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 history of programming languages as a tool to provide syntactic flexibility to programmers going back at least to lisp and `defmacro`.

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. 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.

Figure 1: Simple JS Grammar

$$
\begin{array}{rcl}
Literal & ::= & num \mid str \mid /regex/ \\
\\
Expr & ::= & Literal \\
 & \mid & Expr\ +\ Expr \\
 & \mid & Expr\ /\ Expr
\end{array}
$$

2.1 Proof

The proof of sweet.js reader is something like holding on to a prefix for each non-terminal in the grammar:

$$
\begin{array}{rcl}
NT_g & : & /NT_h \\
NT_{g'} & : & /regex/NT_{h'}
\end{array}
$$

The simplified `read` algorithm can be defined as:

Just need to show that the prefix `g` is contained or identical with the prefixes recognized by the reader algorithm.

The prefix will be some context fee grammar of tokens or something like that.

3. 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.

3.1 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

Figure 2: Read Algorithm

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

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

$$
\begin{array}{lcl}
\texttt{read}([\mathit{num},\ \ldots \text{rest}],\ \text{prefix}) & = & \text{cons}(\mathit{num},\ \texttt{read}(\text{rest},\ \text{cons}(\mathit{num},\ \text{prefix}))) \\
\texttt{read}([\mathit{str},\ \ldots \text{rest}],\ \text{prefix}) & = & \text{cons}(\mathit{str},\ \texttt{read}(\text{rest},\ \text{cons}(\mathit{str},\ \text{prefix}))) \\
\texttt{read}([+,\ \ldots \text{rest}],\ \text{prefix}) & = & \text{cons}(+,\ \texttt{read}(\text{rest},\ \text{cons}(+,\ \text{prefix}))) \\
\texttt{read}([/,\ \ldots \text{rest}],\ [\mathit{num},\ \ldots \text{prefix}]) & = & \text{cons}(/,\ \texttt{read}(\text{rest},\ \text{cons}(/,\ \mathit{num},\ \text{prefix}))) \\
\texttt{read}([/,\ \ldots \text{rest}],\ [\mathit{str},\ \ldots \text{prefix}]) & = & \text{cons}(/,\ \texttt{read}(\text{rest},\ \text{cons}(/,\ \mathit{str},\ \text{prefix}))) \\
\texttt{read}([/,\ \ldots \text{rest}],\ [\mathit{regex},\ \ldots \text{prefix}]) & = & \text{cons}(/,\ \texttt{read}(\text{rest},\ \text{cons}(/,\ \mathit{regex},\ \text{prefix}))) \\
\texttt{read}([/,\ \ldots \text{rest}],\ [+,\ \ldots \text{prefix}]) & = & \text{cons}(\mathit{regex},\ \texttt{read}(\text{regexRest},\ \text{cons}(\mathit{regex},\ +,\ \text{prefix}))) \\
 & & \textit{where}\ (\mathit{regex},\ \text{regexRest}) = \text{scanRegex}(\text{rest}) \\
\texttt{read}([/,\ \ldots \text{rest}],\ [/,\ \ldots \text{prefix}]) & = & \text{cons}(\mathit{regex},\ \texttt{read}(\text{regexRest},\ \text{cons}(\mathit{regex},\ /,\ \text{prefix}))) \\
 & & \textit{where}\ (\mathit{regex},\ \text{regexRest}) = \text{scanRegex}(\text{rest})
\end{array}
$$

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...

3.2 Invoke and Pattern Classes

TODO: motivation and details...

4. Hygiene

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

5. 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).

6. Related Work

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

7. Conclusion