Metacalculi

Marseille Valen Bouchard Demko

March 8, 2023

Abstract

We present several formal systems based on the lambda calculus which are capable of some form of metaprogramming. Exactly what counds as metaprogramming is left informal, but should include Lisp and Scheme macros, as well as Meta-ML and Template Haskell.

1 Splice-Quote Lambda Calculus

We first recall the applied lambda calculus:

Definition 1. A *applied lambda calculus* ($\lambda\delta$) is a formal system.

1. It is parameterized by
   (a) a countable set of constants $\mathcal{C}$, and
   (b) a countably infinite set of variables $\mathcal{X}$, and
   (c) a partial function $\delta : \mathcal{C} \times \{n : \mathbb{N}\} \times \mathcal{C}^n \to \mathcal{C}$.
2. Its syntax is:

$$\begin{array}{rcl} c & \in & \mathcal{C} \\ x & \in & \mathcal{X} \\ e & ::= & c \\ & | & x \\ & | & \lambda x.\, e \\ & | & f\ e \end{array}$$

3. It is equipped with a reduction relation:

$$\begin{array}{rcll} (\lambda x.\, e)\ e' & \longrightarrow & e[x \mapsto e'] & \\ \lambda x.\, e\ x & \longrightarrow & e & \text{where } x \notin \mathrm{fv}(e) \\ c\ c_1\ \ldots\ c_n & \longrightarrow & \delta(c, c_1, \ldots, c_n) & \end{array}$$

4. The reflexive, transitive, symmetric, compatible closure of the reduction relation is the congruence relation

$$t \cong t'$$

The applied lambda calculus can be extended to form the quote-splice calculus. To do so, we first define s-expressions.

Definition 2. Let $\mathcal{A}$ be a set of *atoms* An *s-expression* is a datum generated by the grammar:

$$\begin{array}{rcl} a & \in & \mathcal{A} \\ s & ::= & a \\ & | & (s_1 \ \ldots \ s_n) \end{array}$$

We then set up appropriate injections between s-expressions and the syntax of our calculus.

Definition 3. There is an injection $\lceil \cdot \rfloor : \text{sexpr} \to \text{expr}$:

1. $\forall a \in \mathcal{A}. \exists c \in \mathcal{C}. \lceil a \rfloor = c$
2. $\forall n \in \mathbb{N}. \exists c \in \mathcal{C}. \lceil (s_1 \ldots s_n) \rfloor = c \ \lceil s_1 \rfloor \ \ldots \ \lceil s_n \rfloor$
3. $\lceil s \rfloor = \lceil s' \rfloor$ only when $s = s'$

Definition 4. Specify a set of representable constants $\mathcal{C}' \subseteq \mathcal{C}$ (e.g. numeric literals). There is an injection $\mathcal{S} : \mathcal{X} \cup \mathcal{C}' \to \text{sexpr}$:

1. $\forall x \in \mathcal{X}. \exists s. \mathcal{S}(x) = s$
2. $\forall x \in \mathcal{C}'. \exists s. \mathcal{S}(c) = s$
3. $\mathcal{S}(p) = \mathcal{S}(q)$ only when $p = q$

We then need to extend the syntax and reduction system to cover splices:

Definition 5. The *Quote-Splice Lambda Calculus* is an extension of $\lambda\delta$ where:

1. we extend the syntax with splices

$$\begin{array}{rcl} e & ::= & \ldots \\ & | & \$e \end{array}$$

2. and add a reduction rule showing that quote and splice are dual introduction/eliminatino forms:

$$\begin{array}{rcll} \$\lceil s \rfloor & \longrightarrow & [\![s]\!] & \\ \lambda x.\, e\ x & \longrightarrow & e & \text{where } x \notin \text{fv}(e) \\ c\ c_1\ \ldots\ c_n & \longrightarrow & \delta(c, c_1, \ldots, c_n) & \end{array}$$

3. where the *parsing* function $[\![\cdot]\!] : \textit{sexpr} \twoheadrightarrow \textit{expr}$. To enable parsing, we will need to have $\{\texttt{lambda}, \texttt{quote}, \texttt{eval}\} \subseteq \mathcal{A}$. The parsing function is defined as the smallest (partial) function including the following:

(a) $[\![\mathcal{S}(c)]\!] = c$

(b) $[\![\mathcal{S}(x)]\!] = x$

(c) $[\![(\texttt{lambda}\ s_x\ s)]\!] = \lambda x.\, [\![s]\!]$ where $s_x = \mathcal{S}(x)$

(d) $[\![(s_1\ \ldots\ s_n)]\!] = [\![s_1]\!]\ \ldots\ [\![s_n]\!]$ where $s_1 \notin \{\texttt{lambda}, \texttt{quote}, \texttt{eval}\}$

(e) $[\![(\texttt{quote}\ s)]\!] = \lceil s \rfloor$

(f) $[\![(\texttt{eval}\ s)]\!] = \$[\![s]\!]$

The parsing function is so-called because it represents the extraction of a term of the calculus from some other data structure. Most commonly, parsing operates over character strings, but in this case, it operates on more structured s-expressions.

As a small example, let's implement a simple `let` syntax. Let $\mathbf{c}_i$ be the constant for constructing a size-n combination.

$$let \triangleq \lambda x, v, e.\, \mathbf{c}_2\ (\mathbf{c}_3\ \texttt{lambda}\ x\ e)\ v$$

To use this definition, it will be convenient to define a little syntactic sugar:

$$\$e(s_1\ \ldots\ s_n) \equiv \$(e\ \lceil s_1 \rfloor\ \ldots\ \lceil s_n \rfloor)$$

Then we can see that:

$$\begin{aligned}
\$let(\texttt{x}\ 2\ (\texttt{add x x})) &= \$((\lambda x, v, e.\, \mathbf{c}_2\ (\mathbf{c}_3\ \texttt{lambda}\ x\ e)\ v)\ \lceil \texttt{x} \rfloor\ \lceil 2 \rfloor\ \lceil (\texttt{add x x}) \rfloor) \\
&= \$\lceil (\texttt{lambda x}\ (\texttt{add x x}))\ 2) \rfloor \\
&= [\![(\texttt{lambda x}\ (\texttt{add x x}))\ 2)]\!] \\
&= (\lambda x.\, \text{add}\ x\ x)\ 2 \\
&= \text{add}\ 2\ 2 \\
&= 4
\end{aligned}$$

This is nearly the core of Lisp. Minor differences are that Lisp uses conses instead of lists, passes all arguments to a fucntion as a list, and pattern-matches the input list agains a simple parameter pattern. More difficult is that Lisp has `gensym`, which can produce a fresh symbol. Finally, and most different, is that Lisp need not explicitly splice: it instead recognizes macros and automatically treats them as a splice with the macro parameters filled in with the other s-exprs in the combination. Exactly how this detection of macros is done is not entirely clear to me yet.