The Semantics of Syntax

Applying Denotational Semantics to Hygienic Macro Systems

Neelakantan R. Krishnaswami
University of Birmingham
<N.Krishnaswami@cs.bham.ac.uk>

1. Introduction

Typically, when semanticists hear the words "Scheme" or "Lisp", what comes to mind is "untyped lambda calculus plus higher-order state and first-class control". Given our typical concerns, this seems to be the essence of Scheme: it is a dynamically typed applied lambda calculus that supports mutable data and exposes first-class continuations to the programmer. These features expose a complete computational substrate to programmers so elegant that it can even be characterized mathematically; every monadically-representable effect can be implemented with state and first-class control [4].

However, these days even mundane languages like Java support higher-order functions and state. So from the point of view of a working programmer, the most distinctive feature of Scheme is something quite different – its support for *macros.* The intuitive explanation is that a macro is a way of defining rewrites on abstract syntax trees. However, Scheme programmers typically pair this explanation with the advice that getting macros right is rather subtle, and that one should avoid macros unless the usual abstraction mechanisms of the language have been tried first and have failed.

Because of this subtlety, macros remain a distinctive feature of the Lisp family, and have so far failed to make the jump to other functional languages; neither ML nor Haskell have formal macro systems. (GHC Haskell offers programmatic access to the AST via Template Haskell, though many users seemingly hate this feature. Ocaml used to have a macro system — CamlP4 — but this macro system has actually been removed from recent versions of the language!) This, despite the fact that most functional languages have features, such as pattern matching and do-notation, which have compositional, local desugarings that would lend themselves to being defined and implemented as macros.

This gap arises for surprising and deep reasons, which the description of a macro as an AST rewriting glosses over. Without a deeper understanding of these reasons, the technology developed for Scheme does not immediately transfer to typed languages, because it is not obvious how to report errors in terms of the the source rather than the expansion.

2. Three Features of Macro Systems

Below, we lay out the three key features of modern macro systems which both give them their power, and make them challenging to understand.

2.1 Hygiene

One of the most celebrated features of Scheme's macro system is its support for *hygiene* [10] – names introduced in the body of a macro definition do not interfere with names occurring in the macro's arguments. Consider this definition of a short-circuiting `and` operator:

```
(define-syntax and
  (syntax-rules ()
    ((and e1 e2) (let ((tmp e1))
                   (if tmp
                       e2
                       tmp)))))
```

In this definition, even if the variable `tmp` occurs freely in `e2`, it will not be in the scope of the variable definition in the body of the `and` macro. As a result, it is important to interpret the body of the macro not merely as a piece of raw syntax, but as an alpha-equivalence class.

2.2 Open Recursion

Another essential feature of macro systems (albeit one which is not usually remarked upon explicitly) are their support for open recursion. That is, having defined the `and` macro, we can similarly define a short-circuiting `or` macro as follows:

```
(define-syntax or
  (syntax-rules ()
    ((or e1 e2) (let ((tmp e1))
                  (if tmp
                      tmp
                      e2)))))
```

Then, expressions like `(and (or b1 b2) b3)` will "just work". This illustrates the key advantage of macros over straightforward AST-to-AST rewriting — a macro will continue to work in the presence of other macros, even ones that had not been anticipated ahead of time. If `and` and `or` had been implemented by an AST rewriter, then the presence of the `or` would have confused the `and` rewriter, and vice-versa.

2.3 Discovering Binding Structure

Despite macro hygiene, Scheme still permits the definition of so-called *exotic identifiers* [8], which makes it quite difficult to determine the appropriate notion of alpha-equivalence for source programs. Below, I give a simple macro illustrating the issue:

```
(define-syntax perverse
   (syntax-rules ()
      ((perverse x e)  (cons (lambda (x) e)
                             '((x) e)))))
```

The `x` argument to the `perverse` macro is used both as a binder in the lambda in the first component of the cons

cell that `perverse` creates, and as a symbol in the second component of the cons cell. As a result, `(perverse x (+ x x))` is not alpha-equivalent to `(perverse y (+ y y))`, even though the first argument is a binder.

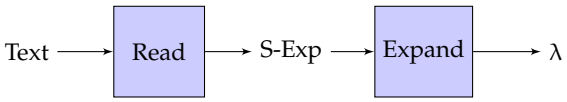
3. Problems and Challenges

The combination of exotic identifiers, hygiene and open recursion is a very challenging one from the theoretical point of view. We want macro bodies to respect alpha-equivalence, but because of open recursion, macro bodies may contain macros we do not know about, and these macros may well be exotic. As a result, it is not immediately clear how to even *define* what alpha-equivalence means for source programs!

The traditional approach to explaining hygiene [2, 10] has been an operational one, where the details of the algorithm are exposed to the programmer so that (with enough study) it becomes clear that alpha-equivalence has been maintained. Herman and Wand [9] extend this approach by giving types to macros, *specifying* their binding structure, and then proving a type safety result to ensure that expansion respects this specification. Adams [1] takes a more semantic approach, using nominal sets to specify invariance under renaming, and uses an unusual doubled interpretation of potential binders to account for potential exotic identifiers.

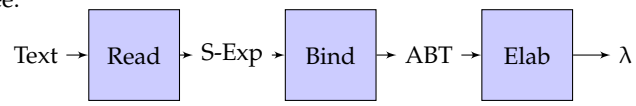
4. Approach

Herman [7] has remarked that the key engineering feature of macros is that where traditional compilers take raw text and parse it into an AST, a language with a macro system takes text, reads it into an intermediate form (e.g., s-expressions), and then recursively runs macro-expansion until a core form is found. This can be illustrated with the following diagram:



Our idea is that a semantic understanding to macro-expansion can be achieved by splitting this process up further. Macro-expansion interleaves the discovery of binding structure with rewriting. We can take a page from the Nuprl [3] playbook, and add a second intermediate stage of *abstract binding trees* [6], which are essentially syntax trees annotated with binding information. Unlike the λ-calculus, they are pure data: the only equations they satisfy are alpha-equivalence.

So we can split macro expansion into two phases, binding and elaboration. Binding takes a tree *without* any binding structure and identifies binding sites and variable usages, and then elaboration is rewriting upon the abstract binding tree.



Happily, semantic models of trees with binding are already available; Fiore et al [5] show how to interpret datatypes with binders as ordinary inductive datatypes in the functor category $\mathrm{Set}^{\mathbb{I}}$ of functors from the category of finite sets and injections $\mathbb{I}$ to Set. Intuitively, the idea is that we work in a Kripke model where the Kripke worlds are finite sets of variable names, with injections corresponding to renaming and weakening.

This idea can be seen as a reformulation of the approach of Adams [1] – this functor category is *also* a model of nominal sets [11]. The naturality condition on morphisms in the functor category then corresponds to invariance under renaming, and separating binding from elaboration permits us to drop the doubled interpretation of identifiers.

While this is seemingly a mere change in viewpoint, it is a very powerful one. Once we begin thinking of free variables as living in a context, we can consider other notions of context. In particular, we can consider the category of *typed* contexts, where each variable also carries a type. Then, by interpreting transformations in the functor category over the category of contexts, the usual naturality conditions will ensure that we do not define transformations which generate ill-typed expressions! So by interpreting a macro language in the internal language of the functor category, we can ensure that all macros are, by construction, typesafe.

Furthermore, each phase of expansion can be understood as a way of taking data from a less-structured category into a more-structured one: s-expressions are merely trees, abstract binding trees are trees with binding, and typed lambda terms have typed binding forms.

5. Conclusion

Because this is work-in-progress, everything written here should be regarded as conjectural. However, we hope to give a talk showing how we can decompose macro systems in such a way that deep ideas from semantics become applicable to them, making their invariants plain and (hopefully) laying out a roadmap to bring macros to typed languages.

References

[1] Michael D. Adams. Towards the essence of hygiene. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 457–469, 2015.

[2] William D. Clinger and Jonathan Rees. Macros that work. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, pages 155–162, 1991.

[3] RL Constable, SF Allen, HM Bromley, WR Cleaveland, JF Cremer, RW Harper, DJ Howe, TB Knoblock, NP Mendler, P Panangaden, et al. *Implementing mathematics*. Prentice-Hall, Inc., 1986.

[4] Andrzej Filinski. Representing monads. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 446–457, 1994.

[5] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 193–202. IEEE Computer Society, 1999.

[6] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

[7] David Herman. Homoiconicity isn't the point. `http://calculist.org/blog/2012/04/17/homoiconicity-isnt-the-point/`. Accessed: 12 Nov 2015.

[8] David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, MA, 5 2010.

[9] David Herman and Mitchell Wand. A theory of hygienic macros. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008*, pages 48–62, 2008.

[10] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *LISP and Functional Programming*, pages 151–161, 1986.

[11] Ian Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1):77–107, 1996.