AT&T Bell Laboratories Technical Report TR-107, 1983. Last revised: May 4, 1983.
Converted from troff: July 1985 (original Blit-generated pictures scanned from hard copy).
Printed *Sunday, July 9, 1995, 10:18 pm*

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.

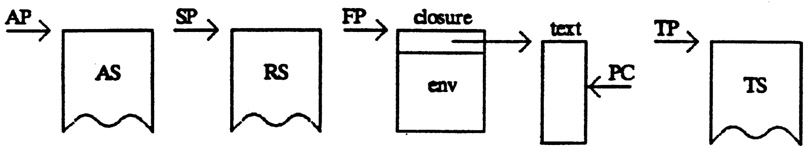
The machine is intended to make compilation from high level languages easy and regular, by providing a rich and powerful set of operations and an open-ended collection of data types. This richness of types can also facilitate portability, because every type can be independently implemented in different 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 State

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 *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 frame 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 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 unessential, 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 [2]. 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:

closure(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:α_1,..,x_n:α_n).PR′, where x_i are the parameters of op.

The *memory* M is a pair of functions:

M = L,F: (address $\rightarrow$ value) $\times$ (streamname $\rightarrow$ 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, denoted 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:

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

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 behavior (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:

$$\begin{array}{l} op(x_1{:}\alpha_1,..,x_n{:}\alpha_n) \\ \quad (AS, RS, FR, \hat{}.PR, TS, ES, M) \Rightarrow \\ \qquad (AS', RS', FR', PR', TS', ES', M') \\ \quad ?\ \text{'op' if } p \end{array}$$

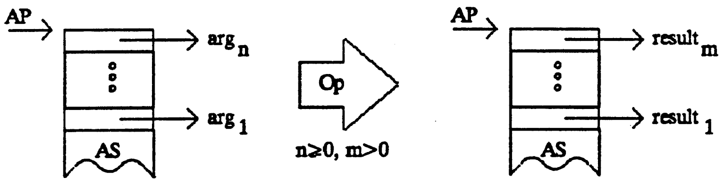
to indicate that we have a failure with reason 'op' (a string) when the predicate p is true of the starting state (AS, RS, FR, ^.PR, TS, ES, M), otherwise the normal state transition will happen. This is only an abbreviation for the following state transitions:

```
op(x_1:α_1,..,x_n:α_n)
    (AS, RS, FR, ^.PR, TS, ES, M) =>                (if not p)
        (AS', RS', FR', PR', TS', ES', M')
op(x_1:α_1,..,x_n:α_n)
    (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 denominator 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 character is position 1) and a substring size (may fail with 'substring'). *Search* searches a pattern in a string; when given a string pattern 'p' as argument it returns a function which acts as a matching machine 'm' for 'p'. The machine 'm' can be applied to pairs string-integer 's,i' to search the longest leftmost substring 's1' of 's' starting at or after position 'i' in 's' and matching 'p'. If 's1' exists, the result is a pair of integers 'j,l' where 'j' is the starting position of 's1' in 's' and 'l' is its length. The tuple 's,j,l' can be passed to SubString to obtain the matching substring. Search fails with string 'search' if the pattern 'p' is malformed; a matching machine 'm' fails with string 'match' when 'i' is out of bounds, or when there is no match. The syntax and semantics of patterns is described in section "String Patterns". *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. *ExplodeAscii* 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; '–' is accepted for negative numbers, blanks are ignored everywhere except 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)

Search ( )
    (AS.p:string, RS, FR, ^.PR, TS, ES, M) =>
       (AS.matchingmachine(p):string×int→int×int, RS, FR, PR, TS, ES, M)
    ? 'search' if p is malformed

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 updatable 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. *Dest-Cons* 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:α_1,..,x_n:α_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.x_1:α_1..x_n:α_n, RS, FR, ^.PR, TS, ES, M) =>
(AS.(|x_1,..,x_n|), RS, FR, PR, TS, ES, M)

Field (i: int≥1≤n)
(AS.(|x_1:α_1,..,x_n:α_n|), RS, FR, ^.PR, TS, ES, M) =>
(AS.x_i, RS, FR, PR, TS, ES, M)

DestRecord (n: int≥0, [m_1: int≥0; .. ; m_p: int≥0])
(AS[n](|x_1:α_1,..,x_p:α_p|), RS, FR, ^.PR, TS, ES, M) =>
(AS[n](|x_1,..,x_p|)[m_1]x_1..[m_p]x_p, 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 length p. As we already mentioned, it is assumed that these situations can never arise at run time because of typechecking at 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 ([c_1: $\alpha_1 \to \beta$, .. , c_p: $\alpha_p \to \beta$])
(AS.[|a_i=x:α_i|], RS, FR, ^.PR, TS, ES, M) =>
(AS.x, RS, FR, c_i, TS, ES, M) (with $1 \le i \le p$)

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,[$l_1$,..,$l_n$]), RS, FR, PR, TS, ES, M[nth(1,e)/$l_1$]..[nth(n,e)/$l_n$])
    where nth(i,e) is the i-th element of e,
    n is the length of e, and $l_i$ are new addresses

Tabulate ( )
    (AS.f:int→α.lb:int.size:int, RS, FR, ^.PR, TS, ES, M) =>
        (AS.array(lb,size,[$l_1$,..,$l_n$]), RS, FR, PR, TS, ES,
            M[f(lb)/$l_1$]..[f(lb+size–1)/$l_n$])
    ? 'tabulate' if size<0
    where $l_i$ are new addresses

LowerBound ( )
    (AS.array(lb,size,[$l_1$,..,$l_n$]), RS, FR, ^.PR TS, ES, M) =>
        (AS.lb, RS, FR, PR, TS, ES, M)

Size ( )
    (AS.array(lb,size,[$l_1$,..,$l_n$]), RS, FR, ^.PR, TS, ES, M) =>
        (AS.size, RS, FR, PR, TS, ES, M)

Sub ( )
    (AS.array(lb,size,[$l_1$,..,$l_n$]).i:int, RS, FR, ^.PR, TS, ES, M) =>
        (AS.M($l_{i-lb+1}$), RS, FR, PR, TS, ES, M)
    ? 'sub' if i<lb or i≥lb+size

Update ( )
    (AS.array(lb,size,[$l_1$,..,$l_n$]).i:int.x:α, RS, FR, ^.PR, TS, ES, M) =>
        (AS.triv, RS, FR, PR, TS, ES, M[x/$l_{i-lb+1}$])
    ? 'update' if i<lb or i≥lb+size

ArrayToList ( )
    (AS.array(lb,size,[$l_1$,..,$l_n$]), RS, FR, ^.PR, TS, ES, M) =>
        (AS.cons(M($l_1$),..cons(M($l_n$),nil)..), RS, FR, PR, TS, ES, M)
```

The operation 'array' used above is the basic allocator of array objects, and 'cons' is the list constructor.

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).

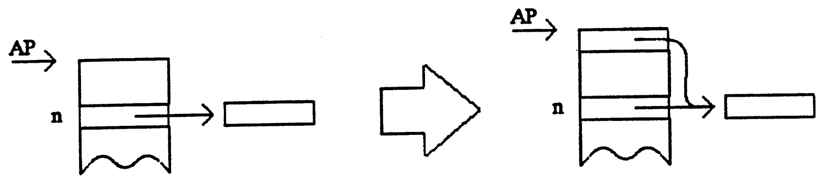
GetLocal (n: int≥0)
    (AS[n]x:α, RS, FR, ^.PR, TS, ES, M) =>
        (AS[n]x.x, RS, FR, PR, TS, ES, M)

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)

$\text{Permute}([p_0: \text{int}{\geq}0{\leq}n{-}1;..;p_{n-1}: \text{int}{\geq}0{\leq}n{-}1])$
$(AS.x_{n-1}{:}\alpha_{n-1}..x_0{:}\alpha_0, RS, FR, \wedge.PR, TS, ES, M) \Rightarrow$
$(AS.x_{p_{n-1}}..x_{p_0}, RS, FR, PR, TS, ES, M)$

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)

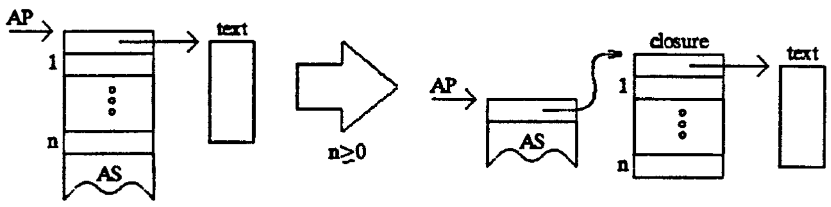
(AS.x_n:α_n..x_1:α_1.t:text[m]closure(triv,triv_1,..,triv_n),
RS, FR, ^.PR, TS, ES, M) =>
(AS[m–n–1]closure(t,x_1,..,x_n), RS, FR, PR, TS, ES, M)

GetGlobal (n: int≥0≤p)

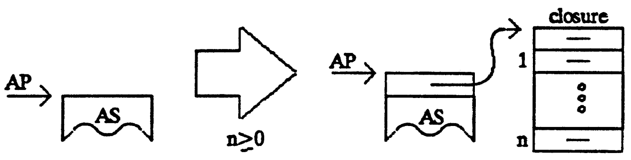
(AS, RS, closure(t:text,x_0:α_0,..,x_p:α_p), ^.PR, TS, ES, M) =>
(AS.x_n, RS, closure(t,x_0,..,x_p), PR, TS, ES, M)

GetLiteral (n: int≥0≤p)

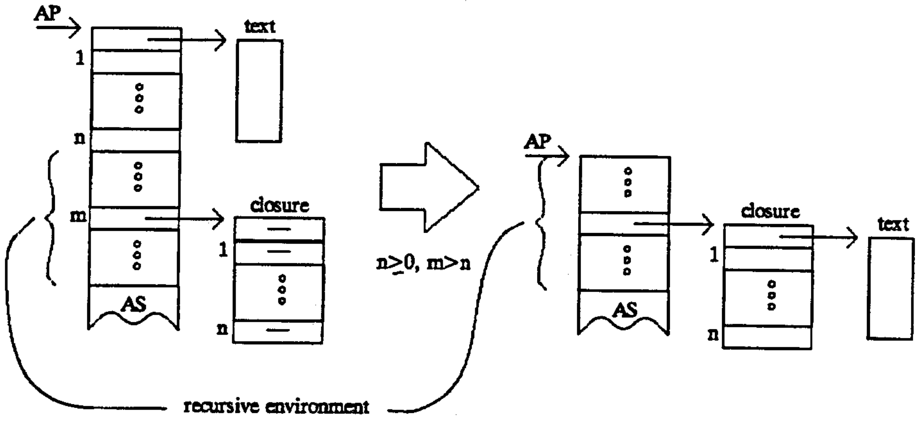
(AS, RS, closure(text(c:code,x_0:α_0,..,x_p:α_p),..), ^.PR, TS, ES, M) =>
(AS.x_n, RS, closure(text(c,x_0,..,x_p),..), PR, TS, ES, M)



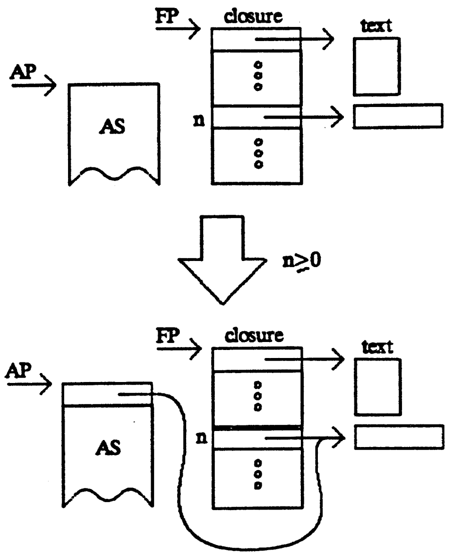
Closure



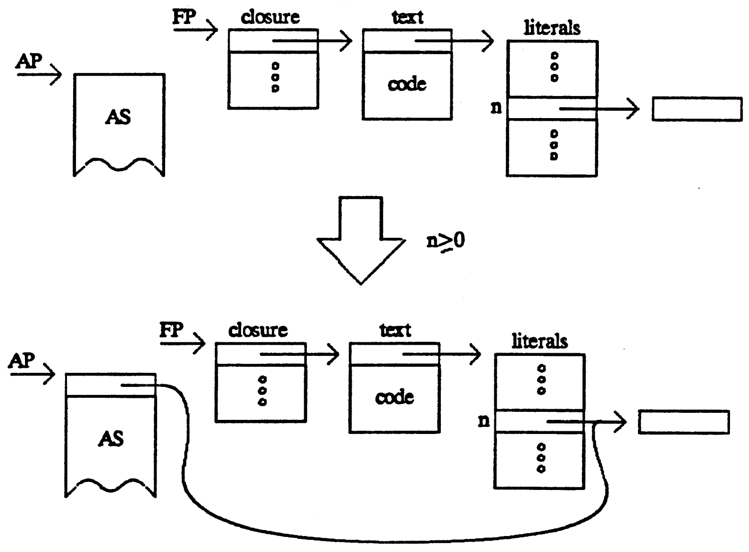
DumClosure



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 canceled 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 penalty. 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)

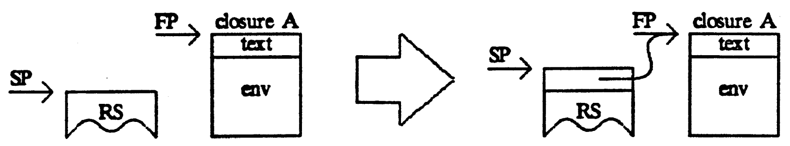
(AS.x_{n+m-1}:α_{n+m-1}..x_0:α_0, RS.c:code, FR, ^.PR, TS, ES, M) =>

(AS.x_{m-1}..x_0, RS, FR, c, TS, ES, M)

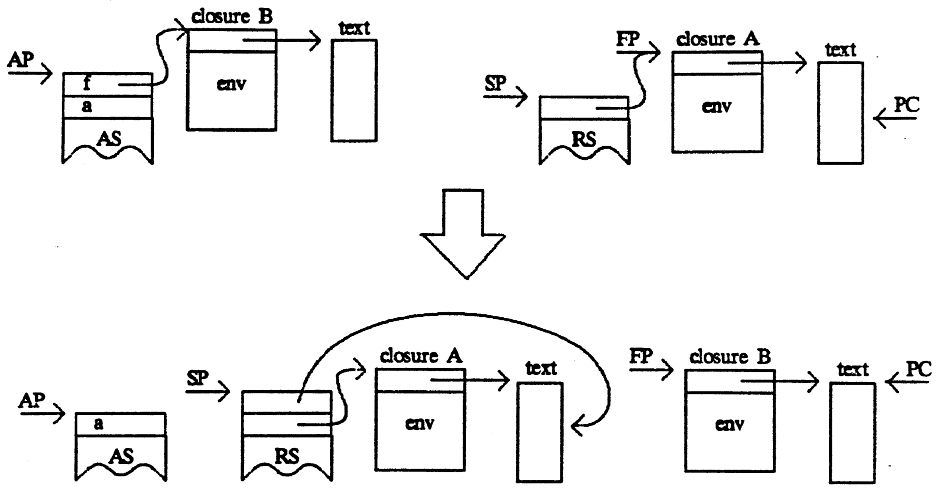
TailApply (n: int≥0, m: int≥0)

(AS.x_{n+m-1}:α_{n+m-1}..x_0:α_0.closure(text(c:code,..),..):$\alpha\to\beta$,
RS, FR, ^.PR, TS, ES, M) =>

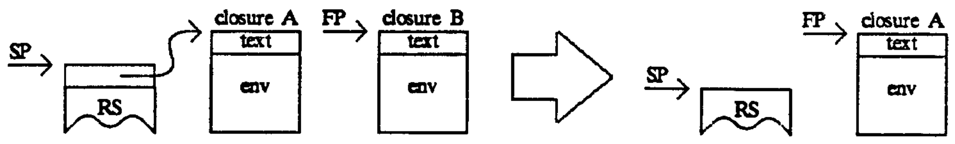
(AS.x_{m-1}..x_0, RS, closure(text(c,..),..), c, TS, ES, M)



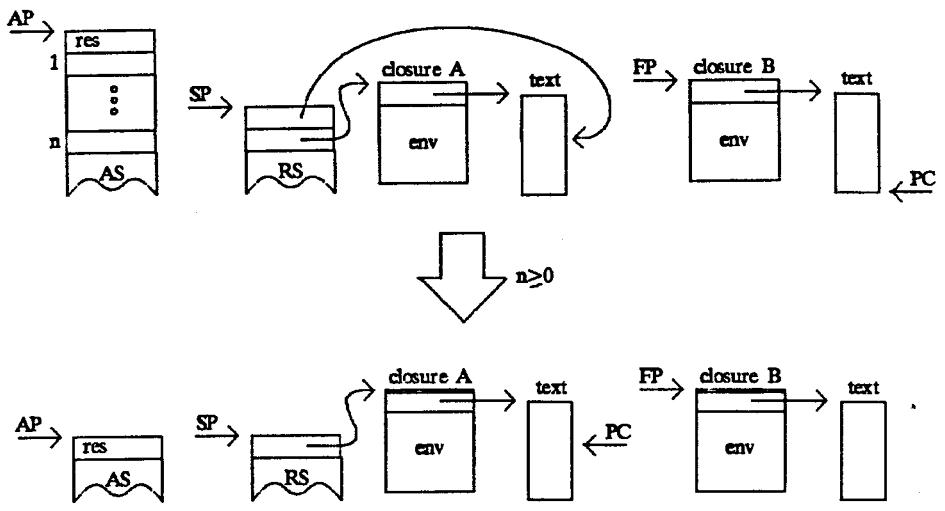
SaveFrame



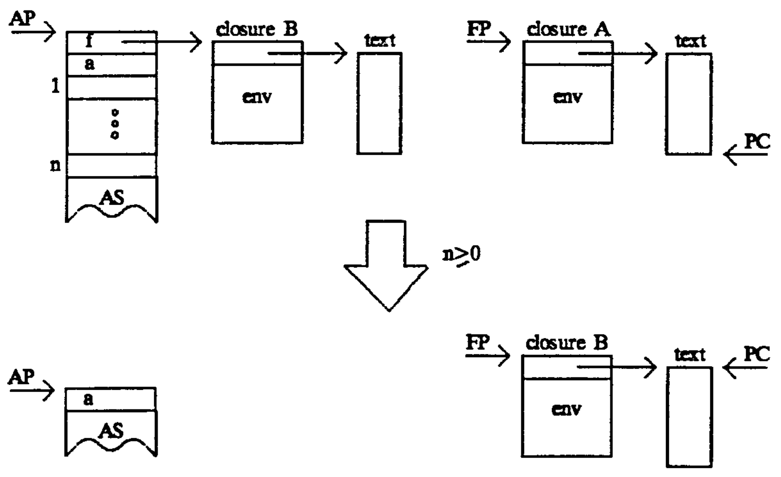
ApplFrame



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)

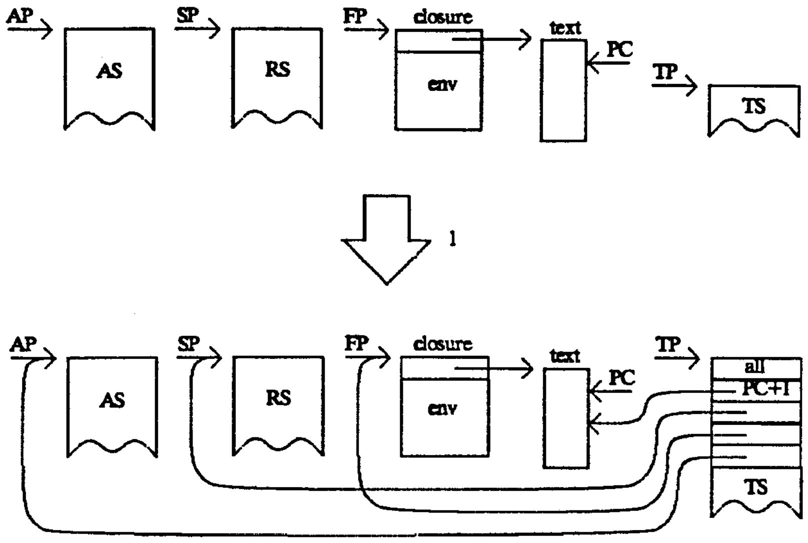
(AS, RS, FR, ^.PR, TS.(all,c,RS′,FR′,AS′), ES, M) =>
(AS, RS, FR, c, TS, ES, M)

(AS, RS, FR, ^.PR, TS.(only(sl),c,RS′,FR′,AS′), ES, M) =>
(AS, RS, FR, c, TS, ES, M)

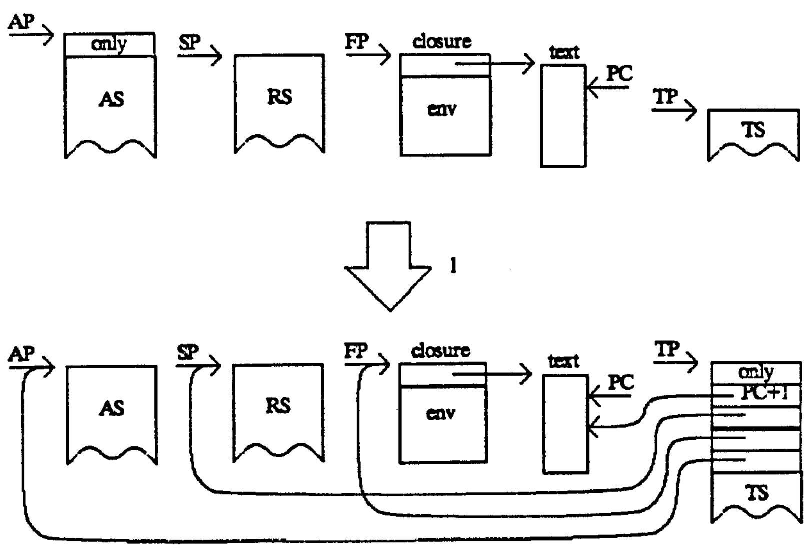
FailWith

(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.

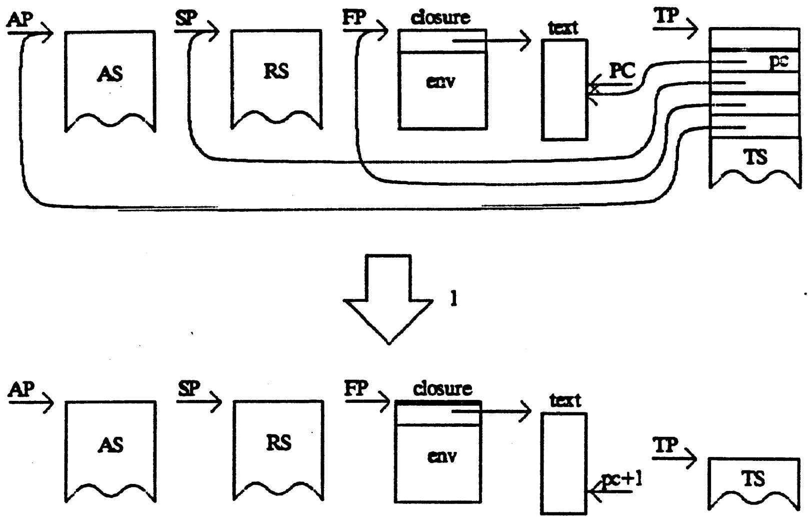
(AS.s:string, RS, FR, ^.PR, TS.(x,PR′,RS′,FR′,AS′).., ES, M) =>
(AS.printfailure(s), RS, FR, <>, <>, ES, M)
if there is no trap block satisfying the above condition.



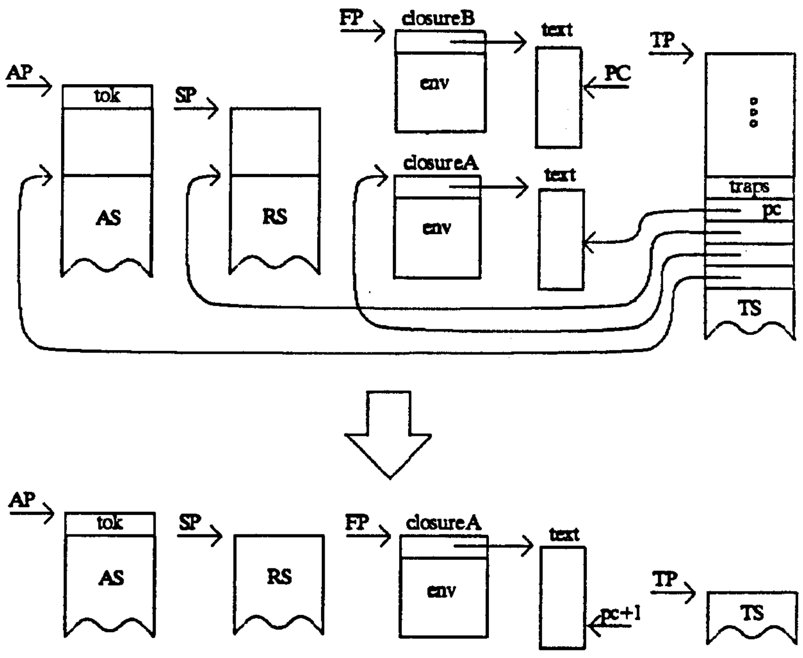
Trap



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 string pattern 'p' and returns a scanning machine 'm' for that pattern. Every time 'm' is applied to a stream, it will read until after the longest leftmost string matching that pattern, if any, and return it. If no match is found the stream is not affected. InString fails with string 'instring' if the pattern is malformed; the machine 'm' fails with string 'scan' when no match is found, or on any I/O error. The syntax and semantics of patterns is described in section "String Patterns". *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.p:string, RS, FR, ^.PR, TS, ES, M) =>
        (AS.scanningmachine(p):stream→string, RS, FR, PR, TS, ES, M)
    ? 'instring' if p is malformed

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 ($n \geq 1$).

? '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 Export and Import

Export takes a stream and an arbitrary Fam object, and saves the objects in the stream, so that it can be later reloaded by the Import operation. Data sharing and loops are preserved. The notation used to store an object in a stream is called MX (for module exchange): it is a printable Ascii representation of Fam objects, having the following syntax (see section "Concrete Syntax" for a description of the metasyntactic notation):

MX (Module Exchange) Syntax

```
mx ::=
    '[' {shared} main ']'

main ::= term

shared :: = term

hexbyte ::= hexdigit hexdigit

hexdigit ::= '0' | .. | '9' | 'A' | .. | 'F'

hex ::= hexbyte hex          (hexbytes may be separated by blanks or newlines)

size ::= int

code ::= hex

chars ::= hex

term ::=
    int                                  small integer
    | '^' int                            indirection (shared or circular data)
    | '[u' size hex ']'                  unbounded precision integer
    | '[s' size chars ']'                string
    | '[p' term term ']'                 pair
    | '[l' {term} ']'                    list
    | '[i' term ']'                      reference
    | '[r' size {term}1 ']'              record
    | '[v' int term ']'                  variant
    | '[a' size term term {term} ']'     array (size, lowerbound, size, items)
    | '[c' size text {term} ']'          closure
    | text                               text

text ::=
    '[t' size '[' size {term} ']' code ']'
```

Note: the double size information in arrays (which seems redundant) is need by 'Import', as the array size field could be a shared bigint and unavailable when needed.

A top-level MX definition ('mx' above) is a list '[t_1 .. t_n t]' of terms (with n≥0): the main object is 't', and 't_1' .. 't_n' are auxiliary definition used to encode sharing and loops: whenever a term of the form '^i' (1≤i≤n) is found in t_1 .. t_n,t, it is interpreted as referring to t_i.

The *Export* operation can be used to implement separate compilation, when the exported objects are closures.

Import takes a stream which is assumed to contain an MX definition and converts it into a Fam object. It is possible to use Import to load compiled programs produced by foreign systems, if the code is fully relocatable and it is converted to MX text or closure notation.

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

11 Eval

Eval takes a special Fam object, called here a *FamProg*, as argument, and interprets it as defining a 'source' Fam program. The FamProg is assembled and executed, and its result is returned as the result of the Eval operation.

A FamProg is a list of statements. Every statement is a pair consisting of a label (an integer) and an operation. Labels are local to (FamProg's denoting) text objects; label 0 means no label. An operation is a variant object, having an OpCode as tag and an operand as contents. The operand can have different structures depending on the OpCode. The OpCode is a numerical encoding of the Fam operations; in addition there is a special pseudo-OpCode, called *Thread*, whose operand is a FamProg.

The Thread OpCode is meant to facilitate translation from high-level languages into FamProg's, by providing a simple way of concatenating two or more FamProg structures. Two FamProg's P1 and P2 can be concatenated by generating the list of statements '...; Thread P1; Thread P2; ...', which is a FamProg. Because of the Thread pseudo-operation, a FamProg for a single text object may be a tree of statements, as opposed to a simple linear list. The correct sequence of statements is given by traversing the tree from left to right in preorder.

The precise structure of a FamProg is defined in section "Abstract Syntax". Eval may fail with string 'eval' if its argument is not a legal FamProg.

```
Eval ( )
    (AS.p:FamProg, RS, FR, ^.PR, TS, ES, M) =>
        (AS, RS, FR, assemble(p).PR, TS, ES, M)
    ? 'eval' if p is not a legal FamProg.
```

where 'assemble' is a straightforward translation form FamProg structures to abstract machine programs.

Note1: FamProg's do not have to be (sources for) texts or closures; they may, for example, simply build a pair by 'Eval[Int 3; Int 5; Pair] = (3,5)'.

Note2: FamProg's executed by Eval should not contain Start and Stop operations.

12 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.

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)
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.
```

13 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 time, recently allocated data points 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 as 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 the latter 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 it 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 compaction (to ensure this migration of stable data we have to alternately 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.

14 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(arg_1,..,arg_n)|] =>
    [|arg_1|] .. [|arg_n|] [|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 .. z .. 's' .. |] =>
    Get(y) Get(z)
    GetLiteral(text([| .. x .. y .. z .. '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(1,1))) RecClosure(1,1)
    GetLocal(0) GetLiteral(text([| .. f .. |]Return(1,1))) 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:
```

15 String Patterns

String patterns are used in the *Search* and *InString* operations; they provide a regular expression pattern matching facility for character strings. A string pattern is itself a string, which is interpreted as regular expression definition (the metasyntactical notation is defined in section "Concrete Syntax"):

```
special ::=  "(" | ")" | "*" | "+" | "?" | "|" | "." | "~" | "#" | "[" | "]" | "/"

basic ::= any Ascii character which is not a special

self ::= basic | "/" special

exp ::=
    self                  match itself ("/" is the escape for specials).
    | "."                 match any character except newline (Ascii 10).
    | "~"                 match beginning of string.
    | "#"                 match end string.
    | "["{self}1"]"       match any of the characters in brackets.
    | exp "*"             match any number of exp's, including zero.
    | exp "+"             match any number of exp's greater than zero.
```

| exp "?"	match exp optionally.
| exp exp	match the concatenation of two exp's.
| exp "|" exp	match one of two exp's.
| "(" exp ")"	match exp.

In the Search operation, '~' matches the beginning of the search string: it is not associated to any substring, but the matching substring, if any, will be an initial segment of the search string. In the InString operation, '~' matches the current beginning of stream; the result of InString will be an initial segment of the stream (this prevents running down the stream in search of a match).

In the Search operation, '#' matches the end of the search string: it is not associated to any substring. In the InString operation, '#' matches the current end of stream; if '#' is not used, InString may hung at the end of stream waiting for more characters to be written in it.

16 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'
| 'Search'
| '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' Label
| 'TrapList' Label
| 'UnTrap' Label
| 'FailWith'
| 'PutStream'
| 'GetStream'
| 'ListStreams'
| 'EmptyStream'
| 'CopyStream'
| 'EndStream'
| 'InChar'
| 'InString'
| 'InInt'
| 'OutChar'
| 'OutString'
| 'OutInt'
| 'Eval'
| '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.

17 Abstract Syntax

This section contains a description of Fam programs in terms of Fam data structures. The main use of this representation is in the Eval operation (see section "Eval").

In what follows, 'list' is the list type, '#' is the pair type and '[| ... |]' is the variant type; when no type is associated to a variant label, triv is intended. The operations are listed in alphabetical order. The numerical opcodes of operations (used by Eval) is given by the position in this list, with 'OpAnd' being '1'. Comments are enclosed in '{ }'.

```
FamProg = FamStatement list

FamStatement = {Label} int # FamOperation

FamOperation =
    [|
    OpAnd;
    OpApplFrame;
    OpArray;
    OpArrayToList;
    OpAs:          {CaseNumber} int;
    OpAssign;
    OpAt;
    OpBoolEq;
    OpCase:        {CaseLabels} int list;
    OpClosure:     {Size} int;
    OpCollect;
    OpCons;
    OpCopyStream;
    OpDefine;
    OpDeflate:     {Size} int # {Displ} int;
    OpDestCons:    {ListDispl} int # {HeadDispl} int # {TailDispl} int;
    OpDestNil:     {ListDispl} int;
    OpDestPair:    {PairDispl} int # {LeftDispl} int # {RightDispl} int;
    OpDestRecord:  {RecordDispl} int # {FieldDispls} int list;
    OpDestRef:     {RefDispl} int # {AtDispl} int;
    OpDestVariant: {CaseNumber} int # {VariantDispl} int # {AsDispl} int;
    OpDiff;
    OpDivide;
    OpDumClosure: {Size} int;
    OpDump;
```

```
OpEmptyStream;
OpEval;
OpExplode;
OpExplodeAscii;
OpExport;
OpFailWith;
OpFalse;
OpFalseJump:    {TargetLabel} int;
OpField:        {FieldNumber} int;
OpFunComp;
OpFunId;
OpGetLocal:     {Displ} int;
OpGetGlobal:    {Displ} int;
OpGetStream;
OpGreater;
OpGreaterEq;
OpHead;
OpImport;
OpImplode;
OpImplodeAscii;
OpInChar;
OpInInt;
OpInString;
OpInflate:      {Size} int # {Displ} int;
OpInt:          {Integer} int;
OpIntEq;
OpIntToString;
OpIs:           {CaseNumber} int;
OpJump:         {TargetLabel} int;
OpLeft;
OpLength;
OpLess;
OpLessEq;
OpListStreams;
OpLowerBound;
OpMinus;
OpModule;
OpNewStream;
OpNil;
OpNot;
```

```
OpNull;
OpOr;
OpOutChar;
OpOutInt;
OpOutString;
OpPair;
OpPermute:     {Permutation} int list;
OpPlus;
OpPutStream;
OpRecClosure:  {Size} int # {Displ} int;
OpRecord:      {Size} int;
OpRef;
OpRestFrame;
OpReturn:      {DeflateSize} int # {DeflateDispl} int;
OpRight;
OpSame;
OpSaveFrame;
OpSearch;
OpSize;
OpSkip;
OpStandAlone;
OpStart;
OpStop;
OpString:      {String} token;
OpStringEq;
OpStringToInt;
OpSub;
OpSubString;
OpTabulate;
OpTail;
OpTailApply:   {DeflateSize} int # {DeflateDispl} int;
OpText:        {Text} FamProg;
OpTimes;
OpTrap:        {TargetLabel} int;
OpTrapList:    {TargetLabel} int;
OpTriv;
OpTrue;
OpTrueJump:    {TargetLabel} int;
OpUnTrap:      {TargetLabel} int;
OpUpdate;
```

```
    OpVariant:     {CaseNumber} int;
    Thread:        FamProg
|]
```

Note: OpText and OpString denote OpGetLiteral operations.

18 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

```
+--+--+--+--+--+--+--+--+ ... +--+--+--+--+
|     n     |  Chunk 1  |     |  Chunk n  |    (n≥1)
+--+--+--+--+--+--+--+--+ ... +--+--+--+--+
^
```

Pair

```
+--+--+--+--+--+--+--+--+
|    Fst    |    Snd    |
+--+--+--+--+--+--+--+--+
^
```

List

```
0        (emptylist)       (unboxed)

+--+--+--+--+--+--+--+--+
|    Hd     |    Tl     |  Tl may only be emptylist
+--+--+--+--+--+--+--+--+  or point to a list
^
```

Record

```
0        (nullrecord)                   (unboxed)

+--+--+--+--+--+--+ ... +--+--+--+--+
|  n  |  Field 1  |     | Field n   |   (n>0)
+--+--+--+--+--+--+ ... +--+--+--+--+
      ^
```

Variant

```
+--+--+--+--+--+--+
|    As     | Is  |
+--+--+--+--+--+--+
^
```

Reference

```
+--+--+--+--+
|    At     |
+--+--+--+--+
^
```

String

```
+--+--+--+--+--+ ... +--+
|     n     |C1|     |Cn|   (n≥0)
+--+--+--+--+--+ ... +--+
^
```

Array

```
+--+--+--+--+--+--+--+--+--+--+--+--+ ... +--+--+--+--+
|LowerBound |     n     | Item 1    |     | Item n    |  (n≥0)
+--+--+--+--+--+--+--+--+--+--+--+--+ ... +--+--+--+--+
^
```

Text

```
+--+--+--+--+--+--+--+--+ ... +--+   (n≥1)
|  n  | Literals  |C1|     |Cn|   (^) when pointed from a Closure
+--+--+--+--+--+--+--+--+ ... +--+   Literals may be Nil or point
      ^          (^)                 to Literals
```

Literals

```
+--+--+--+--+--+--+--+ ... +--+--+--+--+   (n≥1)
|  n  | Literal 1 |     | Literal n |   Literal may be a Text,
+--+--+--+--+--+--+--+ ... +--+--+--+--+   a String or a BigInteger
      ^
```

Closure

```
+--+--+--+--+--+--+--+--+--+--+--+ ... +--+--+--+--+
|  n  |   Text    |  Global 1 |     |  Global n |   (n≥0)
+--+--+--+--+--+--+--+--+--+--+--+ ... +--+--+--+--+   Text points
      ^                                             to a Text
```

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.