Mulholland reference

Spencer Tipping

April 7, 2012

Contents

Chapter 1

Polymorphic consing

Lisp's cons cells have two variables, their `car` and their `cdr`. Generally this is used to form trees where the `car` is used to indicate the role of the `cdr`; for example, the list `(foo bar bif)` is interpreted as a function or macro call to `foo`. The cons cells themselves are transparent to this process.

Generally this works well. There are a few cases where, I think, it makes less sense, including arithmetic expressions. For this case, I think using the `car` to encode the operation is somewhat strange and indirect; better, in my opinion, is to use a different cons operator in the first place. Structurally, Lisp and Mulholland differ in this respect:

```
;; Lisp: monomorphic cons, polymorphism is implied and encoded
;; in the head of the list
(cons '+ (cons 3 (cons 4 nil)))

;; Mulholland: polymorphic cons, polymorphism is universal and
;; encoded as a variant of the cons itself
(cons-+ 3 4)
```

Another significant difference here is that Lisp uses well-formed lists and Mulholland does not. There's a good reason for this having to do with list homomorphism. In Lisp, macros are invoked by the `car` of a list, meaning that `(foo)` and `((foo))` mean two different things. The only reason these lists differ is that each layer of parentheses creates a nontrivial element consed to a trivial `nil`.

1.1 List homomorphisms in Lisp

There are a couple of things worth noting about Lisp's approach. First, all intent is left-focused; each cons cell's purpose is dictated by its `car`, rarely by its `cdr`. This is baked into the macroexpander, and it is often used as a convention

in Lisp libraries. Second, perhaps more interestingly, list homomorphism and evaluation have these equations, where $M[x]$ represents the macroexpansion transformation and $E[x]$ represents evaluation:

$$\begin{aligned} E[x : y] &= E[x](E'[y]) \\ E'[(x : y) : z] &= E[x : y] : E'[z] \\ M[x : y] &= \begin{cases} x(y) & x \text{ is a defined macro} \\ x : M'[y] & \text{otherwise} \end{cases} \\ M'[(x : y) : z] &= M[x : y] : M'[z] \end{aligned}$$

There are two interesting differences between E and M. First, E evaluates lists from the inside out; by the time the function call happens, all of its arguments have been evaluated. This means that evaluation is (nominally) transparent across function arguments, an interpretation that allows arguments to be forced prior to the invocation. Second, the `car` is recursively evaluated under E but not under M.

Macroexpansion and evaluation are convertible:[1]

$$\begin{aligned} E[\texttt{defmacro} : \mathit{name} : \mathit{formals} : \mathit{body} : \texttt{nil}] &\rightarrow (M[\mathit{name} : \mathit{formals}] = E[\mathit{body}]) \\ E[\texttt{macroexpand} : x : \texttt{nil}] &= M[x] \\ E[\texttt{eval} : x : \texttt{nil}] &= E[M[x]] \end{aligned}$$

The $M \rightarrow E$ relationship in the first equation is crucial; it effectively gives the macro body access to both the $M[]$ and $E[]$ transformations. Put differently, it adds the stronger evaluation homomorphism structure to the replacement term.

1.2 Tree homomorphisms in Mulholland

Mulholland has a few significant differences from Lisp. First, there is no $E[]$ transformation available to code; terms exist only in their quoted forms. These quoted forms are then erased in a structure-preserving way when the program is compiled. Second, Mulholland rewriting definitions are built entirely using piecewise combinations of destructuring binds; this is the only way to access argument structure. Third, tree rewriting is contextualized using marker symbols; this is in contrast to Lisp's global symbol namespace.

[1] I'm being imprecise here. `macroexpand` does not expand sublists; it just expands the toplevel cons cell.

Chapter 2

Equations and normalization

Most equations do not indicate action. Instead, they indicate invariants that influence how Mulholland represents things. This means that many equations don't behave like rewrite rules even if they might look that way. For example:

$$x + y = y + x$$
$$x + (y + z) = x + y + z$$

These are not instructions for Mulholland to rewrite + conses. It's a representation constraint for those conses that causes Mulholland to use a bag rather than a tree. Seen differently, Mulholland is required to choose a representation for which every $x + y$ is semantically equivalent to $y + x$ *as it exists in memory*, and that $x + (y + z)$ is equivalent to $x + y + z$. Adding the further constraint that $x + x = x$ causes Mulholland to use a set.

You can use a set as a map by destructuring against pieces of the elements you store in it:

$$\text{put}(k, v, xs) = (k, v) + xs \triangleright \text{put}(k, v, (k, x) + xs) = (k, v) + xs$$
$$\text{get}(k, xs) = \text{nil} \triangleright \text{get}(k, (k, v) + xs) = v$$

2.1 Representation constraints

There are several heuristics Mulholland uses to choose a representation for a structure:

Associativity Causes Mulholland to find a monolithic structure to contain nested conses. Associativity means that elements are flattened into a uniform structure rather than preserving the hierarchical nature of the original cons tree. Associativity alone causes Mulholland to use a linked list.

Commutativity Enables Mulholland to use structural aspects of the values within the structure as indexes. Commutativity with associativity means that elements may be accessed independently of other elements within a cons

structure. Commutativity without associativity does not provide many useful properties that I'm aware of.

Idempotence Enables Mulholland to use idempotent indexing strategies for elements. Generally this is useful only for commutative and associative data structures, but idempotence also causes Mulholland to not allocate cons cells in certain cases. For instance, if $\wedge$ is defined to be idempotent (which it is), then $x \wedge x$ will not allocate a $\wedge$ cons.

These are implemented in Mulholland itself, and you can add your own heuristics and implementations for application-specific optimization.

2.2 Scoped constraints

Sometimes a constraint is not known when a data structure is first used. For example, consider a situation like this:

$$x + (y + z) = x + y + z$$
$$\mathrm{l} = \mathrm{a} + \mathrm{b} + \mathrm{c} + \mathrm{d}$$
$$v \in xs = \mathrm{in}(xs) \vdash \begin{cases} x + y = y + x \\ \mathrm{in}(xs') = \mathrm{false} \rhd \mathrm{in}(v + xs') = \mathrm{true} \end{cases}$$

At this point, $\mathrm{b} \in \mathrm{l}$ will cause the list a + b + c + d to be folded into a bag (since it is being rewritten under a context that supports commutativity), then the bag will be queried for the term in question. This generalizes gracefully for the "contains-multiple" case, as you could easily pass a cons cell in as v to do a bag-subset query: $\mathrm{d} + \mathrm{b} \in \mathrm{l}$. This subset query is meaningful because the bag is associative.

2.3 Cons cell aliasing

Anytime you have a nonrecursive form that matches against one or more cons cells, a cons cell alias is established. This gives Mulholland the option to avoid allocating anything at all, and simply making a function call. The definition of $\in$ above is one such alias. Generally speaking, anything of the form $x \in y$ would be interpreted as a function call instead of a data structure allocation.

Cons cell aliases can be more complex than individual cells. For example:

$$x \in y + z = x \in y \vee x \in z$$
$$x \in y * z = x \in y \wedge x \in z$$

In this case, the alias covers both the $\in$ and the + or $*$; either form can be eagerly evaluated without allocating real cons cells.

Chapter 3

Metaprogramming

Mulholland is deeply metacircular. Because of representation constraints, Mulholland programs can use equation lists in two different ways. On one hand they can be matched against because they are syntax trees, but they take on additional meaning when used with ⊢. This flexibility comes about primarily because the interpretation of each syntax tree is extrinsic, as it is in Lisp.

3.1 Serialization

Every Mulholland value can be serialized because of its isomorphism to syntax trees. Similarly, any serialized value can be read into a different Mulholland interpreter and its semantics can be replicated by entering a context that contains those values.

3.2 Optimization primitives

At the lowest level, Mulholland contains an operator that converts low-level syntax trees to machine code, writes the machine code to an executable region of memory, and executes that memory. These optimization primitives are provided by the base runtime, and they're the only native functions that must be implemented by an interpreter. Ultimately these primitives are how everything is executed.

The Mulholland interpreter core contains a series of rewriting rules that convert high-level syntax trees, including rewriting rules, into low-level machine code. These rewriting rules are applied to themselves to create the Mulholland runtime environment that executes your program.