Sweet.js - Hygienic Macros for JavaScript

Tim Disney
UC Santa Cruz

Nate Faubion

David Herman
Mozilla

1. Introduction

Sweet.js is a new hygienic macro system for JavaScript.

Macros systems have a long history in the design of extensible programming languages going back at least to lisp as a tool to provide programmers syntactic flexibility.

While powerful macro systems have been used extensively in lisp derived languages there has been considerable less movement for macros systems for languages with an expression based syntax such as JavaScript. This is due to a variety of technical reasons that have held back macro systems in expression based languages which we address in this paper.

Recently the Honu project has shown how to overcome some of the existing challenges in developing a macro system for expression based language. The Honu technique was designed for an idealized JavaScript like language. In this paper we show how to extend the ideas of Honu for full JavaScript and present additional techniques that target expression based languages.

The design of sweet.js attempts to overcome the following technical challenges:

- a correct implementation of `read` that structures the token stream before expansion begins
- parser class annotation (e.g. `:expr`) in patterns to allow macro authors easier declaration of a macro shape
- operator overloading
- infix macros
- the `invoke` primitive to allow custom parser classes and more powerful matching

2. Overview

TODO: syntax, main features etc…

3. Read

The syntax of JavaScript presents a challenge to correctly implement the critical `read` function. This challenge is not present in Honu because their language is an idealized syntax that misses the problematic interaction of delimiters and regular expression literals.

TODO: motivate `read` with examples etc.

3.1 Proof of `read`

We first define a simplified grammar that captures just the essential complexity we want to address, namely the interaction of `/` as a divide punctuator and the regular expression literal `/x/`.

$$\begin{array}{rcl} \mathit{Expr}_e & ::= & x\ \mathit{ExprRest}_{x,e} \\ & | & /x/\ \mathit{ExprRest}_{/x/,e} \\ \\ \mathit{ExprRest}_{e,(e/e')} & ::= & /\ \mathit{Expr}_{e'} \\ \mathit{ExprRest}_{e,e} & ::= & \epsilon \end{array}$$

There are two alphabets that interest us: the standard Tokens and the more structured $\mathit{ReadTree}$ which the `read` function will produce.

$$\begin{array}{rcl} \mathit{Token} & ::= & x \mid / \\ \mathit{ReadTree} & ::= & x \mid / \mid /x/ \end{array}$$

We define Expr_T to be the grammar as defined above over the alphabet *Token* and Expr_R as the grammar as defined above but over the alphabet *ReadTree*. The function *tok* takes words in the *ReadTree* alphabet and converts them into words in the *Token* alphabet by splitting regular expression literals apart.

The following lemma notes that any word that is valid for Expr_R is also valid for Expr_T when the *tok* function is applied.

Lemma 1.

$$w \in \mathit{Expr}_R \Rightarrow \mathit{tok}(w) \in \mathit{Expr}_T$$

We can also go the other direction with the *read* function.

Lemma 2.

$$w \in \mathit{Expr}_T \Rightarrow \mathit{read}(w) \in \mathit{Expr}_R$$

Proof. By induction and some other fun stuff. ☐

Which leads us to what we want to prove:

Theorem 1.

$$w \in \mathit{Expr}_T \Rightarrow \mathit{tok}(\mathit{read}(w)) \in \mathit{Expr}_T$$

Proof. Follows from the Lemmas 1 and 2. ☐

To prove that `read` correctly distinguishes between the divide operator and a regular expression literal we first extend the grammar to carry a prefix.

$$\begin{array}{rcl} \mathit{Expr}_p & ::= & \mathit{Literal} \\ & \mid & \mathit{Expr}_p \ + \ \mathit{Expr}_{+\ p} \\ & \mid & \mathit{Expr}_p \ / \ \mathit{Expr}_{/\ p} \end{array}$$

Then "something something by induction" QED.

4. Enforestation

The core algorithm introduced by Honu is called *enforestation* which is basically responsible for expanding macros and building a partial syntax tree with enough structure to match on parse classes. Sweet.js implements this algorithm mostly as described with some additions to provide infix macros and invoke pattern classes described below.

4.1 Infix Macros

The macros we have described so far must all be prefixed by the macro identifier and syntax after the macro name is matched. This is sufficient for many kinds of macros but some syntax forms require the macro identifier to sit between patterns.

Honu addresses this need in a limited way by providing a way to define new binary and unary operators which during expansion can manipulate their operators. However, those operators must be fully expanded and must match as an expression.

Sweet.js provides *infix macros* which allows a macro identifier to match syntax before it. For example, the following implements ES6-style arrow functions via infix macros:

```
macro (=>) {
    rule infix {
        ($params ...) | { $body ... }
    } => {
        function ($params ...) {
            $body ...
        }
    }
}

var id = (x) => { return x; }
```

TODO: details and limitations…

4.2 Invoke and Pattern Classes

TODO: motivation and details…

5. Hygiene

Mostly straightforward implementation from scheme with some details to handle `var`.

6. Implementation

Sweet.js is written in JavaScript and runs in the major JS environments (i.e. the brower and node.js). This is in contrast to Honu which translates its code to Racket code and reuses the hygienic expansion machinery already built in Racket. While this simplifies the implementation of Honu it also requires an installation of Racket which in some cases is not feasible (e.g. sweet.js is able to run in mobile device browsers).

7. Related Work

- Scheme/Racket
- Honu
- Template Haskell
- Nemerle
- Scala
- Closure

8. Conclusion

Figure 1: Read Algorithm

$$\texttt{Token} ::= \mathit{num} \mid \mathit{str} \mid + \mid /$$

```
read :: [Token] -> [ReadTree] -> [ReadTree]

    read([num, ...rest], prefix)             =  cons(num, read(rest, cons(num, prefix)))
    read([str, ...rest], prefix)             =  cons(str, read(rest, cons(str, prefix)))
    read([+, ...rest], prefix)               =  cons(+, read(rest, cons(+, prefix)))
    read([/, ...rest], [num, ...prefix])     =  cons(/, read(rest, cons(/, num, prefix)))
    read([/, ...rest], [str, ...prefix])     =  cons(/, read(rest, cons(/, str, prefix)))
    read([/, ...rest], [regex, ...prefix])   =  cons(/, read(rest, cons(/, regex, prefix)))
    read([/, ...rest], [+, ...prefix])       =  cons(regex, read(regexRest, cons(regex, +, prefix)))
                                                where (regex, regexRest) = scanRegex(rest)
    read([/, ...rest], [/, ...prefix])       =  cons(regex, read(regexRest, cons(regex, /, prefix)))
                                                where (regex, regexRest) = scanRegex(rest)
```