

Mulholland reference

Spencer Tipping

April 3, 2012

Contents

1	Introduction	2
2	Polymorphic consing	3
2.1	List homomorphisms in Lisp	3
2.2	Tree homomorphism in Mulholland	4
3	Contexts	5

Chapter 1

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 2

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.

2.1 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 .

Macro bodies have a predictable evaluation scheme:

$$E[(\text{defmacro} : name : formals : body : nil)] \Rightarrow (name \rightarrow E[body])$$

This is crucial; it effectively provides *body* with a way to use M .

2.2 Tree homomorphism in Mulholland

Chapter 3

Contexts