

VLisp: A Formally Verified Lisp

or

Explorations in the Formal Verification of McCarthy's Equations

K. Isom

October 24, 2023

1 Background

Alan Kay once described a certain half-page of code in the Lisp 1.5 manual¹ as

These were “Maxwell’s Equations of Software!” This is the whole world of programming in a few lines that I can put my hand over².

(These equations are described in Figure 1.) If we posit that these equations are, in fact, an analogue of Maxwell’s equations that capture a description of “the whole world of programming” in a mathematically-precise language, can a formally-verified implementation be built? What does this implementation tell us about the world of programming?

Some core questions this might immediately invoke are,

1. What is the syntax of a formally-verified Lisp?
2. What are the semantics of a formally-verified Lisp?
3. What guarantees can we make about the behaviour of this Lisp?
4. Working under the assumption that Lisp (in particular, a Lisp derived from the equations in Figure 1) can be considered as a sort of “McCarthy’s Equations of Software” (MEoS), what theorems can be made regarding its behaviour and properties?
5. What are the practical, useful implications of these theorems?

While the end goal is to build a formal model of Lisp from McCarthy’s equations, it will be necessary to implement the first chapter of the Lisp 1.5 manual, “The Lisp Language”.

¹<http://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf>

²<http://queue.acm.org/detail.cfm?id=1039523>

evalquote is defined by using two main functions, called eval and apply. apply handles a function and its arguments, while eval handles forms. Each of these functions also has another argument that is used as an association list for storing the values of bound variables and function names.

```
evalquote[fn;x] = apply[fn;x;NIL]
where
  apply[fn;x;a] =
    [atom[fn] → {eq[fn;CAR] → caar[x];
      eq[fn;CDR] → caddr[x];
      eq[fn;CONS] → cons[car[x];cadr[x]];
      eq[fn;ATOM] → atom[car[x]];
      eq[fn;EQ] → eq[car[x];cadr[x]];
      T → apply[eval[fn;a];x;a]];
    eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
    eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
      caddr[fn]];a]]]
  eval[e;a] = {atom[e] → cdr[assoc[e;a]];
    atom[car[e]] →
      [eq[car[e];QUOTE] → cadr[e];
      eq[car[e];COND] → evcon[cdr[e];a];
      T → apply[car[e];evlis[cdr[e];a];a]];
    T → apply[car[e];evlis[cdr[e];a];a]]
  pairlis and assoc have been previously defined.
  evcon[c;a] = [eval[caar[c];a] → eval[caddr[c];a];
    T → evcon[cdr[c];a]]
  and
  evlis[m;a] = [null[m] → NIL;
    T → cons[eval[car[m];a];evlis[cdr[m];a]]]
```

13

Figure 1: McCarthy's Equations of Software

2 The Lisp Language

Before delving into a software implementation, let us consider Lisp, the language.

McCarthy splits the language into two formats:

- *S-expressions* (symbolic expressions) are the notation of Lisp programs,
- *M-expressions* (meta-expressions) are the notation of the source language that specifies how to process S-expressions.

Immediately we are presented with the question of representation; that is, how to separate the quotidian task of rote parsing of input text from the extraction of the central tents, the core of the language? What does it mean to be an S-expression? An S-expression consists of either

- an atomic symbol
- or composed of a pair of S-expressions

Syntactically, an atomic symbol is a string of characters (starting with a letter); the Lisp 1.5 manual imposes the limitation that the characters composing an atomic symbol be restricted to numbers and capital letters. In this paper, we'll follow this form for atomic symbols (and represent them in fixed width font) to distinguish them from other symbols. An S-expression is either one of these atomic symbols, or composed of the form (sexpr . sexpr).

Examples of S-expressions:

- ATOM
- (A . B)
- (A . (B . C))

3 VLisp: a verified Lisp

This is a Coq source file, documented and annotated using *coqdoc*. It aims to document the semantics and syntax of a formally-verified Lisp, built from McCarthy's Equations of Software.

The Lisp 1.5 manual distinguishes between two types of expressions. *S-expressions* are the symbolic expressions that comprise Lisp programs; *M-expressions* are the *source language which specifies in what way the S-expressions are to be processed*. It is the job of the interpreter to convert S-expressions into M-expressions; this program will attempt to codify a representation of M-expressions.

3.1 Symbolic Expressions

Atoms are the basic building blocks of Lisp; the Lisp 1.5 manual notes the distinction between *atomic symbols*:

Definition: an atomic symbol is a string of no more than thirty numerals and capital letters; the first character must be a letter.

and *atom*, the predicate:

```
atom[EXTRALONGSTRINGOFLETTERS]=T
atom[(U . V)]=F
atom[car[(U . V)]]=T
```

Inductive `atomic_symbol` : Set := .

A cell (as represented by an S-expression) contains of a first and second element; it is essentially a tuple (x, y) where x and y may be either an atomic symbol or a cell.

Inductive `cell` : Set := NIL.

3.2 Elementary Functions

cons is used to build S-expressions from atomic symbols and other S-expressions.

```
cons[A;B]=(A . B)
cons[(A . B);C]=((A . B) . C)
cons[cons[A;B];C]=((A . B) . C)
```

Definition *cons* (*a b* : **cell**) : **cell** := NIL.

car and *cdr* do the opposite of *cons*: they extract elements from an S-expression.

The function *car* has one argument. Its value is the first part of its composite argument. *car* of an atomic symbol is undefined.

Definition *car* (*a* : **cell**) : **cell** := NIL.

The function *cdr* has one argument. Its value is the second part of its composite argument. *cdr* is also undefined if its argument is atomic.

Definition *cdr* (*a* : **cell**) : **cell** := NIL.

Given any S-expression, it is possible to produce any subexpression of it by a suitable comp

```
car[cons[x;y]]=x
cdr[cons[x;y]]=y
```

The following identity is also true for any S-expression *x* such that *x* is composite (non-atomic):

```
cons[car[x];cdr[x]]=x
```

Theorem *car_x_is_x* : $\forall x y : \mathbf{cell}, \text{car} (\text{cons } x y) = x$.
Abort.

Theorem *cdr_x_is_y* : $\forall x y : \mathbf{cell}, \text{cdr} (\text{cons } x y) = y$.
Abort.

Variables are used to give names to S-expressions.

A function whose value is either true or false is called a predicate. In LISP the values true and false are represented by the atomic symbols T and F, respectively. A LISP predicate is therefore a function whose value is either T or F.

The predicate *eq* is a test for equality on atomic symbols. It is undefined for non-atomic arguments.

Inductive **pred** : **atomic_symbol** $\rightarrow$ **bool** $\rightarrow$ Set :=.

Definition *eq* (*x y* : **atomic_symbol**) : **bool** := false.