A Module Exchange Format

In this note I'll try to define some practices 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-time's memory, in such a way that:

- Dumps can be read back without recirculation
- 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-independent feature of MEX is the `join` construct, which is used to represent sharing and circularity of expressions:

\[ \text{JoinOccurrence} ::= \left[ j \right] \text{ integer } \]

\[ \text{JoinDeclaration}(\text{Exp1, Exp2}) ::= \left[ j \right] \text{ integer } \text{Exp1 Exp2 } \]

For example consider the following data structures and their MEX translations (where \( i \) \( j \) is the translation function; \( \bullet \) = e, and \( i i a j b ) = [p[[x]][y]] \))

\[ \text{Sharing:} \quad \left[ \begin{array}{c}
\circ \vspace{0.5cm}
\end{array} \right] = \left[ j \right] \left( j \right) [p\left[[x][y]]\left[[p\left[[x][y]\right]]e\right]\right] \]

\[ \text{Circularity:} \quad \left[ \begin{array}{c}
\circ \vspace{0.5cm}
\end{array} \right] = \left[ j \right] \left( j \right) \left[[p\left[[x][y]\right]]\left[[p\left[[x][y]\right]]e\right]\right] \]

Of course a translation from data structures to MEX requires two passes...
Value ::= \text{Join Occurrence} / \text{Join Declaration} (\text{Value}, \text{Value}) /

"e" / % Empty %
"t" / % True %
"f" / % False %
"E" integer "J" / % Integer %
"N" number "J" / % Number %
"T" token "J" / % Token %
"\{p\} Value Value "J" / % Pair %
"\{p\} \{\text{Value} \} "J" / % List %
"[v" ("e" | "t") Value "]" / % Injection %
"[v" Value "]" / % Reference %
"[v" \{\text{Value} \} | "J" / % Closure %
"[x\{\"\{\text{Value}\}"\}"\{\text{Hex}\} | "J" / % Text %

(1) The first "Value" is the text, and the others are the values of the global variables in the text.
(2) The "Values" are the subtexts, and "Hex" is a hexadecimal number.
Type ::= Join Occurrences

Join Declaration (Type, Type)

Type Var |
"D" |
"B" |
"I" |
"N" |
"T" |
"[P" Type Type "P"]" |
"[O" Type Type "O"]" |
"[L" Type "L"]" |
"[F" Type Type "F"]" |
"[R" Type "R"]" |
"[D" ... Type "D"]" |
"[A" ... Type "A"]"

% Dot %
% Bool %
% Int %
% Number %
% Token %
% Pair %
% Union %
% List %
% Function %
% Reference %
% Definied Type %
% Abstract Type %
Value Env ::= "[V] \{Value Binding\} [J]"

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

Type Env ::= "[T] \{Type Binding\} [J]"

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

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