

Mulholland reference

Spencer Tipping

April 4, 2012

Contents

1	Polymorphic consing	3
1.1	List homomorphisms in Lisp	3
1.2	Tree homomorphisms in Mulholland	4
2	Contexts	5
2.1	Pattern locality	5
3	Equations and normalization	7
3.1	Representation constraints	7
3.2	Scoped constraints	8
3.3	Cons cell aliasing	8

Introduction

Mulholland is a term-rewriting language that arose from the idea that abstractions should be erased at compile-time as an optimization measure. Unlike a lot of languages that do this, Mulholland enables the compiler to perform representation abstraction; that is, the data structures and function calls modeled in source code may be completely changed in the compiled result.

Caterwaul has the ability to do things like this, but it has other shortcomings:

1. Its broad interoperation with Javascript means that it is difficult to port to non-JS runtimes.
2. It generates unreadable output that makes debugging difficult.
3. Rewriting uses no static analysis and therefore is quite slow.
4. Rewriting is assumed to be tree-homomorphic (as is Lisp rewriting).
5. Its syntax trees are lower-level than is generally useful.

Mulholland is, structurally, an infix Lisp with polymorphic consing. Operators have well-defined precedence based purely on their syntactic characteristics. Contexts are used to encapsulate the rewriting process; this allows you to use destructuring binds to create rewrites. Tree-homomorphism is partial and explicitly specified, not tacitly assumed and universal as it is in Lisp's macro system.

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

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

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, the distributive nature of Mulholland trees is explicitly, not implicitly, indicated. Third, Mulholland rewriting definitions are built entirely using piecewise combinations of destructuring binds; this is the only way to access argument structure.

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

Chapter 2

Contexts

Mulholland uses contexts to dictate how terms should be rewritten. Generally these contexts are stored by name by modifying the toplevel context; this is similar to Lisp’s global macro table. This behavior is produced by a few toplevel rewriting rules:¹

$$\begin{aligned}x \vdash (P_x = y) &= y \\x \vdash ((P_x = y_1) \triangleright (P_x = y_2)) &= y_2\end{aligned}$$

Here, P_x represents any pattern that matches and destructures x . Lisp implements P_x such that $x_1 : y_1$ matches $x_2 : y_2$ iff $x_1 = x_2$. The resulting destructure data is, roughly, an association list of y_1 zipped with y_2 .

2.1 Pattern locality

Lisp patterns have fixed locality; the car of a list is the only match point. Caterwaul uses tree structure patterns with continuous paths of polymorphic conses terminated by either constants or wildcards as leaves. Each of these languages assumes that:

1. Tree semantics are preserved only locally.
2. Semantic transformations that occur nonlocally are not generally structure-preserving with respect to the given rewrite rule.

In many cases these assumptions are unnecessarily conservative. Many of caterwaul’s rewriting rules apply universally throughout the source tree, for example.² In other cases, the source tree can be partitioned into a known set of contexts, each of which supports nonlocal transformation.

¹Parentheses are included here to show the cons structure, but they aren’t necessary in the resulting program.

²This is why caterwaul rmaps its macroexpander.

Perhaps more interestingly, destructuring patterns are ways of encoding the instructions that perform a pattern match. Caterwaul and Lisp are both pragmatic by requiring that any pattern executes in $O(n)$ in its size; this is in contrast to something like regular expressions, which can have linear-time subcomponents. Caterwaul and Lisp also both have the property that trees are represented logically in memory; no representation optimization is performed in most cases.

Chapter 3

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:

$$\begin{aligned}x + y &= y + x \\x + (y + z) &= x + y + z\end{aligned}$$

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:

$$\begin{aligned}\text{put}(k, v, (k, x) + xs) &= (k, v) + xs \\ \text{put}(k, v, xs) &= (k, v) + xs \\ \text{get}(k, (k, v) + xs) &= v\end{aligned}$$

3.1 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.

3.2 Scoped constraints

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

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

At this point, $b \in l$ 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: $d + b \in l$. This subset query is meaningful because the bag is associative.

3.3 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:

$$\begin{aligned} x \in y + z &= x \in y \vee x \in z \\ x \in y * z &= x \in y \wedge x \in z \end{aligned}$$

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