: int≥0)
 (AS[n][|a'=x:α|], RS, FR, ^.PR, TS, ES, M) =>
    (AS[n][|a'=x|][m]x, RS, FR, PR, TS, ES, M)
 ?'destvariant' if a≠a'

Case ([c1: α1→β, .. , cp: αp→β])
 (AS.[|ai=x:αi|], RS, FR, ^.PR, TS, ES, M) =>
    (AS.x, RS, FR, ci, TS, ES, M)        (with 1≤i≤p)
```

4.10 Array Operations

Arrays are considered a constant access time implementation of lists of references, and hence are assignable. Arrays can be built from lists or by tabulating functions in an interval, and can be disassembled again into lists. Their LowerBound and Size attributes are also computed in constant time.

Array takes a lowerbound and a list and makes an array with that lowerbound, whose i-th element is the i-th element of the list; the size of the array is the length of the list. *Tabulate* takes a function f with integer domain, a lowerbound and a size, and makes an array with that size and lowerbound whose i-th element is f(i), for i ranging from lowerbound to lowerbound+size−1; it fails if the size is negative. *LowerBound* takes an array and returns its lowerbound. *Size* takes an array and returns its size. *Sub* takes an array and an index i and returns the value of the i-th element of the array; it fails if i is out of bounds. *Update* takes an array, an index i, and a value x and updates the i-th element of the array by x, returning triv; it fails if i is out of bounds. *ArrayToList* takes an array and makes a list of its contents in their order.

```
Array ( )
 (AS.lb:int.e:α list, RS, FR, ^.PR, TS, ES, M) =>
    (AS.array(lb,n,[l1,..,ln]), RS, FR, PR, TS, ES, M[nth(1,e)/l1]..[nth(n,e)/ln])
 where nth(i,e) is the i-th element of e,
 n is the length of e, and li are new addresses

Tabulate ( )
```

```
(AS.f:int→α.lb:int.size:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.array(lb,size,[l1,...,ln]), RS, FR, PR, TS, ES, M[f(lb)/l1]..[f(lb+size−1)/ln])
?'tabulate' if size<0
where li are new addresses

LowerBound ( )
(AS.array(lb,size,[l1,...,ln]), RS, FR, ^.PR TS, ES, M) =>
    (AS.lb, RS, FR, PR, TS, ES, M)

Size ( )
(AS.array(lb,size,[l1,...,ln]), RS, FR, ^.PR, TS, ES, M) =>
    (AS.size, RS, FR, PR, TS, ES, M)

Sub ( )
(AS.array(lb,size,[l1,...,ln]).i:int, RS, FR, ^.PR, TS, ES, M) =>
    (AS.M(li−lb+1), RS, FR, PR, TS, ES, M)
?'sub' if i<lb or i≥lb+size

Update ( )
(AS.array(lb,size,[l1,...,ln]).i:int.x:α, RS, FR, ^.PR, TS, ES, M) =>
    (AS.triv, RS, FR, PR, TS, ES, M[x/li−lb+1])
?'update' if i<lb or i≥lb+size

ArrayToList ( )
(AS.array(lb,size,[l1,...,ln]), RS, FR, ^.PR, TS, ES, M) =>
    (AS.cons(M(l1),...cons(M(ln),nil)..), RS, FR, PR, TS, ES, M)
```

The operation 'array' used above is the basic allocator of array objects.

4.11 Equality

There are two general-purpose equality operations, apart from the equality operations on ground types already described. They are Equal (structural equality) and Isomorphic (structural equality on possibly circular data).

Equal checks the structural equality of data, but it diverges on circular data structures. It fails with string 'equal' if the structures contain functional objects.

Isomorphic is like Equal, but on circular structures it returns true iff the infinite unfoldings of the structures are equal. It fails with string 'isomorphic' if the structures contain functional objects.

```
Equal ( )
(AS.x:α.y:α, RS, FR, ^.PR, TS, ES, M) =>
    (AS.equal(x,y), RS, FR, PR, TS, ES, M)
?'equal' if x or y contain closures

Isomorphic ( )
(AS.x:α.y:α, RS, FR, ^.PR, TS, ES, M) =>
    (AS.isomorphic(x,y), RS, FR, PR, TS, ES, M)
?'isomorphic' if x or y contain closures
```

5. Stack Operations

Stack operations manipulate the argument stack. *GetLocal(n)* copies the n-th cell from the top of AS onto AS. *Inflate(n,m)* inserts n null cells above the m-th cell from the top of AS. *Deflate(n,m)* deletes n cells starting from the m-th cell from the top of AS; cells above and below the deleted area are recompacted. *Permute([p_0; .. ;p_{n-1}])* permutes the top n cells of AS simultaneously copying the i-th cell from the top into the p_i-th one, for every i in 0..n−1 (p_0..p_{n-1} must be a permutation of 0..n−1).

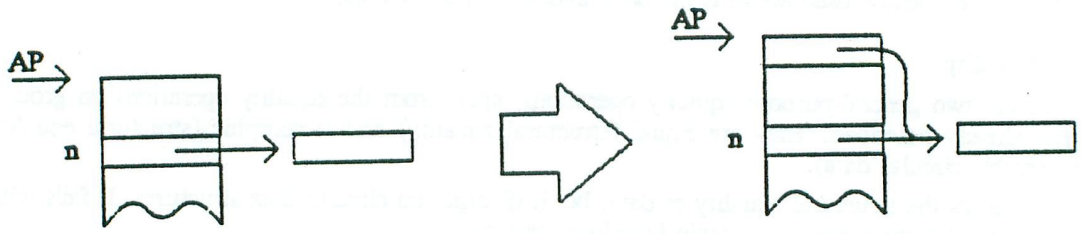
GetLocal (n: int≥0)
(AS[n]x:α, RS, FR, ^.PR, TS, ES, M) =>
(AS[n]x.x, RS, FR, PR, TS, ES, M)

Inflate (n: int≥0, m: int≥0)
(AS.x_{m-1}:α_{m-1}..x_0:α_0, RS, FR, ^.PR, TS, ES, M) =>
(AS.$triv_{n+m-1}$..$triv_m$.x_{m-1}..x_0, RS, FR, PR, TS, ES, M)

Deflate (n: int≥0, m: int≥0)
(AS.x_{n+m-1}:α_{n+m-1}..x_0:α_0, RS, FR, ^.PR, TS, ES, M) =>
(AS.x_{m-1}..x_0, RS, FR, PR, TS, ES, M)

Permute ([p_0: int≥0≤n−1;..;p_{n-1}: int≥0≤n−1])
(AS.x_{n-1}:α_{n-1}..x_0:α_0, RS, FR, ^.PR, TS, ES, M) =>
(AS.$x_{p_{n-1}}$..x_{p_0}, RS, FR, PR, TS, ES, M)

An Inflate operation with m=0 pushes n cells on top of AS; similarly Deflate with m=0 pops n cells from the top of AS.



GetLocal

6. Closure Operations

A closure is a data object representing a function; it contains the text of the function and the value of its free variables. The text of a function is in itself a rather complex structure; it contains a sequence of instructions in some suitable machine language, and a set of *literals* which may be strings or other text cells. Text literals are needed because a function may return another function which is textually contained in it, and we must be able to extract its text (because of garbage collection problems there can only be pointers *to* cells, never pointers pointing *inside* cells). String literals are useful when a function contains constant strings in its text. It is not necessary to allocate those strings every time that function is executed; the allocation can be done once at assembly time, and the strings can be saved in literals to be retrieved later.

Closures are created by placing the values for free variables and the text of the function on AS, and then storing this information in a newly allocated closure cell. Closures for (mutually) recursive functions may contain loops, and are allocated in two steps: dummy closures for a set of mutually recursive functions are first allocated in the heap, and later on recursive closures are built by

filling the dummy closures. This way the closures may mutually contain pointers to the other (dummy) closures.

The operations on closures are *Closure* (which creates a closure with arguments on AS), *DumClosure* (which allocates an empty closure), *RecClosure* (which fills in dummy closures), *GetGlobal* (which retrieves the value of a free (global) variable) and *GetLiteral* (which retrieves a literal from the text of the closure). Moreover the closures *FunId* (identity function) and *FunComp* (function compositions) are provided as primitives (mostly to allow peephole optimizers to optimize occurrences of Id(x), Comp(f,Id) and Comp(Id,f)).

FunId ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
  (AS.id, RS, FR, PR, TS, ES, M)

FunComp ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
  (AS.comp, RS, FR, PR, TS, ES, M)

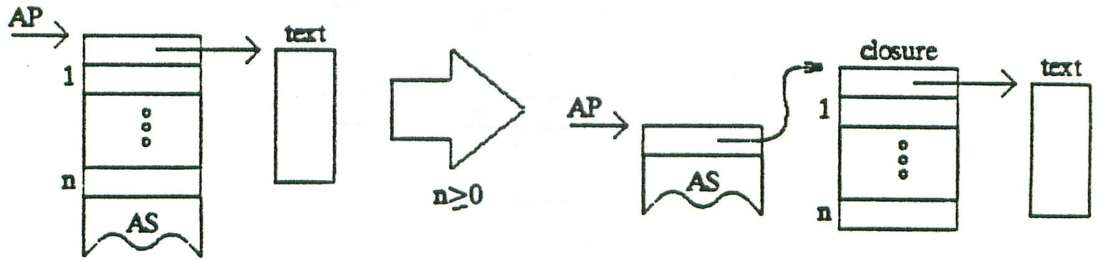
Closure (n: int≥0)
 (AS.x_1:α_1..x_n:α_n.t:text, RS, FR, ^.PR, TS, ES, M) =>
  (AS.closure(t,x_1,..,x_n), RS, FR, PR, TS, ES, M)

DumClosure (n: int≥0)
 (AS, RS, FR, ^.PR, TS, ES, M) =>
  (AS.closure(triv,$triv_1$,..,$triv_n$), RS, FR, PR, TS, ES, M)

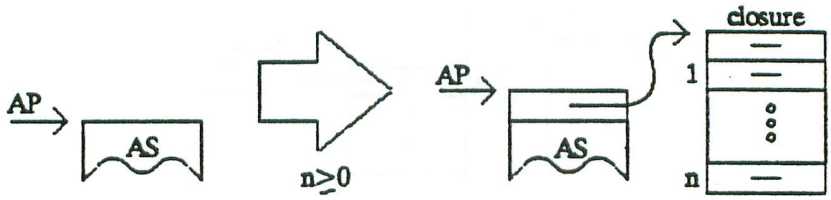
RecClosure (n: int≥0, m: int>n)
 (AS.x_n:α_n..x_1:α_1.t:text[m]closure(triv,$triv_1$,..,$triv_n$),
  RS, FR, ^.PR, TS, ES, M) =>
  (AS[m−n−1]closure(t,x_1,..,x_n), RS, FR, PR, TS, ES, M)

GetGlobal (n: int≥0≤p)
 (AS, RS, closure(t:text,x_0:α_0,..,x_p:α_p), ^.PR, TS, ES, M) =>
  (AS.x_n, RS, closure(t,x_0,..,x_p), PR, TS, ES, M)

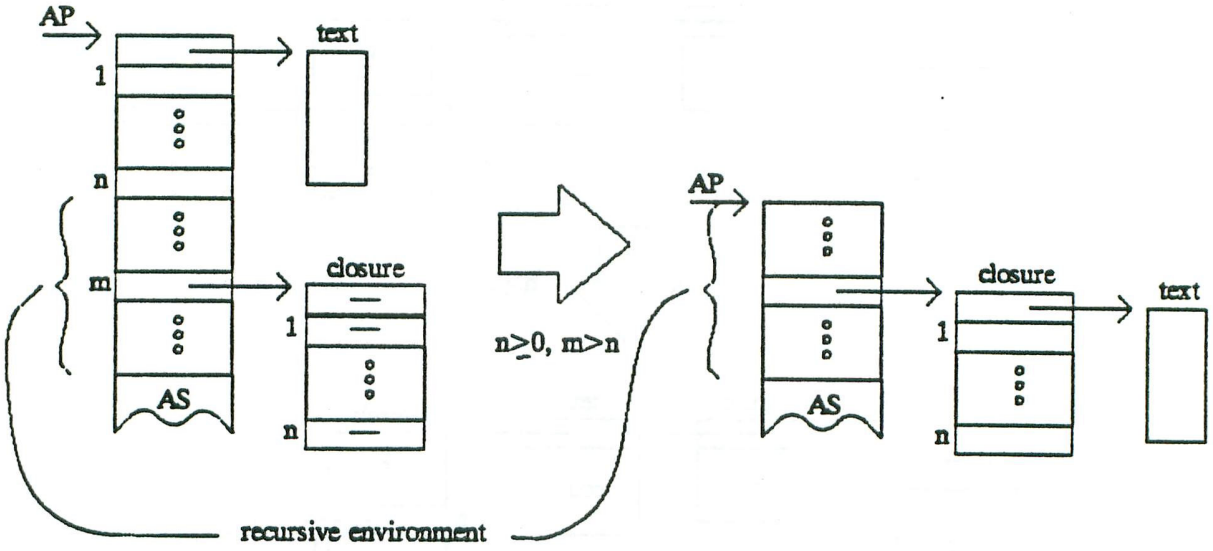
GetLiteral (n: int≥0≤p)
 (AS, RS, closure(text(c:code,x_0:α_0,..,x_p:α_p),..), ^.PR, TS, ES, M) =>
  (AS.x_n, RS, closure(text(c,x_0,..,x_p),..), PR, TS, ES, M)



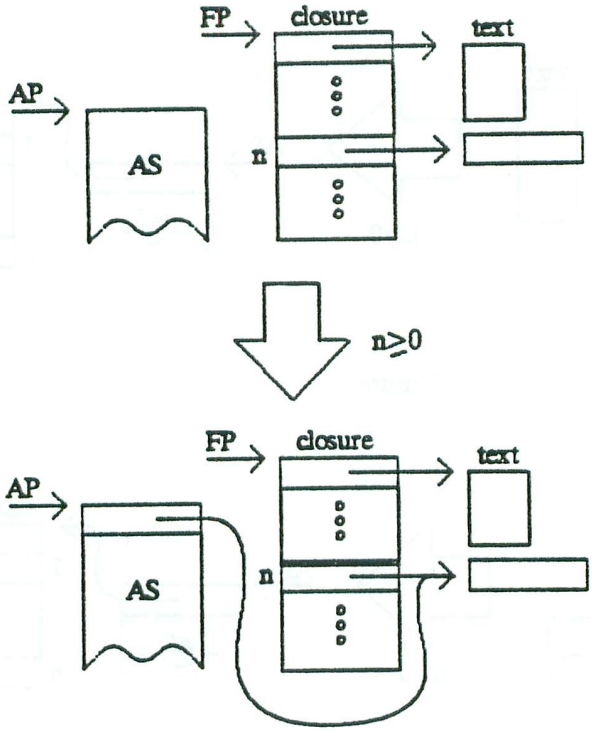
Closure



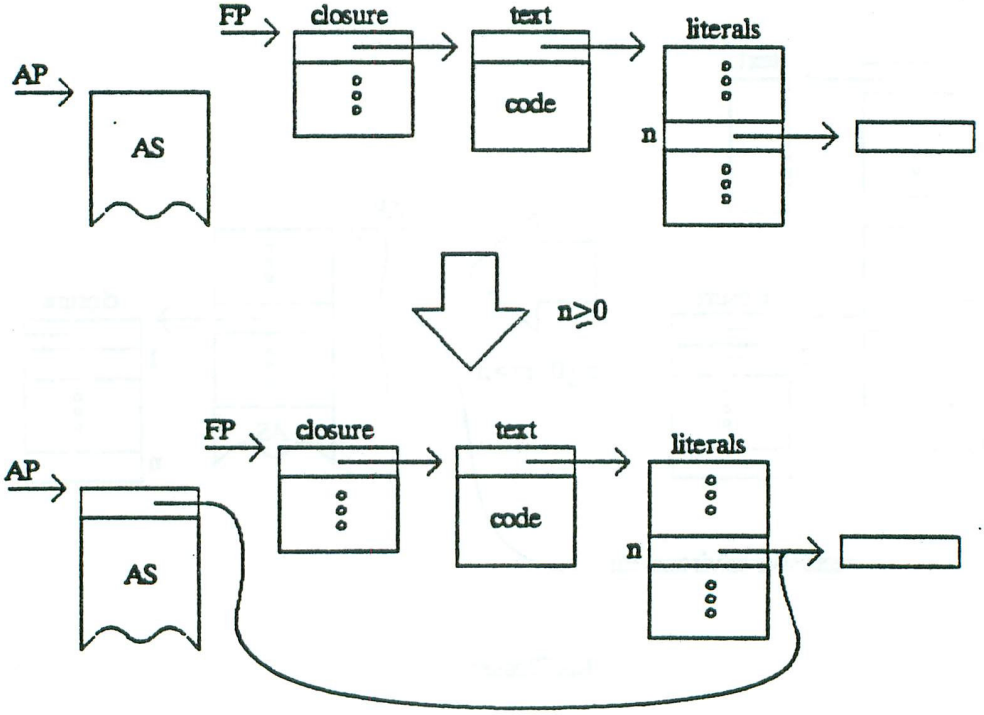
DumClosure



RecClosure



GetGlobal



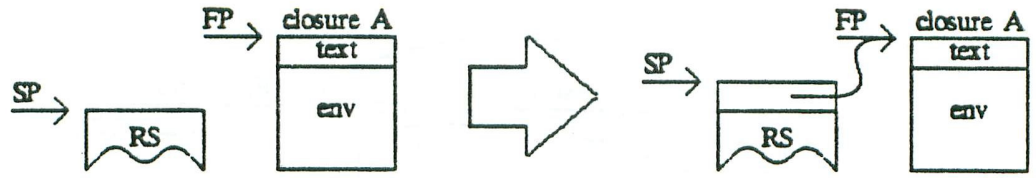
GetLiteral

7. Control Operations

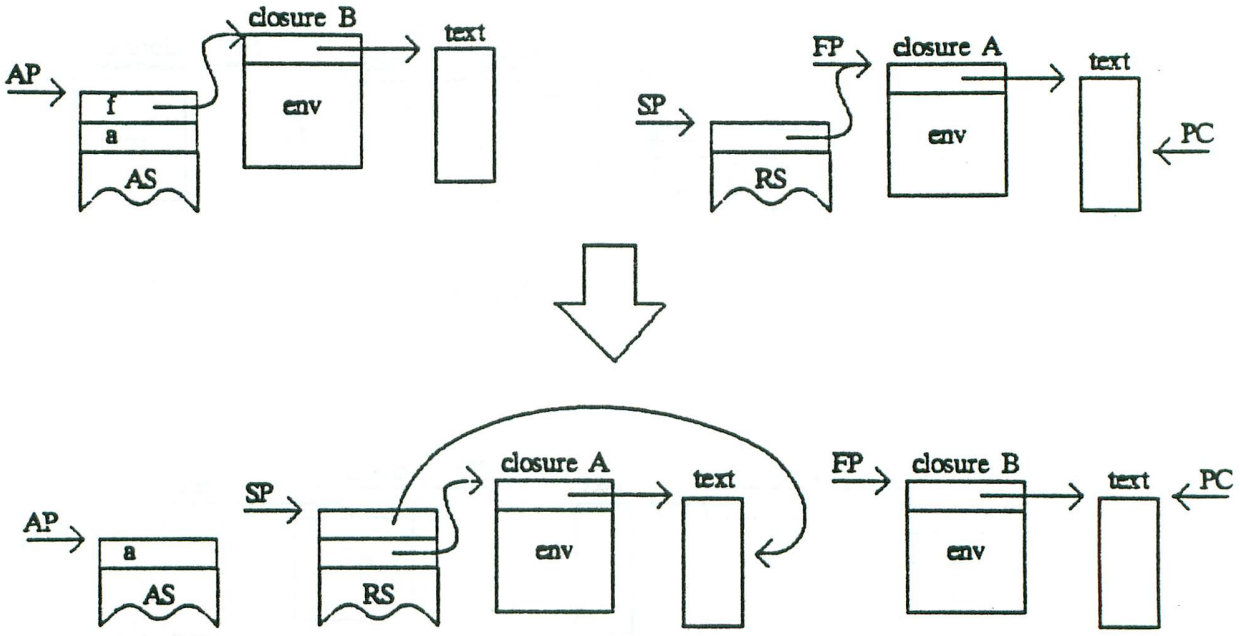
These are operations affecting the Program Counter or the Stack Pointer. *Jump* is an unconditional branch to another point in the same program text. *TrueJump* and *FalseJump* are conditional branches which jump when the top of AS is respectively true and false; otherwise the normal execution flow continues.

Function application is split into three operations: *SaveFrame* (which saves the calling closure on RS), *ApplFrame* (which saves the calling program counter on RS, and activates the called closure sitting on the top of AS by making it the one pointed by FP and by setting PC at its entry point) and *RestFrame* (which restores the calling closure from RS). This means that SaveFrame and RestFrame are inverses and can be cancelled out in multiple (curried) applications. The called closure uses Return to restore the calling program counter and return to the calling function (where a RestFrame is normally executed). The sequence SaveFrame, ApplFrame, RestFrame, Return can be optimized to *TailApply*, which uses a jump to pass control to the called function (there is no point in going back to the calling function because this would immediately execute a Return). The advantage of TailApply is that the control stack does not grow, hence iteration can be programmed by (tail) recursion without any penality. Return and TailApply also incorporate a Deflate operation, and hence take two arguments for deflating n cells below the m cell from the top of AS.

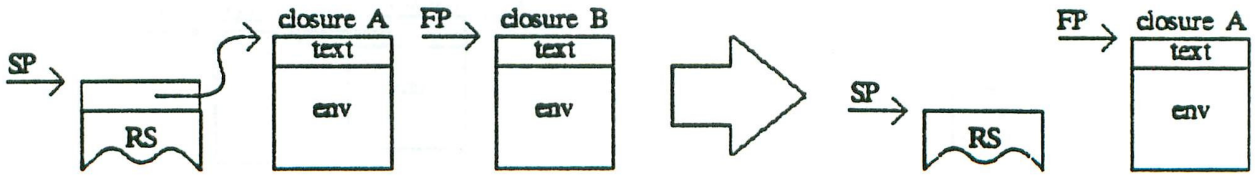
```
Jump (c: code)
 (AS, RS, FR, ^.PR, TS, ES, M) =>
    (AS, RS, FR, c, TS, ES, M)

TrueJump (c: code)
 (AS.true, RS, FR, ^.PR, TS, ES, M) =>
    (AS, RS, FR, c, TS, ES, M)
 (AS.false, RS, FR, ^.PR, TS, ES, M) =>
    (AS, RS, FR, PR, TS, ES, M)

FalseJump (c: code)
 (AS.false, RS, FR, ^.PR, TS, ES, M) =>
    (AS, RS, FR, c, TS, ES, M)
 (AS.true, RS, FR, ^.PR, TS, ES, M) =>
    (AS, RS, FR, PR, TS, ES, M)

SaveFrame ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
    (AS, RS.FR, FR, PR, TS, ES, M)

ApplFrame ( )
 (AS.x:α.closure(text(c:code,..),..):α→β, RS, FR, ^.PR, TS, ES, M) =>
    (AS.x, RS.PR, closure(text(c,..),..), c, TS, ES, M)

RestFrame ( )
 (AS, RS.FR', FR, ^.PR, TS, ES, M) =>
    (AS, RS, FR', PR, TS, ES, M)

Return (n: int≥0, m: int≥0)
 (AS.x_{n+m-1}:α_{n+m-1}..x_0:α_0, RS.c:code, FR, ^.PR, TS, ES, M) =>
    (AS.x_{m-1}..x_0, RS, FR, c, TS, ES, M)

TailApply (n: int≥0, m: int≥0)
 (AS.x_{n+m-1}:α_{n+m-1}..x_0:α_0.closure(text(c:code,..),..):α→β, RS, FR, ^.PR, TS, ES, M) =>
    (AS.x_{m-1}..x_0, RS, closure(text(c,..),..), c, TS, ES, M)
```



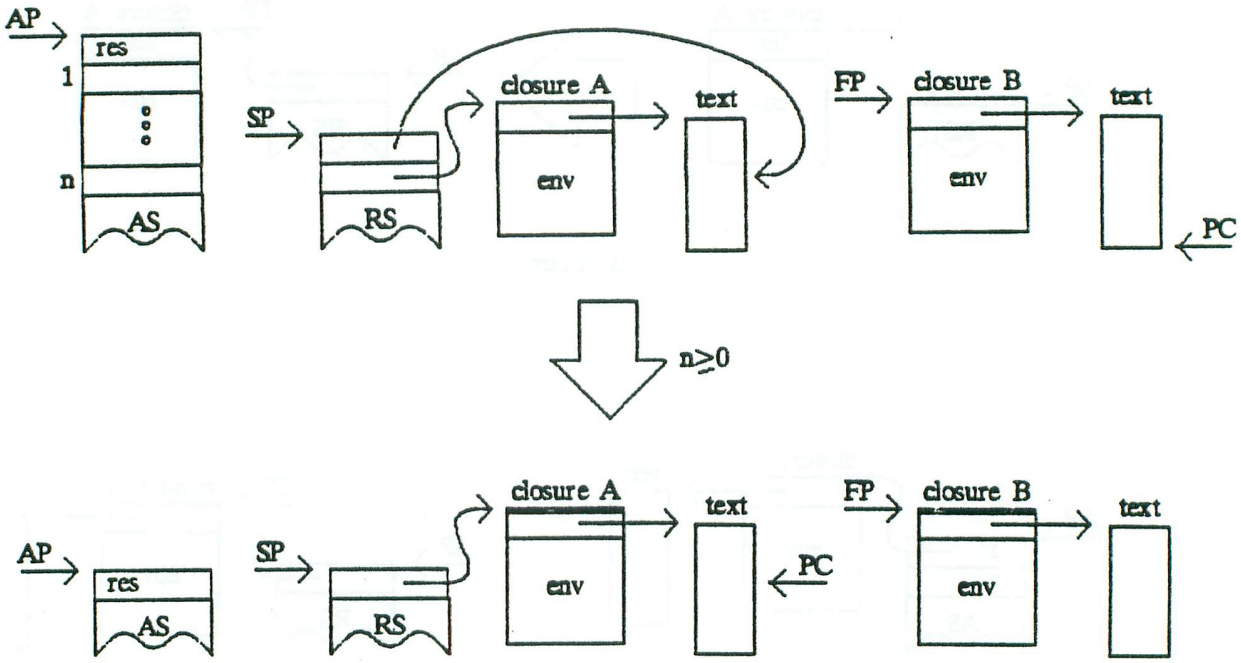
SaveFrame



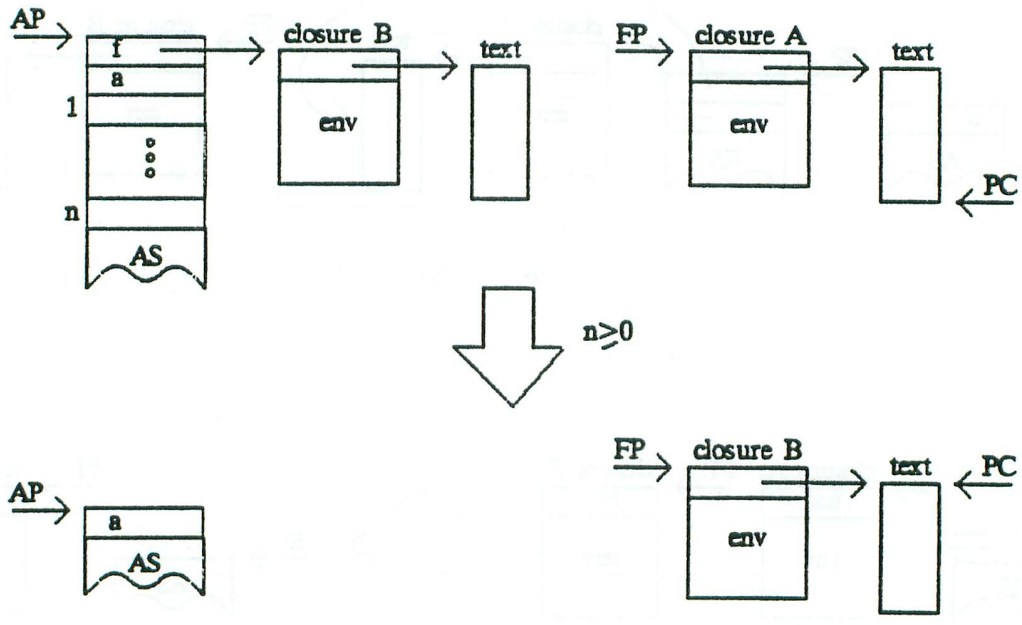
ApplFrame



RestFrame



Return



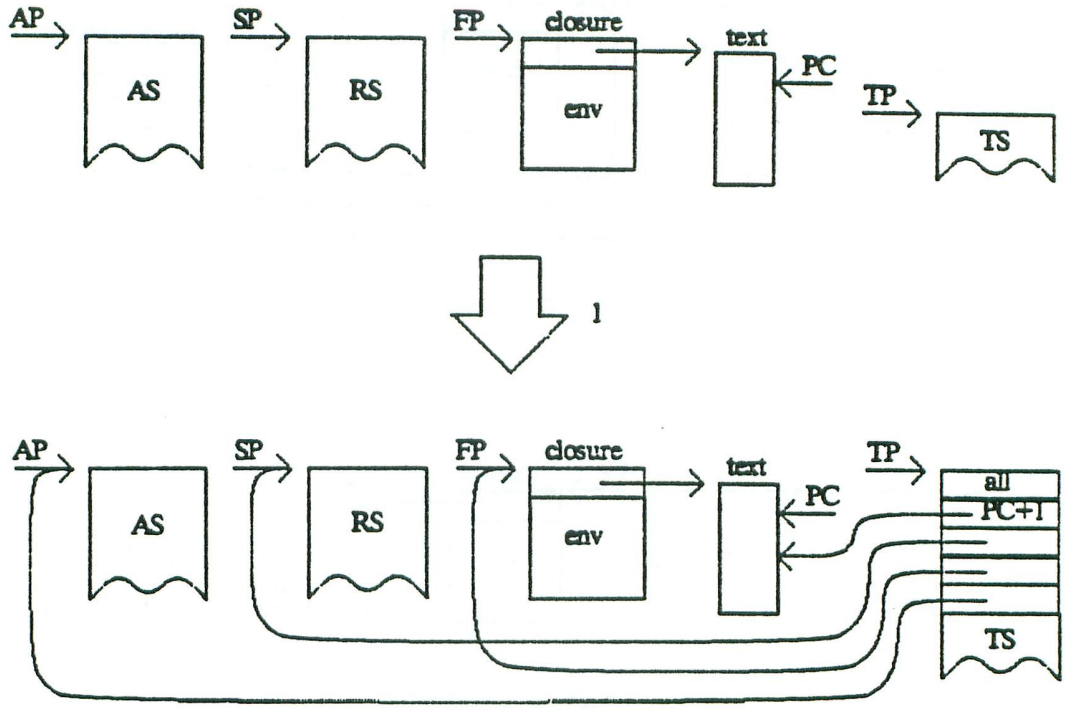
TailApply

8. Trap Operations

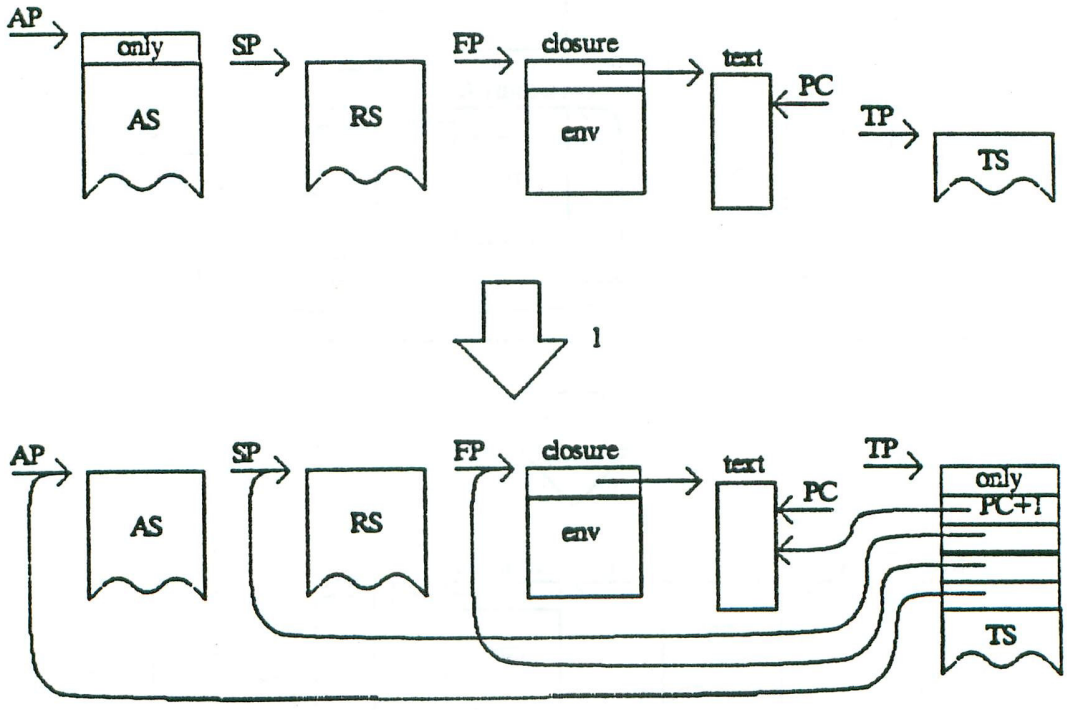
The FailWith operation takes a string and generates a failure with that string as failure reason. The failure can be trapped by a previously executed Trap or TrapList instruction, which saved the state of the machine (except the heap) at a failure recovery point.

Trap saves AP, FP, SP and a PC, corresponding to the failure handler, on the trap stack TS together with a flag meaning that all failures will be trapped. *TrapList* takes a list of strings and saves AP, FP, SP and the PC of the handler on TS, together with the list of strings which is used to selectively trap failures. *UnTrap* reverts the effect of the most recent Trap or TrapList. *FailWith* takes a string s and searches the trap stack from the top for a Trap block or a TrapList block with a list of strings containing s. If one is found, the corresponding state of the machine (AP, FP, SP, PC) is restored and the Trap or TrapList block and all the ones above it are removed. If no matching trap is found, the message 'Failure: ' followed by the failure string is printed on the standard output stream, and the machine stops.

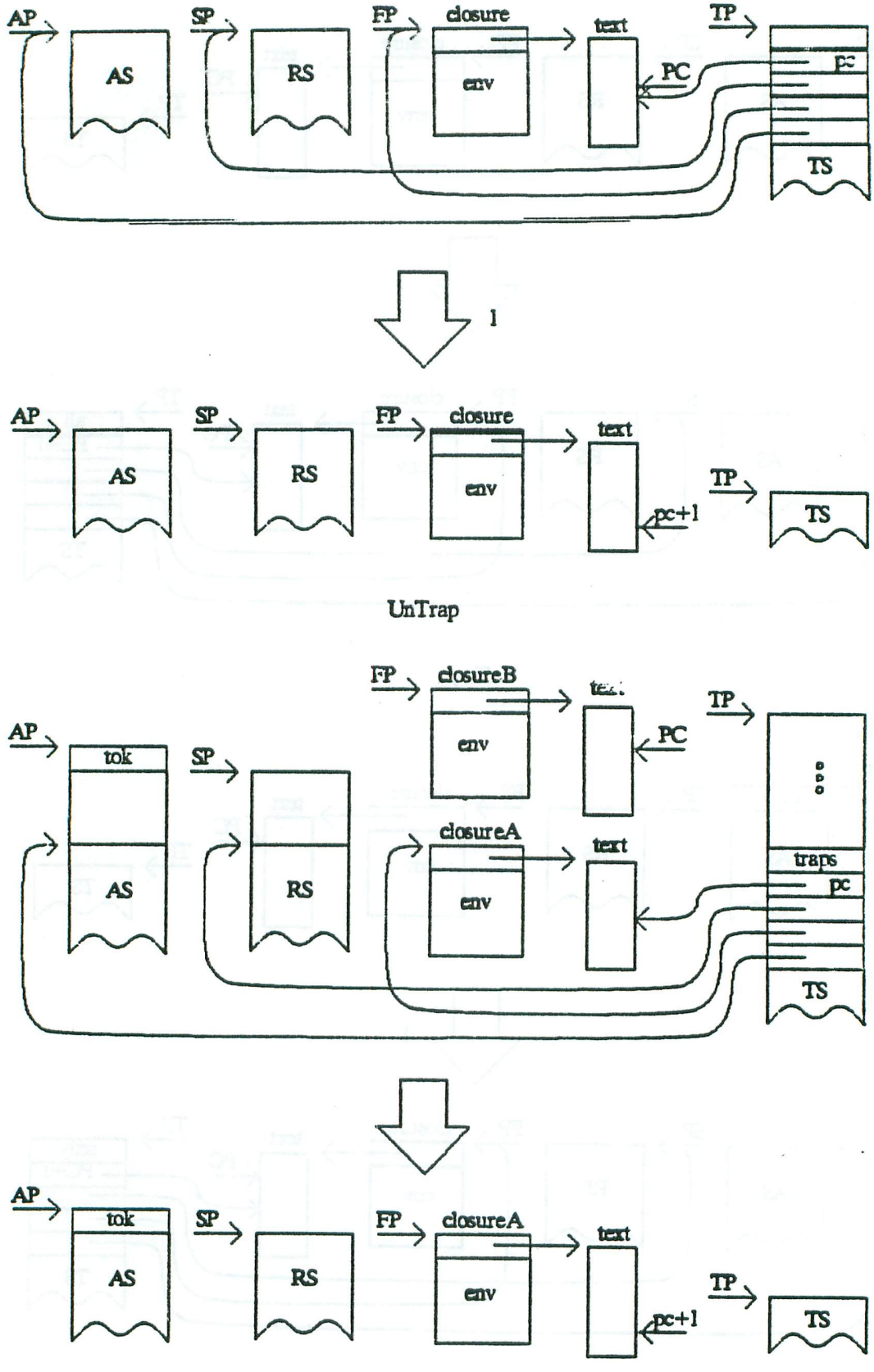
```
Trap (c: code)
 (AS, RS, FR, ^.PR, TS, ES, M) =>
    (AS, RS, FR, PR, TS.(all,c,RS,FR,AS), ES, M)

TrapList (c:code)
 (AS.sl:string list, RS, FR, ^.PR, TS, ES, M) =>
    (AS, RS, FR, PR, TS.(only(sl),c,RS,FR,AS), ES, M)

UnTrap (c: code)
 (AS, RS, FR, ^.PR, TS.(all,c,RS',FR',AS'), ES, M) =>
    (AS, RS, FR, c, TS, ES, M)
 (AS, RS, FR, ^.PR, TS.(only(sl),c,RS',FR',AS'), ES, M) =>
    (AS, RS, FR, c, TS, ES, M)

FailWith
 (AS.s:string, RS, FR, ^.PR, TS.(x,PR',RS',FR',AS').., ES, M) =>
    (AS'.s, RS', FR', PR', TS, ES, M)
    where (x,..) is the first trap block from the top of TS
    such that x=all or x=only(sl) and s is contained in sl.
 (AS.s:string, RS, FR, ^.PR, TS.(x,PR',RS',FR',AS').., ES, M) =>
    (AS.printfailure(s), RS, FR, <>, <>, ES, M)
    if there is no trap block satisfying the above condition.
```



Trap



TrapList



UnTrap

FailWith

9. Input-output

Input-output is done on streams. A stream is like a queue; characters can be read from one end and written on the other end. Reads are destructive, and they wait indefinitely on an empty stream for some character to be written. In what follows, a "file" is a character file on disk which has a "file name"; a "stream" is an abstract machine object (it is a pair of file descriptors, one open for input and the other one open for output).

Streams are associated with file *names* in the operating system. A copy of an existing stream can be associated with a file name by the *PutStream* operation which takes a string (the file name) and a stream and returns triv. The stream is unaffected by this operation. A failure with string 'putstream' occurs if the association cannot be carried out.

The operation *GetStream* takes a string (a file name) and returns a new stream whose initial content is the content of the corresponding file. It fails with string 'getstream' if the stream is not available (e.g. the file name syntax is wrong, or the file is locked). If no file exists with that file name, a new empty stream is returned (hence, empty files and streams are indistinguishable from non-existing ones). The same file name can be requested several times; every time a new independent stream is generated.

The standard terminal streams are obtained by GetStream('input'), GetStream('output') and GetStream('error'); note that these are streams, hence it is possible to write on input (what is written will then be read back) and to read from output (output is generally empty).

ListStreams returns a list of the non-empty streams associated to names, as a list of strings (file names).

Reads and writes on streams do not affect the files they were generated from by GetStream. Conversely, a PutStream operation on a file does not affect the streams which have been extracted from that file; it only affects the result of a succeeding GetStream. Multiplexed read and multiplexed write operations can be obtained by passing the same stream to several readers and writers respectively (i.e. to different parts of a program).

The operation *NewStream* returns a new empty stream. It accounts for temporary (unnamed) files. A stream-filename association can be removed by reassociating an empty stream with that file name.

The operation *CopyStream* creates a stream B which is a copy of the current state of the stream A. Reads and writes on A will not affect reads and writes on B, and vice versa. The stream A is not affected.

Input operations are destructive; the characters read are removed from the stream. *InChar* reads a single character from a stream. *InString* takes a stream and a string of terminator characters and reads a sequence of characters until one of the terminators is found. *InInt* reads an integer from a stream (which should start with a digit, or with '−' immediately followed by a digit), stopping before the first non-digit character; it fails with string 'inint' if it cannot read an integer. *EmptyStream* tests for the empty stream; the input operations do not fail on empty streams: they wait indefinitely for something to be written on the stream. All the input operations are unbuffered, e.g. when reading from terminal all editing and control characters are not interpreted.

Output operations are constructive; the characters written are appended to the end of the stream. *OutChar* writes a single character at the end of a stream. *OutString* writes a whole string. *OutInt* writes an integer (preceded by '−' if negative). All the write operations are unbuffered; the effect of buffered output can be obtained by OutString.

Refer back to the operational semantics section for clarifications about how streams are contained in the memory M.

```
PutStream ( )
  (AS.s':string.s:streamname, RS, FR, ^.PR, TS, ES, M[q:stream/s]) =>
     (AS.triv, RS, FR, PR, TS, ES, M[q/s][q/s'])
  ?'putstream' on any I/O error
```

GetStream ( )
 (AS.s:string, RS, FR, ^.PR, TS, ES, M[q:stream/s]) =>
  (AS.s':streamname, RS, FR, PR, TS, ES, M[q/s][q/s'])
  where s' is a new stream name.
 (AS.s:string, RS, FR, ^.PR, TS, ES, M) =>
  (AS.s:streamname, RS, FR, PR, TS, ES, M[<>/s])
  where s is not defined in M.
 ?'getstream' on any I/O error

ListStreams ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
  (AS.liststreams(M), RS, FR, PR, TS, ES, M)
  where liststreams(M) is the list of all the strings s
  such that s is a stream name in M and M(s)≠<>.
 ?'liststreams' on any I/O error

NewStream ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
  (AS.s:streamname, RS, FR, PR, TS, ES, M[<>/s])
  where s is a new stream name.
 ?'newstream' on any I/O error

CopyStream ( )
 (AS.s:streamname, RS, FR, ^.PR, TS, ES, M[q:stream/s]) =>
  (AS.s':streamname, RS, FR, PR, TS, ES, M[q/s][q/s'])
  where s' is a new stream name.
 ?'copystream' on any I/O error

EmptyStream ( )
 (AS.s:streamname, RS, FR, ^.PR, TS, ES, M[q:stream/s]) =>
  (AS.q=<>, RS, FR, PR, TS, ES, M[q/s])
 ?'emptystream' on any I/O error

InChar ( )
 (AS.s:streamname, RS, FR, ^.PR, TS, ES, M[c.q:stream/s]) =>
  (AS.c:string, RS, FR, PR, TS, ES, M[q/s])
 ?'inchar' on any I/O error

InString ( )
 (AS.s:streamname.s':string, RS, FR, ^.PR, TS, ES, M[$c_1..c_n$.q:stream/s]) =>
  (AS.$c_1..c_{n-1}$:string, RS, FR, PR, TS, ES, M[c_n.q/s])
  where c_n (n≥1) is the first of the c_i to be a member of s'.
 ?'instring' on any I/O error

InInt ( )
 (AS.s:streamname, RS, FR, ^.PR, TS, ES, M[$c_1..c_n$.q:stream/s]) =>
  (AS.stringtoint($c_1..c_{n-1}$),
   RS, FR, PR, TS, ES, M[c_n.q/s])
   where c_n is the first of the c_i not to be part
   of a valid int representation, or a trailing blank (n≥1).
 ?'inint' on any I/O error, or if $c_1..c_{n-1}$ is not a valid int representation.

OutChar ( )
 (AS.s:streamname.c:string, RS, FR, ^.PR, TS, ES, M[q:string/s]) =>

```
      (AS.triv, RS, FR, PR, TS, ES, M[q.c/s])
   ?'outchar' on any I/O error, or if length(c)≠1

OutString ( )
   (AS.s:streamname.s':string, RS, FR, ^.PR, TS, ES, M[q:stream/s]) =>
      (AS.triv, RS, FR, PR, TS, ES, M[q.s'/s])
   ?'outstring' on any I/O error

OutInt ( )
   (AS.s:streamname.n:int, RS, FR, ^.PR, TS, ES, M[q:stream/s]) =>
      (AS.triv, RS, FR, PR, TS, ES, M[q.inttostring(n)/s])
```

In order to model user interaction, and in general other processes which act on the file system, we add some state transitions which happen nondeterministically and change the file system:

```
(AS, RS, FR, PR, TS, ES, M[s'.q:stream/s:streamname]) =>
   (AS, RS, FR, PR, TS, ES, M[q/s])

(AS, RS, FR, PR, TS, ES, M[q:stream/s:streamname]) =>
   (AS, RS, FR, PR, TS, ES, M[q.s'/s])

(AS, RS, FR, PR, TS, ES, M) =>
   (AS, RS, FR, PR, TS, ES, M[q:stream/s:streamname])
```

The first transition models an external process which reads from a stream, the second one a process which writes on a stream, and the third one a process which creates, deletes, replaces or renames a stream.

I/O errors can be treated by taking the predicate "on any I/O error" above to be constantly true, i.e. an I/O error may unpredictably happen at any time.

10. Other operations

Start initializes the abstract machine. It takes three parameters: ES, which is the initial environment containing all the predefined values and functions; M, containing values for all the locations mentioned by EP, and the initial file system; and a closure which is the program to execute (by convention the start closure takes a triv argument). The initial M must contain a stream called 'input' and a stream called 'output', which are normally attached to the user terminal. These standard streams can be obtained normally by GetStream, and all the stream operations are valid, including writing on input, reading from output and performing PutStream and CopyStream on them.

Stop terminates the execution. Normally the value on the top of AS after a Stop is the final result of the computation.

There is a notion of *top-level environment* which is implemented by EP (environment pointer), pointing to the argument stack AS. EP always points to some point of AS below AP, and AP never descends below EP. The operation *Define* rises EP to incorporate more values in the top-level environment.

Collect takes any value (which can be used as a garbage collector parameter, if any) and provokes a garbage collection, returning triv.

Skip has no effect.

Import takes a stream which is assumed to contain an executable, fully relocatable piece of code. That code is blindly converted into a closure with no global values, and the closure is returned on AS. Import can operate on programs produced by foreign systems.

Export takes a stream and a closure, and saves the closure in the stream, so that it can be reloaded

by Import. This is to implement separate compilation, and the exported closure is not expected to be runnable outside the Fam system.

StandAlone takes a stream and a closure, and saves the closure in the stream, together with all the environment needed to support the closure (e.g. the global variables and the run time system). The file produced can be executed independently of the Fam system. The input parameter accepted by a stand-alone function is installation-dependent; it can be for example a list of strings (e.g. options) passed by the operating system.

Dump saves the current state of the Fam system (file system excluded) in a stream and continues normal execution. A dump file can then be executed, reactivating the system at the 'instant' of dump.

```
Start (ES: stack,
       M[q:stream/'input'][q':stream/'output']: memory,
       closure(text(c:code,..)..): triv→α)
 (<>, <>, −, ^.<>, <>, <>, <>) =>
       (ES.triv, <>, closure(text(c:code,..)..), c, <>, ES, M)
       where M defines the addresses and stream names mentioned by ES.

Stop ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
       (AS, RS, FR, <>, TS, ES, M)

Define ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
       (AS, RS, FR, PR, TS, AS, M)

Collect ( )
 (AS.x:α, RS, FR, ^.PR, TS, ES, M) =>
       (AS.collect(x), RS, FR, PR, TS, ES, M)

Skip ( )
 (AS, RS, FR, ^.PR, TS, ES, M) =>
       (AS, RS, FR, PR, TS, ES, M)

Import ( )
 (AS.s:streamname, RS, FR, ^.PR, TS, ES, M) =>
       (AS.import(M(s)), RS, FR, PR, TS, ES, M)
 ?'import' on any I/O error.

Export ( )
 (AS.s:streamname.f:α→β, RS, FR, ^.PR, TS, ES, M) =>
       (AS.triv, RS, FR, PR, TS, ES, M[M(s).export(f)/s])
 ?'export' on any I/O error.

StandAlone ( )
 (AS.s:streamname.f:α→β, RS, FR, ^.PR, TS, ES, M) =>
       (AS.triv, RS, FR, PR, TS, ES, M[M(s).standalone(f)/s])
 ?'standalone' on any I/O error.

Dump ( )
 (AS.s:streamname, RS, FR, ^.PR, TS, ES, M) =>
       (AS.triv, RS, FR, PR, TS, ES,
        M[M(s).dump(AS,RS,FR,PR,TS,ES,M)/s])
 ?'dump' on any I/O error.
```

11. Garbage Collection

This section describes a garbage collection algorithm for languages with a low percentage of side-effectable data. The algorithm is due to Lieberman and Hewitt [3] for a much more general situation. The assumption of working with semi-applicative languages confers simplicity and elegance to the algorithm.

The basic idea is that of a copying garbage collector: there are two equal data areas called *spaces* of which only one is active at any given time (we don't consider here incremental garbage collection). When one space is full, it is copied into the other space by following the reachable pointers. This copying operation can be done simply by a recursive procedure if we are not short of space, otherwise by a well known pointer-reversing technique which runs in constant space. Care must be taken to copy correctly circular and shared structure. Finally we swap the spaces.

Copying garbage collection is appealing because the time spent in copying only depends on the amount of active data, not on the size of the spaces; together with recursive copying this amounts to a very fast collection. Moreover the data is automatically compacted during copying, reducing the rate of page faults in virtual memory systems.

The problem with copying collectors it that they need a very large address space. The two-spaces algorithm 'wastes' 50% of the memory.

The algorithm proposed by Hewitt and Lieberman generalizes copying collectors to n spaces (of which only one is 'wasted' for copying the other ones in turn), in such a way that not all the memory has to be searched in general when copying a little part of it. This can work only under assumptions about which spaces contain pointers to which other spaces. These assumptions must be preserved during allocation and collection.

It turns out that the assumptions hinted at above are much easier to verify in applicative languages. We obtain a garbage collector which can work on extremely large collections of data with only a limited working area. Moreover some areas can be collected more often then others, so that 'stable' data tends to migrate towards rarely collected areas while 'volatile' data is quickly reclaimed.

Here is an overview of the algorithm. The basic observation is that, most of the times, recently allocated data point to previously allocated data because it has been built on top of it. We refer to this fact by saying that pointers generally point to the past. There are two exceptions which have to be treated specially: recursive functions (which may contain environment loops) and references (which may point to the future after an assignment).

The available space is (dynamically) partitioned into a 'monotonic' area containing data which only points to its past, and a 'paradoxical' area which may contain pointers to the future. The paradoxical area is used to allocate recursive closures and references, and it is assumed to be relatively small.

At some point during the execution of a program we may decide that garbage collection is needed. Let us consider the monotonic area first. We split the monotonic area arbitrarily in three contiguous sections, called past, present and future. The idea is to copy the 'present' space only, given a big enough buffer to contain it; past and/or future may be empty. We start following the reachable pointers. If some data is in the future we keep following it without copying it. If it is in the present space we copy it like in the normal copying garbage collector. If it is in the past we even stop following the pointers because we *pretend* that they cannot lead us back to the present (actually they might, going through the paradoxical area, but the past will be on average rather big and we do not want to search it all). In the paradoxical area we also stop following pointers because we treat this area separately in the second phase of collection.

So, our target is to find out all the reachable pointers pointing to the present, and up to now we have found all those coming from the future and from the present itself, and we know that there are no pointers coming directly from the past. But there may be pointers in the paradoxical area pointing to the present, which are only reachable from the past. On the assumption that the past is on average much bigger than the paradoxical area, it is more convenient to search that area rather then the past. But we cannot do this by following reachable pointers, because we have chosen not

to follow some pointers. Hence we must scan the whole paradoxical area for pointers to the present, and take the conservative view that all those pointers are reachable. This scanning process is called *scavenging*. When we find a pointer to the present we proceed copying and following the pointers as before.

Finally we have a copy of the present, and we must substitute it for the old present; this can be done by maintaining a linked list of the pages in the monotonic area in monotonic order (this is the reason why we could split neatly past present and future at the beginning).

We still have to describe how to collect the paradoxical area. This is done quite simply by following the reachable pointers everywhere in both areas, copying when we find something in the paradoxical area. In other words, we consider the paradoxical area as present, and the monotonic area as future with no past. This is a heavy operation because involves searching the whole memory: again the paradoxical area should be small and stable, so that it can be collected infrequently.

The general strategy is then to collect very frequently the extreme future, which presumably contains very dynamic data, and less and less frequently as we move towards the past, which comes to contain more and more stable data because of compactification (to ensure this migration of stable data we have to alternatively collect overlapping presents). This should be intermixed with the collection of the paradoxical area. Some simple adaptive scheme is probably the best way of implementing this strategy.

12. Compilation Hints

Here are some suggestions about how to compile high-level language expressions into Fam operations. There is a translation function '[| |]' from expressions to Fam programs, for example '[|3|] => Int(3)' means that the expression '3' is translated into the Fam operation 'Int(3)'.

Primitive operations (like '+') which have a corresponding Fam machine operation are translated by translating their arguments left to right, and then suffixing the appropriate Fam operation:

```
[|op(arg1,...,argn)|] =>
    [|arg1|] .. [|argn|] [|op|]
```

Variables are converted to a GetLocal or GetGlobal operation, depending on where they are defined; strings are converted to GetLiteral:

```
[| .. x .. y .. 'string' .. |] =>
     .. GetLocal(n) .. GetGlobal(m) .. GetLiteral(p) ..
```

Function applications are translated by translating the argument, the function and then appending the three parts of the apply operation:

```
[|f(a)|] =>
    [|a|] [|f|] SaveFrame ApplFrame RestFrame
```

Functions are compiled into sequences of operations which, at run time, build closures. First all the global variables of the function are collected from the appropriate environments (we informally use Get(x) for GetLocal or GetGlobal with the appropriate displacement), then the text of the function is fetched by GetLiteral, and finally a Closure is generated.

```
[|λx. .. x .. y .. 's' .. |] =>
    Get(y) Get(z)
    GetLiteral(text([| .. x .. y .. 's' .. |]Return(1,1),'s'))
    Closure(2)
```

Recursive functions involve DumClosure and RecClosure. Here is the compilation of two mutually

recursive functions f and g:

```
[|let rec f = .. g .. and g = .. f .. |] =>
    DumClosure(1) DumClosure(1)
    GetLocal(0) GetLiteral(text([| .. g .. |]Return(1))) RecClosure(1,1)
    GetLocal(0) GetLiteral(text([| .. f .. |]Return(1))) RecClosure(1,0)
```

Here is how to use trap operations. 'A ? B' is a program which starts evaluating 'A' and if no failure occurs B is ignored; however if a failure occurs in A then the "exception handler" B is executed. 'A ? B' is compiled by setting a Trap which in case of failure produces a jump to label1 (hence executing B); if no failure occurs in A the execution reaches the UnTrap operations which undoes the trap and jumps to label2 (hence ignoring B). Failures are produced by the Failwith operation or by exceptions arising from primitive operations (e.g. divide by zero).

```
[|A ? B|] =>
    Trap(label1) [|A|] UnTrap(label2) label1:[|B|] label2:
```

13. Concrete Syntax

This section defines a textual syntax for abstract machine programs; this is essentially an assembly language for Fam.

The syntactic notation is as follows:
strings between quotes "" are terminals;
identifiers are non-terminals;
juxtaposition is syntactic concatenation;
'|' is syntactic alternative;
'[ ]' is the empty string;
'[...]' is zero or one times (i.e. optionally) ' ... ';
'{ ... }n' is n or more times ' ... ' (default n=0);
'{ ... / --- }n' means n (default 0) or more times '...' separated by '---';
Parentheses '(...)' are used for precedence.

```
Digit ::= 0|1|2|3|4|5|6|7|8|9

LabelChar ::= <any printable character different from space,
                newline, tab and ':'>

StringChar ::= <any printable character different from "",
                or an escape sequence starting with '\'>

Int ::= Digit | Digit Int

String ::= "" {StringChar} ""

Label ::= {LabelChar}

Program ::= '[' {Instruction / ';'} ']'

Instruction ::= Operation | Label ':' Instruction

Operation ::=
    'GetLocal' Integer
  | 'Inflate' Integer ',' Integer
  | 'Deflate' Integer ',' Integer
```

```
| 'Permute'  '[' {Integer / ';'} ']'
| 'Triv'
| 'True'
| 'False'
| 'Not'
| 'And'
| 'Or'
| 'Xor'
| 'BoolEq'
| 'Int' Integer
| 'Minus'
| 'Plus'
| 'Diff'
| 'Times'
| 'Divide'
| 'Modulo'
| 'Greater'
| 'Less'
| 'GreaterEq'
| 'LessEq'
| 'IntEq'
| 'String' String
| 'Length'
| 'SubString'
| 'Explode'
| 'Implode'
| 'ExplodeAscii'
| 'ImplodeAscii'
| 'IntToString'
| 'StringToInt'
| 'StringEq'
| 'Ref'
| 'At'
| 'Assign'
| 'DestRef' Integer ',' Integer
| 'Pair'
| 'Left'
| 'Right'
| 'DestPair' Integer ',' Integer ',' Integer
| 'Nil'
| 'Cons'
| 'Head'
| 'Tail'
| 'Null'
| 'DestNil' Integer
| 'DestCons' Integer ',' Integer ',' Integer
| 'Record' Integer
| 'Field' Integer
| 'DestRecord' Integer ',' '[' {Integer / ';'} ']'
| 'Variant' Integer
| 'As' Integer
| 'Is' Integer'
| 'DestVariant' Integer ',' Integer ',' Integer
| 'Case' '[' {Label / ';'} ']'
```

```
| 'Array'
| 'Tabulate'
| 'LowerBound'
| 'Size'
| 'Sub'
| 'Update'
| 'ArrayToList'
| 'Equal'
| 'Isomorphic'
| 'Id'
| 'Comp'
| 'Text' Label
| 'Closure' Integer
| 'DumClosure' Integer
| 'RecClosure' Integer ',' Integer
| 'GetGlobal' Integer
| 'Jump' Label
| 'TrueJump' Label
| 'FalseJump' Label
| 'SaveFrame'
| 'RestFrame'
| 'ApplFrame'
| 'Return' Integer ',' Integer
| 'TailApply' Integer ',' Integer
| 'Trap'
| 'TrapList'
| 'UnTrap'
| 'FailWith'
| 'PutStream'
| 'GetStream'
| 'ListStreams'
| 'EmptyStream'
| 'CopyStream'
| 'EndStream'
| 'InChar'
| 'InString'
| 'InInt'
| 'OutChar'
| 'OutString'
| 'OutInt'
| 'Start'
| 'Stop'
| 'Define'
| 'Collect'
| 'Skip'
| 'Import'
| 'Export'
| 'StandAlone'
| 'Dump'
```

Note1: String and Text are converted to GetLiteral by the assembler.

Note2: Escape sequences for Strings:

\z	0	nul

\x	4	eot
\b	8	backspace
\t	9	tab
\n	10	newline
\r	13	carriage return
\e	27	escape
\f	28	form feed
\d	127	del
\^<c>	<c> mod 64	control<c> for any printable <c>
\<c>	<c>	<c> for any other printable <c>

Note3: Comments can be introduced within curly brackets '{' and '}', and they can be nested.

14. Abstract Syntax

It may be useful to be able to generate and manipulate Fam code from a language built on top of the Fam, e.g in order to write an optimizer. This section contains a representation of Fam code in terms of Fam data structures.

```
FamCode = FamOp list

FamOp =
  [|GetLocal:      (|Displ: int|);
    Inflate:       (|Size: int; Displ: int|);
    Deflate:       (|Size: int; Displ: int|);
    Permute:       (|Permutation: int list|);
    Triv;
    True;
    False;
    Not;
    And;
    Or;
    Xor;
    BoolEq;
    Int:           (|Integer: int|);
    Minus;
    Plus;
    Diff;
    Times;
    Divide;
    Modulo;
    Grater;
    Less;
    GreaterEq;
    LessEq;
    IntEq;
    String:        (|String: string|);
    Length;
    SubString;
    Explode;
    Implode;
    ExplodeAscii;
    ImplodeAscii;
    IntToString;
    StringToInt;
```

```
StringEq;
Ref;
At;
Assign;
DestRef:      (|RefDispl: int; AtDispl: int|);
Pair;
Left;
Right;
DestPair:     (|PairDispl: int; LeftDispl: int; RightDispl: int|);
Nil;
Cons;
Head;
Tail;
Null;
DestNil:      (|ListDispl: int|);
DestCons:     (|ListDispl: int; HeadDispl: int; TailDispl: int|);
Record:       (|Size: int|);
Field:        (|FieldNumber: int|);
DestRecord:   (|RecordDispl: int; FieldDispls: int list|);
Variant:      (|CaseNumber: int|);
As:           (|CaseNumber: int|);
Is:           (|CaseNumber: int|);
DestVariant:  (|CaseNumber: int; VariantDispl: int; AsDispl: int|);
Case:         (|Cases: FamCode list|);
Array;
Tabulate;
LowerBound;
Size;
Sub;
Update;
ArrayToList;
Equal;
Isomorphic;
Id;
Comp;
Text:         (|Text: FamCode|);
Closure:      (|Size: int|);
DumClosure:   (|Size: int|);
RecClosure:   (|Size: int; Displ: int|);
GetGlobal:    (|FRDispl: int|);
Jump:         (|Target: FamCode|);
TrueJump:     (|Target: FamCode|);
FalseJump:    (|Target: FamCode|);
SaveFrame;
RestFrame;
ApplFrame;
Return:       (|Size: int; Displ: int|);
TailApply:    (|Size: int; Displ: int|);
Trap;
TrapList;
UnTrap;
FailWith;
PutStream;
GetStream;
```

```
  ListStreams;
  EmptyStream;
  CopyStream;
  EndStream;
  InChar;
  InString;
  InInt;
  OutChar;
  OutString;
  OutInt;
  Start;
  Stop;
  Define;
  Collect;
  Skip;
  Import;
  Export;
  StandAlone;
  Dump;
]
```

15. VAX Data Formats

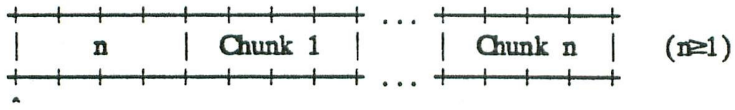
Comments to the pictures. Each segment "+--+" is one byte. The symbol "^" below a data structure represents the location pointed by pointers to that structure; fields preceding "^" are only used during garbage collection and are inaccessible to the Fam operations (and hence to the user). Unboxed data is kept on the stack, or in the place of pointers in other data structures; unboxed data does not require storage allocation. Pointers can be distinguished from unboxed data as the former are > 64K. There are automatic conversions between SmallIntegers and BigIntegers, so that the Fam operations only see the type Integer.

```
Triv

 0        (triv)              (unboxed)
```

```
Boolean

 0        (false)             (unboxed)
 1        (true)              (unboxed)
```

```
SmallInteger

 -32768 .. +32767             (unboxed)
```

BigInteger



Pair

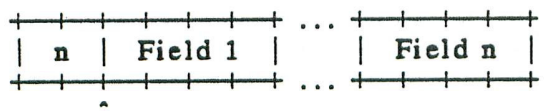
List

0 (emptylist) (unboxed)

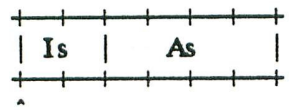
Tl may only be emptylist
or point to a list

Record

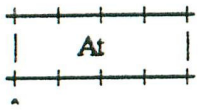
0 (nullrecord) (unboxed)



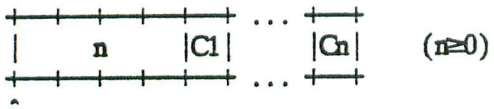
Variant



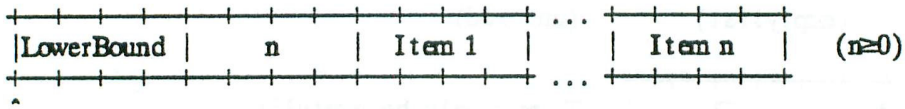
Reference



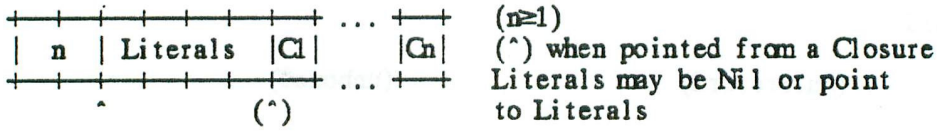
String



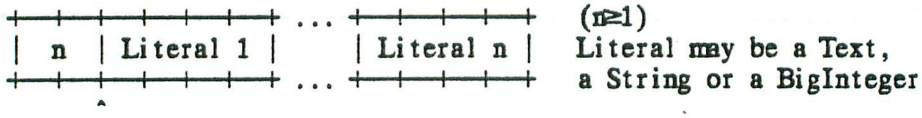
Array



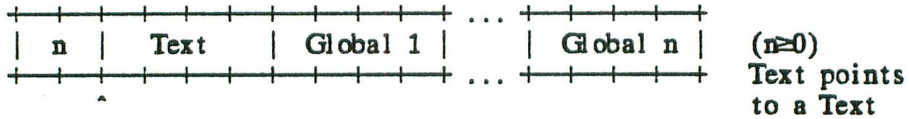
Text



Literals



Closure



References

[1] P.J.Landin: "The Mechanical Evaluation of Expressions", Computer Journal, Vol. 6, No. 4, 1964, pp. 308-320.

[2] G.D.Plotkin: "A Structural Approach to Operational Semantics", Internal Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

[3] H.Lieberman, C.Hewitt: "A Real Time Garbage Collector Based on the Lifetime of Objects", A.I. Memo No. 569A, October 1981.

SCIENCE AND ENGINEERING RESEARCH COUNCIL
RUTHERFORD APPLETON LABORATORY

COMPUTING DIVISION

SOFTWARE TECHNOLOGY INITIATIVE

ML, LCF, and HOPE
Meeting at RAL on 17 November 1982

Present: R W Witty, RAL (Chairman)
C P Wadsworth, RAL (Notes)
R Milner, Edinburgh
R M Burstall, Edinburgh
J Scott, Edinburgh
M Hennessey, Edinburgh
K Mitchell, Edinburgh
D Schmidt, Edinburgh
A Mycroft, Edinburgh
B Sufrin, Oxford
M Raskovsky, Oxford
J Hughes, Oxford
J Darlington, Imperial College
J Cunningham, Imperial College
S Zappacosta, Imperial College
C Jones, Manchester
A Wills, Manchester
J Welsh, UMIST
R Gallimore, UMIST
D Coleman, UMIST

Apologies for absence were received from

M Gordon, Cambridge
L Paulson, Cambridge
C A R Hoare, Oxford
K Hanna, Kent
J Cramer, Imperial College
I Pyle, York
R Boot, NCC

Gordon and Paulson had sent written comments for the meeting, a copy of which is appended to these notes.

1. INTRODUCTION

RWW welcomed participants. The meeting was the first of many he hoped to organise under the SERC's Software Technology Initiative (STI) in which researchers in particular areas are brought together to identify common needs and goals and to provide guidance to the STI on how best to exploit their research achievements.

CPW explained the background to this meeting. The SERC's Software Technology Panel has already reviewed favourably several proposals for the further exploitation of ML, LCF, and HOPE, and is keen to see that the right decisions are taken now both to meet the needs of particular research projects and to propagate knowledge and availability of these systems to the wider community. There is particular interest in mounting versions on the ICL PERQ since PERQs running Unix will be widely available in the SERC community shortly. The Panel also felt there is a specific need for a tutorial on ML, LCF, and HOPE.

2. REVIEW OF ML, LCF, AND HOPE

2.1 Background

CPW characterised ML, LCF, and HOPE for discussion purposes as being at the centre of an area of research activity that can be distinguished as typed functional programming, derived from Landin's ISWIM (not itself typed).

ML is an ISWIM-style higher order functional programming language whose distinctive features are

(a) a polymorphic type discipline, with compile-time type checking, and

(b) a facility for abstract type modules.

LCF is an application and extension of ML as a programming metalanguage for conducting proofs in a particular object logic (called PPLAMBDA). The LCF proof style relies heavily on the use of higher order functions, for goal-directed proof via tactics. The logic is embedded in ML as particular data types term, form, and thm with relevant primitives (abstract syntax constructors and destructors on terms and formulae, axioms and inference rules as operations producing theorems), with quotations providing a concrete syntax for input. A hiearchical filing system for theories of the logic is an important practical component of the system, to store useful theorems in theory files for later retrieval.

HOPE may be viewed as ML without assignment (a deliberate design choice) with additional features:

(a) pattern matching with respect to data constructors,

(b) overloading of variable meanings (different meanings for the same symbol at different types),

for some years yet.

J Darlington outlined work at Imperial and Westfield Colleges. The ALICE project at Imperial is building a programming environment centred around HOPE, including a transformation component based on the ideas of the LCF metalanguage (tactics and simplification). ALICE includes a Compiler Target Language, CTL. Compilers written in HOPE producing ALICE CTL code have been implemented for both HOPE and PROLOG. ALICE CTL is quite different to Cardelli's AM and uses the technique of graph reduction in its implementation. In the medium term a fast interpreter for CTL would be needed, possibly microcoded. All the work at Imperial is being done on the Edinburgh DEC10. S Eisenbach (Westfield College) is producing HOPE compilers and run-time systems targetted at 16-bit micros. There is close liaison between Edinburgh and Imperial to maintain compatability of HOPE languages.

There was some discussion about the merits of propagating two languages (HOPE and ML). R Milner summarised the general view that HOPE and ML represent two points in a spectrum of possible (typed) functional languages. Differing aims had led to different compromises in the design of the two languages and both should continue to be developed. The ultimate functional language could not be envisaged yet - variety and experimentation, rather than standardisation, were needed at this stage.

5. TUTORIALS on ML, LCF, HOPE

The meeting welcomed the suggestion that a tutorial be held to propagate knowledge of ML, LCF, and HOPE, and discussed the form this should take.

It was felt that LCF per se was a specialised interest and should be separated from the general concepts embodied in ML and HOPE. It was also felt inadvisable to attempt to use two languages in the same tutorial.

Accordingly it is recommended that three separate tutorials be held, one each on ML and HOPE with a further tutorial on LCF. Each should be at least two days with approximately half the time spent on practical work. Summer 1983 is the suggested date, since it is reasonable to expect the PERQ implementations to be up and running by then. (If backup arrangements should prove necessary, access to the VAX implementations was the preferred alternative since these are the versions that will become widely available on the PERQ.)

Edinburgh and Imperial College offered to act as host sites.

C O M P U T E R

L A B O R A T O R Y

University of Cambridge

3 November 1982

First, we would like to apologize that we cannot attend the meeting, and hope that this letter will convey our views. The paragraphs are numbered in correspondence to the Agenda.

2. Gerard Huet (of INRIA) has ported the DEC-10 implementation of LCF to Multics MacLisp, cleaned up the code, and made it portable over the MacLisp family. He and Larry Paulson have agreed to maintain a common source file. Some modules have been entirely rewritten. In particular:

> Gerard has rewritten the theory package for greater efficiency, flexibility, and robustness. He plans to allow complex hierarchies of theories, and theories that can be shared by several users. We hope that recent work by Sannella and Burstall will allow us to incorporate ideas from the specification language, CLEAR.
>
> Larry has installed a pretty-printer, and rewritten all the code for manipulating PPLAMBDA objects. He has extended PPLAMBDA to include disjunction, existential quantifiers, and predicates.
>
> Gerard and Larry have extended the ML-to-Lisp translator so that the Lisp code it generates can be given to the Lisp compiler. Compiling the code makes it run several times faster, and allows it to be re-loaded almost instantaneously, since the slow parser and type-checker are not invoked. Compiled code occupies much less storage when loaded.

This LCF is running on INRIA's Multics system (in MacLisp), and at Cambridge's VAX/Unix system (in Franz Lisp). We would like to bring it up on Edinburgh's VAX/VMS (Franz Lisp), and on the Edinburgh DEC-10 (MacLisp or Rutgers Lisp), if people there are interested and willing to help.

Further system work may have to wait, since our real goal is to use LCF, not to develop it.

The Goteborg group has extended the LCF system in different directions, including an improved top-level and a new logic, Martin-Lof's Intuitionistic Type Theory. Larry will be in Goteborg on the date of this SERC meeting, and hopes to find a common ground with their efforts. Though they are also using Franz Lisp under VAX/Unix, the two Lisp sources are incompatible.

3. Luca Cardelli's ML system should be made available on PERQs. His language is quite different from the ML in LCF. It is perhaps too baroque, but has convenient data structures and more flexible declarations. Its reference variables may provoke controversy, but are more powerful than those in ML. A further plus is the speed of its compiled code.

However, we feel it is too early to consider implementing LCF in Luca's ML. Luca's system is unsupported and provides no input/output. Although LCF is partly implemented in ML, this code would need extensive editing to run under Luca's system. The Lisp code may be impossible to re-write. Even if the porting succeeded, it would be yet another implementation of LCF to maintain.

We feel that the best way to mount LCF on to the PERQ is to first mount Lisp, then use the existing Lisp implentation of LCF. We hear that Carnegie-Mellon University has implemented a system called Spice Lisp on PERQ's. This is an implementation of Common Lisp, which is compatible with MacLisp and should accept our system with little change. Furthermore, Lisp is useful in its own right, and will make PERQ's more attractive to AI researchers.

(c) dynamic data structures (possibly "infinite") through lazy evaluation.

Currently, HOPE lacks a mechanism for trapping exceptions/failures. Pattern matching and the use of equations to define functions give HOPE a quite distinct character compared to ML that makes it more suitable for program specification and transformation.

2.2 Implementations

2.2.1 LCF/ML

(a) DEC10/TOPS10 in Stanford Lisp, since ported to Rutgers Lisp (Edinburgh).

(b) Multics system in MacLisp family (INRIA).

(c) VAX Unix in Franz Lisp (Cambridge), with an extended logic including disjunction and existential quantification.

(d) VAX Unix in Franz Lisp (Goteborg, Sweden), with a completely new object logic - Martin Lof's Intuitionistic Type Theory.

Cambridge(c) and INRIA(b) are maintaining common source files by using only the MacLisp subset of Franz Lisp.

2.2.2 Cardelli ML

A second version of ML only (called Cardelli ML for these notes) without the LCF object logic has been developed by L Cardelli (Bell Labs). This has named records and sums (variants), a reference operator for arbitrary types, and more flexible declarations and data abstraction constructs (sufficient to encode those of LCF ML).

Cardelli ML is implemented in Pascal under VAX VMS, since ported to VAX Unix. This produces code for an Abstract Machine (called Cardelli AM for these notes). Cardelli AM is implemented in VAX assembler.

2.2.3 HOPE

(a) DEC10/TOPS10 in POP2 and in PROLOG (Edinburgh).

(b) VAX Unix in Franz Lisp, compiling HOPE to Cardelli AM, with Lisp interpreter for Cardelli AM (from D MacQueen, Bell Labs).

(c) Part of ALICE project at Imperial College (see section 4 below).

3. PORTING TO THE PERQ

Users would increasingly find the restriction to linear text for I/O a limiting factor in the existing implementations of ML, LCF, and HOPE. The more sophisticated forms of interaction possible with the PERQ (pointing, menu selection, windows, etc) make it a natural vehicle for the further development of all these systems.

It was noted that steps are already in hand to mount Franz Lisp under PERQ Unix, through the Lisp support team of R Rae in Edinburgh AI. It was decided that the best way to mount LCF and HOPE on the PERQ is therefore to port their existing Franz Lisp implementations from VAX Unix. This should not require more than a few days work once Franz Lisp is running under PERQ Unix. For LCF it will be essential that Franz Lisp on the PERQ includes the DUMPLISP operation for saving core images.

Several research projects (Hanna, Hennessey/Mitchell, Sufrin) had a separate and more immediate need for ML on the PERQ without the LCF logic. Ideally this should be as efficient as possible, though R Milner suggested, with general agreement, that the initial requirement is for an ML workbench that is pleasant to use so that people can get into it quickly. It was felt that such an ML system would also be more suitable for dissemination of ML to a wider community.

Cardelli ML could be used as a starting point. It was easier to use than LCF ML and considerably faster in its compiled code, though currently it lacked a supporting environment. Porting to the PERQ would require

(a) an interpreter or code generator for Cardelli AM,

(b) a garbage collector, and

(c) possible changes due to differences in Pascal dialect.

It was estimated that this would involve two man-months of work.

B Sufrin suggested that the medium term requirement for a fast implementation be met by microcoding Cardelli's AM when the 16K writeable control store is available for PERQ. (This would also be useful as a target machine for HOPE - see section 4 below.) RWW commented that where there was a clearly felt need SERC funds can be generated through the STI to pay ICL to do the work. A complete specification of Cardelli's AM to commercial standards would be needed to put the proposal to ICL.

RWW would pursue the question of porting Cardelli ML to the PERQ with the two research projects that had indicated a willingness to do some of the work (Hennessey/Mitchell, Sufrin).

Extending Cardelli ML to full LCF was not considered worthwhile. It was felt better to keep Franz Lisp as a common base for LCF on VAX and on PERQ.

4. DEVELOPMENT OF HOPE

R Burstall described briefly current work with HOPE at Edinburgh. Examples from the category theory approach to program specification are being worked on - approximately 100 pages had now been programmed in HOPE. Extensions to HOPE for better modularity, and a mechanism for handling exceptions, are being considered. Current implementations were slow and would need to be improved. Edinburgh were considering either proceeding via Spice Lisp or seeking a microcoded version of Cardelli's AM. It was noted, however, that Spice Lisp would not be a viable route

4. We have not used HOPE ourselves, but are impressed with the language's design and strongly encourage more development. HOPE has the same general aim as ML, but is simpler and cleaner. We would like to see some of David Turner's ideas incorporated (not necessarily normal order evaluation), and a fast implementation using the techniques developed by Lucc and others. Eventually, HOPE could become a language for building systems like LCF.

We would like to see research into the roles of failure trapping, input/output, assignable variables, and normal order evaluation in such languages.

We hope someone will work on debugging tools for HOPE and ML. Their polymorphism makes it hard to write debuggers, because there is no type information at run-time and only partial information at compile-time.

5. We are not familiar with this work but would be interested in learning more about it.

6. We favor a tutorial on HOPE and ML. It must take place where there are many terminals running these languages, for no one can understand what polymorphism is really about without personally typing in function definitions. Recent extensions to LCF, discussed below, are bringing it closer to conventional logics and making it easier to understand. We will require a few months to discover the consequences of these extensions, and suggest that no LCF tutorial be held before then.

7. Larry Paulson is interested in performing structural induction in LCF. He has extended the logic with disjunction and existential quantifiers, in order to allow recursive structures to be axiomatized directly, rather than through domain equations. Such axiom systems are simpler and more natural than the previous constructions that used lifting and coalesced sums. He plans to automate the construction of theories of recursive structures, including (when applicable) proof that a domain is flat and construction of its theory of equality. In particular, this would facilitate compiler proofs, since it would automatically construct the theory of a language's abstract syntax. He has formalized part of Manna and Waldinger's proof of the Unification Algorithm, and plans to continue this.

Mike plans to attempt some non-trivial digital system correctness proofs in an extension of PPLAMBDA containing terms denoting sequential machines, together with inference rules based on the laws of SCCS. Some simple examples have already been done in a prototype implementation, and the approach seems promising.

Mike has a student interested in comparing and (perhaps) combining functional and logical programming. One possibility is to provide a mechanism for deducing consequences of PPLAMBDA theorems with a very efficient derived inference rule based on Prolog. The general idea is to approach the ideal of Program=Logic+Control by experimenting with PPLAMBDA as the logic and ML as control.

Dave Matthews at Cambridge is developing a high performance system programming language, called Poly, which is derived from Pascal and Russell. It is beginning to look a lot like ML, and it is hoped to eventually unify the two approaches. Poly might be a source of ideas for extending the ML type discipline, it might also be a good programming language for future LCF implementations. At present it is not clear whether continued SERC support for Dave will be forthcoming.

Larry Paulson

Michael C. Gordon

Mike Gordon

Addenda to the Mailing List

Hans Boehm
Computer Science Department, FR-35
Sieg Hall
University of Washington
Seattle, WA 98195
USA

Corrado Bohm
Istituto Matematico G. Castelnuovo
Universita' di Roma
Piazzale Aldo Moro 5
00185 Roma
Italy

Alex Borghida
Dept. of Computer Science
Hill Center
Rutgers University
New Brunswick, NJ 08903
USA

John Darlington
Dept. of Computing
Imperial College of Science
and Technology
180 Queens Gate
London SW7 2BZ
England

James Donahue
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
USA

Gerard Huet
INRIA
P.O. Box 105
Domain de Voluceau
Rocquencourt
78150 Le Chesnay
France
Tel 3-9549020

Finn V. Jensen
Institue of Electronic Systems
Aarlborg University Center
Aarlborg
Denmark

Richard B. Kieburtz
Dept of Computer Science and Eng.
The Oregon Graduate Center
19600 NW Walker Road
Beaverton, Oregon 97006
USA
Tel 503-645-1121

Mike Levy
Dept. of Computer Science
Univ. of Victoria
Box 1700
Victoria, B. C. V8W 2Y2
Canada
Tel 604-477-6911 x 4757

Lockwood Morris
School of Computer and
Information Science
Syracuse University
313 Link Hall
Syracuse, NY 13210
USA
Tel 315-423-2368

Gordon Plotkin
Room NE43-837
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
USA

Nick Rothwell
Computing Laboratory
Claremont Tower
University of Newcastle Upon Tyne
Newcastle Upon Tyne NE1 7RU
England

Peter Wegner
Dept. of Computer Science
Brown University
BOX 1910
Providence, RI 02912
USA