COMPARISON TO PROLOG

William Bricken
January 1986

I consider parens notation as a sort of machine language for deduction. Consequently any system with an interface needs to retranslate into a higher level language for comprehensibility. Simplification of Prolog takes the rough form of translating Prolog code into Losp, reducing, and then re-transcribing back into Prolog. Logically redundant code gets boiled out, and (using extract) the most global location for variables is identified. Logical variables are treated as free, so they pass into and out of Losp unchanged. We're working with a minimal form of Prolog; most of our experience is with canonical forms of LISP, and with First Order Logic specifications. In this regard, Prolog is a subset of FOL, so it doesn't get much attention. The goal is full on programming in logic, for which Losp is the compiler.

Example of Membership

The logical definition of member is:

```
member{a, S} =

        (a = (first S)) or (a = (first (first S))) or ... or (a = {})
```

The last clause is the halting condition that yields false when S is empty.

I'll use {} instead of () for Prolog arguments, and numerals for logical variables standardized apart:

```
member{1, [1|_]}.
member{2, [_|3]} :- member{2, 3}.

?- member{b, [a, b, c]}.
```

Transcribing the logic of Prolog (in its inference engine):

```
(mem{1, [1|_]}) ( (mem{2, 3}) mem{2, [_|3]} ) mem{b, [a, b, c]}
```

The *mem{2,3}* term is a formal condition for Prolog matching, really needed by the inference engine for sequencing of operations, and not necessary as part of the representation of the problem. The terms in the parens (i.e. the rules) have unlimited number of copies. From this point on, all processing is by unify-and-extract, no "natural deduction" is necessary.

Unify-and-extract

```
      mem{2, [_|3]}
      2 => b
      3 => [b, c]

   ==>  (mem{1, [1|_]}) ( (mem{b, [b, c]) ) mem{b, [a, b, c]}

   ==>  (mem{1, [1|_]})    mem{b, [b, c]    mem{b, [a, b, c]}

   ==>  (mem{1, [1|_]}) ( (mem{4,5}) mem{4, [_|5]} ) mem{b, [b, c]}
```

Unify-and-extract

```
      mem{1, [1|_]}
      1 => b

   ==>  (             ) ( (mem{4,5}) mem{4, [_|5]} ) mem{b, [b, c]}

   ==>  ( )
```

I've skipped over various pointer maintenance operations that make unification choices simpler, and keep track of results. The only necessary transformations we do are the the Losp erasure rules, although Transposition does provide a variety of final forms.

Actually the *member* function is expressable directly in Losp:

```
      member{1, [2, 3, 4]}  is

           ( ((1) 2 3 4) (1 (2) (3) (4)) )

      member{b, [a, b, c]}  is

           ( ((b) a b c) (b (a) (b) (c)) )

      ==>  ( (( ) a b c) (b (a) ( ) (c)) )
      ==>  (                             )  ==>  true
```

This example gives a flavor of how extract-and-unify works. The point is that it is a sufficient mechanism for deduction when combined with Transposition.