Mulholland reference

Spencer Tipping

April 5, 2012

Contents

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]

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

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 can be 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

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]

$$x \vdash (P_x = y) = y$$
$$x \vdash ((P_x = y_1) \rhd (P_x = y_2)) = y_2$$

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.

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

[2] This 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.

I considered implementing some form of nonlocality natively in Mulholland, but I think it would be the wrong move. There are good reasons to limit the match distance of a given pattern, and metaprogramming can be used to implement specific forms of nonlocality later on.

2.2 Wildcards

Lisp doesn't use wildcards because a symbol's role is inferred from its position. Caterwaul and Mulholland match against arbitrary trees and therefore need some way to indicate whether a given pattern term is a constant or a bind variable. Caterwaul uses the convention that symbols beginning with underscores are interpreted as wildcards when matching trees. Because those symbols are also legal Javascript identifiers, you can use a pattern to match against another pattern:

```
'_foo + _bar'.qs /~match/ '_x + _y * _z'.qs
// {_foo: '_x'.qs, _bar: '_y * _z'.qs}
```

Mulholland does something similar, but the notation is an implementation detail of the reader. The reference implementation uses a colon prefix for constants, e.g. `:x + :y`. Variables are written verbatim. In this PDF, variables are italicized: foo(x, y) contains two variables, x and y, and one constant symbol literal.

It is not possible to use a wildcard to select any polymorphic variant of cons while destructuring its children. That is, you can't construct a tree that has a wildcard in a cons position. Doing this creates some ambiguity about intention, and is almost always an excessively broad way to select things. For instance, the intent behind these equations is unclear:

$$(x + y) \langle cons \rangle z = cons$$
$$x + y + z = x \langle y \rangle z$$

Most of the problems in these examples arises from the fact that cons symbols are always just symbols, whereas wildcards can match entire subtrees. The preferred way to match against polymorphic cons trees is to use metaprogramming to build a list of monomorphic patterns and replacements.

2.3 Marker symbols

Generally Mulholland relies on deliberate namespace collision to allow rewriting rules to match each other. However, sometimes collision needs to be avoided. For instance, when inspecting a tree regardless of its semantics it's often useful to create markers to track state. Here's how that could be done to detect a sequence a : b within a list:

$$\text{afterab}(xs) = \text{nothing}(xs) \vdash \begin{cases} \text{nothing(nil)} & = \text{false} \\ \text{nothing(a} : xs') & = \text{gota}(xs') \\ \text{gota(nil)} & = \text{false} \\ \text{gota(b} : xs') & = xs' \end{cases}$$

Now suppose xs contains the subsequence a : b : nothing(nil). The goal is not to rewrite this element into false, despite the fact that the rewriting rules appear to imply that this should happen.

Put differently, we want to limit the scope of an equivalence to a given set of values. We don't want to say that every occurrence of gota means the same thing; particularly, the word gota within the context of this particular $\vdash$ is not the same as any gota that appears in xs.

The way to do this is to define nothing and gota as being unique. This is done at the equation level because the erasure should happen before any values are substituted into the right-hand side (otherwise the intent could be confused with metaprogramming):

$$\left[\text{afterab}(xs) = \text{nothing}(xs) \vdash \begin{cases} \text{nothing(nil)} & = \text{false} \\ \text{nothing(a} : xs') & = \text{gota}(xs') \\ \text{gota(nil)} & = \text{false} \\ \text{gota(b} : xs') & = xs' \end{cases}\right] \ll \begin{cases} \text{nothing} \\ \text{gota} \end{cases}$$

Using $\ll$ causes the symbols to be erased when the context is compiled. Those symbols will be specific to the $\vdash$ tree and will occur nowhere else in the program.

2.4 Creating a context

Contexts are first-class values created using the $\vdash$ operator. $\vdash$ takes two cons forms; the left-hand side is a tree that should be rewritten, and the right-hand side is a structure consisting of one or more equations separated either by $\wedge$ or by $\triangleright$. $\wedge$ is used to indicate that both equations are simultaneously true; so, for instance, you could use it to bind variables:

$$\text{x} = 1 \wedge \text{y} = 5$$

I should mention at this point that in this PDF, I generally imply a $\wedge$ by stacking equations:

$$\begin{aligned} x &= 1 \\ y &= 5 \end{aligned}$$

$\triangleright$ is used when you want to define multiple cases for the same logical rewrite. For example, it would be misleading and incorrect to define a factorial function this way:

$$\begin{aligned} \text{factorial}(0) &= 1 \\ \text{factorial}(n) &= n * \text{factorial}(n-1) \end{aligned}$$

The problem is that the second equation covers the first case, so both equations aren't always true. The first equation is a special case of the second, so in Mulholland you use a domain merge:

$$\text{factorial}(n) = n * \text{factorial}(n-1) \triangleright \text{factorial}(0) = 1$$

$\triangleright$ means, "match the right-hand side first, then proceeding to the left-hand side if the match failed." Mulholland's standard library defines $\triangleleft$, which does what you'd expect.

2.5 Properties of $\wedge$ and $\triangleright$

Equations are flattened out into a set of conjoined piecewise definitions by the following equivalences:

$$\begin{aligned} a \wedge b &= b \wedge a \\ a \wedge (b \wedge c) &= a \wedge b \wedge c \\ a \wedge a &= a \\ a \triangleright (b \triangleright c) &= a \triangleright b \triangleright c \\ a \triangleright a &= a \end{aligned}$$

Note that = is not associative! This allows you to match against and rewrite equations: $(a = b+1) = (a-1 = b)$. = is also not commutative because it specifies a normalization gradient. The right-hand side of any equation is considered to be the "more normal" side, so in cases when the representation can't capture an equivalence the terms will be rewritten from left to right.

2.6 Toplevel context

The toplevel context contains a few universal rewrite rules such as the ones defined above. Source code can be accessed in a variety of ways, but canonical loading can be done using `require`, which will return a syntax tree that you can later use in rewriting operations. For example:

```
# foo.mh: Define some basic stuff
x = 5,
y = 10

# bar.mh: Load foo.mh and use it
main = print (x + y) /- require 'foo.mh'
```

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:

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

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:

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

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.

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.

3.2 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.

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:

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

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

Chapter 4

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.

4.1 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.

4.2 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.