LC/12-80

A Module Exchange Format

In this note I'll try to define some primitives needed to implement an Environment facility for ML. No (conscious) assumption is made on the character of these environments, apart the fact that they will be stored into files and that there should be no limit whatsoever to their total or individual size.

The key idea is that it should be possible to make a clever dump of an ML run-times memory, in such a way that:

- Dumps can be read back without recompilation.
- Dumps can be merged and manipulated.
- Non ML-generated routines can be interfaced

Security issues, like protecting dumps from user manipulation or not trusting external routines, are not considered here. Dumps should be specified in such a way that:

- They are compact
- They preserve shared and circular structures
- They are address and address-length independent.

MEX

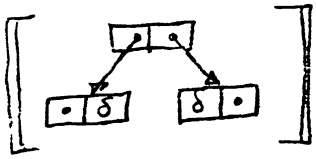
MEX is the language for Module EXchange; it is a textual language based on Lisp-like expressions.

The main data-type-independet feature of MEX is the join construct, which is used to represent sharing and circularity of expressions:

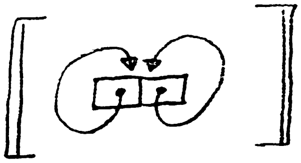
JoinOccurrence ::= "[j" integer "]"

JoinDeclaration(Exp1, Exp2) ::= "[J" integer Exp1 Exp2 "]"

For example consider the following data structures and their MEX translations (where $[\![\ ]\!]$ is the translation function; $[\![\bullet]\!] = e$, and $[\![\boxed{\alpha}\boxed{\beta}]\!] = [p\,[\![\alpha]\!][\![\beta]\!]]$)

Sharing: 

= [J 1 $[\![\delta]\!]$ [p [p e [j 1]][p [j 1] e]]]

Circularity: 
= [J 1 [p [j 1][j 1]] [j 1]]

Of course a translation from data structures to MEX requires two passes.

```
Value ::= JoinOccurrence |
          JoinDeclaration(Value, Value) |
          "e" |                                   %Empty%
          "t" |                                   %True%
          "f" |                                   %False%
          "[i" integer "]"                        %Integer%
          "[n" number "]"                         %Number%
          "[t" token "]"                          %Token%
          "[p" Value Value "]"                    %Pair%
          "[l" {Value} "]"                        %List%
          "[u" ("l" | "r") Value "]"              %Injection%
          "[r" Value "]"                          %Reference%
          "[c" {Value}1 "]"                       %Closure% (1)
          "[x[" {Value} "]" {Hex}1 "]"            %Text% (2)
```

[1] The first "Value" is the text, and the others are the values of the global variables in the text.

[2] The "Value"s are the subtexts, and "Hex" is a hexadecimal number.

```
Type ::=  JoinOccurrence |
          JoinDeclaration(Type, Type) |
          TypeVar |
          "D" |                                   % Dot %
          "B" |                                   % Bool %
          "I" |                                   % Integer %
          "N" |                                   % Number %
          "T" |                                   % Token %
           "[P" Type Type "]" |                   % Pair %
           "[U" Type Type "]" |                   % Union %
           "[L" Type "]" |                        % List %
           "[F" Type Type "]" |                   % Function %
            "[R" Type "]" |                       % Reference %
            "[D"  ...   Type "]" |                % Defined Type %
            "[A"  ...   Type "]"                  % Abstract Type %
```

```
Value Env      ::= "[V" {ValueBinding} "]"

ValueBinding ::= Join Occurrence |
                 Join Declaration(Atom, {ValueBinding}) |
                 Join Declaration(Value, {ValueBinding}) |
                 "[" Atom Value "]"


Type Env       ::= "[T" {TypeBinding} "]"

TypeBinding  ::= Join Occurance |
                 Join Declaration(Atom, {Type Binding}) |
                 Join Declaration(Type, {Type Binding}) |
                 "[" Atom Type "]"


Module ::= {{Type Env}{Value Env}}
```

Note: It is not clear to me at this point how details of atoms and abstract and defined types should be filled-in -