Macro-by-Example:
Deriving Syntactic Transformations from their Specifications

Eugene E. Kohlbecker, Indiana University
Mitchell Wand, Northeastern University

Authors' addresses: Eugene E. Kohlbecker, Computer Science Department, University of Rhode Island, Kingston, RI 02881. Mitchell Wand, College of Computer Science, Northeastern University, 360 Huntington Avenue #161CN, Boston, MA 02115.

Abstract

This paper presents two new developments. First, it describes a "macro-by-example" specification language for syntactic abstractions in Lisp and related languages. This specification language allows a more declarative specification of macros than conventional macro facilities do by giving a better treatment of iteration and mapping constructs. Second, it gives a formal semantics for the language and a derivation of a compiler from the semantics. This derivation is a practical application of semantics-directed compiler development methodology.

1. Introduction

Modern programming languages offer powerful facilities for procedural and data abstraction which encapsulate commonly-used patterns of procedure invocation and of data usage, respectively. Often, however, one encounters typical patterns of linguistic usage that do not fall neatly into either category. An example is the `let` expression, which encapsulates an often-used pattern of function creation and call:

```
(let ((i e) ...) b ...)
  ==> ((lambda (i ...) b ...) e ...)
```

This is not a procedural abstraction; rather, it is a *syntactic* abstraction: an abstraction of a typical pattern of syntax.

Most modern languages do not provide tools for the creation of syntactic abstractions comparable to those available for procedural or data abstractions. Even in languages such as Lisp that allow syntactic abstractions, the process of defining them is notoriously difficult and error-prone. To define `let` as a macro, we must write a procedure that transforms any expression which matches the left-hand pattern into a corresponding instance of the right-hand pattern. The code for this looks

like:

```
(lambda (s)
  (cons
   (cons 'lambda
         (cons (map car (cadr s))
               (cddr s)))
   (map cadr (cadr s))))
```

This code can hardly be considered transparent. It bears no obvious relation to the transformation it engenders. It is difficult to see the shape of the structure it is building or the shape of the structure it is trying to take apart. Furthermore, it does no error checking on its input.

Modern Lisps supply some tools to help the macro-writer, most notably backquote and `defmacro` (*e.g.,* [Foderaro, Sklower, & Layer 83]). Backquote makes the code for building the output look more like the output itself, and `defmacro` includes a pattern-matching facility. Using these tools, the `let` macro might be defined as:

```
(defmacro let (decls . body)
  `((lambda ,(map car decls) . ,body)
    . ,(map cadr decls)))
```

This is considerably better, but the mapping functions are still mysterious. Furthermore, the backquote mechanism is itself error-prone: one *always* leaves out at least one comma on the first try!

In our facility, one defines `let` as follows:

```
(declare-syntax let
  [(let ((i e) ...) b ...)
   ((lambda (i ...) b ...) e ...)])
```

This is close to the language used for specifying syntactic extensions in the revised Scheme report [Steele and Sussman 78] and the 1986 Scheme report [Rees, Clinger, *et al.* 86], except that it is executable. The specification language has the following features:

1. Pattern-matching, including error-checking, is done on the input.

2. The output specification matches the form of the output. No commas or other symbols are needed.

3. Repetitive elements are specified naturally on both input and output.

This specification mechanism has been used in various versions of Scheme since 1982, most recently in Chez Scheme [Dybvig 87], and has proved to be a robust and highly useful feature. Only recently, however, did the need for formal documentation of the mechanism lead us to develop the formal semantics and the semantically-derived compiler presented here.

We call this mechanism macro-by-example, or MBE. It has allowed us to embed a number of interesting languages in Scheme, such as:

- a type-checked Scheme variant called SPS [Wand 84]
- a coroutine mechanism [Friedman, Haynes, & Wand 86]
- an import/export mechanism for modules [Felleisen and Friedman 86]
- two quite different semantically-derived subsets of Prolog [Felleisen 85, Wand 85]

The presence of this convenient syntactic extension tool provides an important design dimension: given any proposed language extension, we get to decide how much of it should be treated as procedural abstraction, how much as data abstraction, and how much as syntactic abstraction. The presence of a good syntactic abstraction mechanism means that a language extension which might be highly cumbersome expressed in a purely procedural way can be made far more usable by a propitious choice of syntax. (Indeed, the `declare-syntax` facility itself may be viewed as such a propitious choice of syntax). Conversely, given a complex syntactic abstraction, we can try to convert some of it to a procedural or a data abstraction.

In Section 2, we give a brief description of the specification language itself and additional examples. In Section 3, we present a formal semantics of the specification language. This semantics could, if desired, be used to create an interpreter for this language of transformations. In Sections 4 and 5, we show how the semantics can be transformed into a compiler which takes a transformation specification and produces Scheme code for performing the transformation (including error checking and mapping functions).

2. The Specification Language

Consider the following example of MBE:

```
(declare-syntax and
  [(and) true]
  [(and e) e]
  [(and e1 e2 ...) (if e1 (and e2 ...) false)])
```

It illustrates some of the important features of MBE:

1. An MBE specification consists of a series of input-output patterns, which are tried in order. If a call matches none of the input patterns, then an error is signalled.

2. Backquotes and commas in the output patterns are avoided by a simple convention: identifiers which appear in the input pattern are treated as pattern variables; all other identifiers are constants. No special treatment is necessary for `lambda` or `quote` in the transcription proper, but the result of the transcription can be processed using the α-converting expander of [Kohlbecker, *et al.* 86] to avoid capture of bound variables in macros such as:

```
(declare-syntax or2
  [(or2 x y) (let ((v x)) (if v v y))])
```

which would otherwise capture any occurrences of `v` in its second argument. (The production version of MBE also allows the user to specify the capture of particular variables).

3. Whenever the ellipsis (indicated by the atom "`...`") is used, it must be preceded by a pattern, and it must be the last element of the pattern list or sublist. (Again, the production version is somewhat more generous; we do not illustrate this here). Ellipses may be nested, so pattern variables may denote S-expressions, lists of S-expressions, lists of lists of S-expressions, *etc.*

As an illustration of MBE as a language-embedding tool, consider the following macros, which are taken from [Wand 85]. These macros define two new special forms whose translation produces (through details that are of no concern here) an interface to an underlying Prolog semantics.

```
(declare-syntax clauses
   ((clauses (var1 ...) cl1 ...)
    (lambda (z)
       (let ((var1 (genvar)) ...)
            (call/kf
               (lambda (kf)
                   (ff-set-ref! **cut-point** kf)
                   (choose (clause cl1) ...)))))))))

(declare-syntax clause
   ((clause (pat act1 ...))
    (begin
       (unify (pattern pat) z)
       (action act1) ...)))

(declare-syntax action
   ((action (fn e1 ...))
    (fn (list (pattern e1) ...))))
```

Without the ability to adapt the syntax, the interface would have been impossible to use (indeed, an earlier version of our Prolog semantics foundered on precisely this point). With these macros in place, we can write Prolog predicates as Scheme procedures in a moderately convenient syntax, for example:

```
(define Append
   (clauses (y a d u v)
      ((nil y y))
      (((a . d) u (a . v))
       (Append d u v))))
```

The production version [Kohlbecker 86] includes a number of additional bells and whistles. First, arbitrary tests may be specified by an optional argument called

a *fender*. For example, the macro for `let` should include a test to make sure the bound variables are atoms:

```
(declare-syntax let
  [(let ((i e) ...) b ...)
   (mapand atom? '(i ...))
   ((lambda (i ...) b ...) e ...)])
```

Second, arbitrary processing of the macro call may be performed by a `with`-specification, which binds its variable to the result of a Scheme calculation.

```
(declare-syntax new-lang
  [(new-lang e ...)
   (with ([code (translate-to-scheme '(e ...))])
         (top-level code))])
```

And third, the macro-writer may also specify *keywords* to serve as pattern constants in the input patterns (*e.g.,* `else`) and as terminators for ellipses.

3. Semantics

In this section, we sketch the formal semantics of the MBE mechanism. We restrict ourselves to a single input-output pair, emphasizing the correct treatment of ellipses.

In our description, we employ relatively standard notation. We use three closely-spaced dots (...) for the ellipsis symbol and center dots ($\cdots$) for ellipsis in the meta-language. We use $(x)_1$, $(x)_2$ for projection, $(\alpha \Rightarrow \beta, \gamma)$ for conditional, and $A \rightarrowtail B$ for the set of partial functions from A to B.

We define the following domains:

$$S\text{-}exp ::= Ident \mid (S\text{-}exp \; \cdots \; S\text{-}exp)$$
$$S^\sharp ::= S\text{-}exp \mid (S^\sharp \; \cdots \; S^\sharp)$$
$$Pat ::= (\,) \mid Ident \mid (Pat \,.\, Pat) \mid (Pat \,...)$$
$$Env = Ident \rightarrowtail (Int \times S^\sharp)$$

The function *fv*, defined over *Pat*, returns a list of identifiers contained in a pattern. The meta-variable a ranges over *Ident*, p over *Pat*, s over *S-exp* and $S^\sharp$, and ρ over *Env*.

The semantics is comprised of three main functions:

$$\mathcal{B} \colon Pat \rightarrow S\text{-}exp \rightarrow Bool$$
$$\mathcal{D} \colon Pat \rightarrow S\text{-}exp \rightarrowtail Env$$
$$\mathcal{T} \colon Pat \rightarrow Env \rightarrowtail S\text{-}exp$$

$\mathcal{B}$ takes an input pattern and an S-expression; it returns a boolean indicating whether the S-expression matches the pattern. $\mathcal{D}$ takes an input pattern and an

S-expression and (if it matches the pattern) returns an environment associating pattern variables with their resulting bindings. Each binding is a pair consisting of a non-negative integer and an element of $S^\sharp$. The integer indicates the variable's level, the number of ellipses enclosing it. The element of $S^\sharp$ is of the corresponding type: if the level is 0, it is an S-expression; if the level is 1, it is a list of S-expressions; if the level is 2, a list of lists of S-expressions, *etc.*

$\mathcal{T}$ takes an output pattern and an environment and expands the pattern in that environment. Thus a transform function with type *S-exp* $\multimap$ *S-exp* for one input-output pattern pair (*lhs rhs*) may be described as:

$$\mathcal{E}[\![(lhs\ rhs)]\!] = \lambda s.(\mathcal{B}[\![lhs]\!]s \Rightarrow \mathcal{T}[\![rhs]\!](\mathcal{D}[\![lhs]\!]s),$$
$$error)$$

The variable s represents a macro invocation.

We begin with the clauses for $\mathcal{B}$. They are as follows:

$$\mathcal{B}[\![()]\!] = \lambda s.null?\,s$$
$$\mathcal{B}[\![a]\!] = \lambda s.true$$
$$\mathcal{B}[\![(p_1\ .\ p_2)]\!] = \lambda s.pair?\,s \wedge \mathcal{B}[\![p_1]\!](hd\,s) \wedge \mathcal{B}[\![p_2]\!](tl\,s)$$
$$\mathcal{B}[\![(p\ ...)]\!] = \lambda s.list?\,s \wedge mapand\,(\mathcal{B}[\![p]\!])\,s$$

The pattern () matches only the empty list. The pattern a is a variable and matches anything. The pattern $(p_1\ .\ p_2)$ matches s if and only if s is a pair (that is, not an atom and not empty), and the corresponding subpatterns match. The pattern $(p\ ...)$ matches s if and only if s is a list and p matches each element of s.

The equations for $\mathcal{D}$ are guaranteed to be sensible only on the condition that $\mathcal{B}[\![p]\!]s$ is true:

$$\mathcal{D}[\![()]\!] = \lambda s.\emptyset$$
$$\mathcal{D}[\![a]\!] = \lambda s.\{(a \mapsto (1, s))\}$$
$$\mathcal{D}[\![(p_1\ .\ p_2)]\!] = \lambda s.\mathcal{D}[\![p_1]\!](hd\,s) \cup \mathcal{D}[\![p_2]\!](tl\,s)$$

Here we assume that there are no variables in common between p_1 and p_2.

To create an environment from a pattern $(p\ ...)$ and a subject S-expression s, we map $\mathcal{D}[\![p]\!]$ across the elements of s, creating a list of environments. Then the environments are combined componentwise, incrementing the level associated with each variable:

$$\mathcal{D}[\![(p\ ...)]\!](s_1 \cdots s_n) = combine\text{-}envs\,(\mathcal{D}[\![p]\!]s_1 \cdots \mathcal{D}[\![p]\!]s_n)$$

$$combine\text{-}envs = \lambda(\rho_1 \cdots \rho_n).\lambda i.\langle(\rho_1 i)_1 + 1, ((\rho_1 i)_2 \cdots (\rho_n i)_2)\rangle$$

Thus, in the pattern (let ((i e) ...) b ...), the variable i becomes bound with level 1 to the list of matching elements in an invocation s.

We can now move on to the processing of the output pattern. Again, the equations are straightforward, except for ellipses:

$$\mathcal{T}[\![()]\!]\rho = ()$$
$$\mathcal{T}[\![a]\!]\rho = (a \in Dom\, \rho) \Rightarrow$$
$$(((\rho a)_1 = 0) \Rightarrow (\rho a)_2, error),$$
$$a$$
$$\mathcal{T}[\![(p_1\ .\ p_2)]\!]\rho = cons\,(\mathcal{T}[\![p_1]\!]\rho)\,(\mathcal{T}[\![p_2]\!]\rho)$$

If the output pattern is an identifier, we first check to see if it is bound in the environment. If it is, and its level is zero, then the result is its value. If the level is non-zero, then it was bound to a list of values, not a value, so it may not be transcribed. If it is not bound in the environment, then it is treated as a constant.

If the output pattern is a pair $(p_1\ .\ p_2)$, its head and tail are each processed separately. The two results are then joined.

In processing an output pattern $(p\ ...)$ with $\mathcal{T}$, we have the inverse of the problem with $\mathcal{D}$: we must split a single environment into a list of environments. To do this, we first restrict the environment to the free variables of p. We then check whether the variables of p include at least one variable of level greater than 0. If not, the prototype p is rejected, since there is no way to determine the length of the repetition. If the prototype passes, then we decompose the environment into a list of environments and map $\mathcal{T}[\![p]\!]$ over this list. The decomposition is complicated by three considerations: First, we must decrement the level of each non-scalar variable in the environment. Second, we copy scalar (level 0) variables across the list, being sure *not* to decrement their level, so that code like

```
(declare-syntax copy-it
  [(copy-it i j ...) (bar '(i j) ...)])
```

correctly transcribes

```
(copy-it a 1 2 3)
```

into

```
(bar '(a 1) '(a 2) '(a 3)).
```

Last, we must determine the length of the list. This is the length of the lists in the environment if they are all the same length. If they are not the same length, an

7

error is signalled. These tasks are performed by the following clauses:

$$\mathcal{T}[\![(p\ ...)]\!]\rho = controllable\ p\ \rho \Rightarrow$$
$$map\ (\mathcal{T}[\![p]\!])\ (decompose\ (\rho \mid fv(p))),$$
$$error$$

$$controllable\ p\ \rho = \exists v\ (v \in dom(\rho \mid fv(p)) \wedge ((\rho\,v)_1 > 0))$$

$$decompose\ \rho = UnequalLengths?\ \rho \Rightarrow error,$$
$$StopNow?\ \rho \Rightarrow (),$$
$$cons\ (split\ hd \circ \rho)$$
$$decompose\ (split\ tl \circ \rho)$$

$$split\ f\ \langle n, c \rangle = (n = 0) \Rightarrow \langle 0, c \rangle, \langle (n-1), (fc) \rangle$$

This completes the semantics.

A fender or `with`-specification may easily be modeled using this semantics. They are the same as ordinary output patterns: when encountered they are expanded using $\mathcal{T}$ and then evaluated. The result of the evaluation is then used either as a guard to decide whether to proceed with the expansion or as a value to be inserted in the environment for the main expansion.

4. Deriving the Compiler

The original versions of MBE (along with the current production version, detailed in [Kohlbecker 86]), used *ad hoc* code generation methods to compile a specification into Scheme code. The code generator is quite clever; for example, for `let` it produces the code shown in the first display. The evolution of this code generator has been tortuous, and therefore MBE seemed to be a good real-world example on which to exercise semantic methods.

The first step in the derivation was the production of the formal semantics sketched above. From there, the derivation proceeded in two phases: *staging* [Jorring and Sherlis 86] and *representation* [Wand 82].

The goals of staging are to identify opportunities for early binding and to re-order and re-curry the arguments to the various functions so that the early-bindable arguments appear first.

Our particular goal is to avoid building intermediate structures such as environments whenever possible. Recall that $\mathcal{D}$ has functionality

$$Pat \rightarrow S\text{-}exp \multimap (Ident \multimap (Int \times S^\sharp))$$

We delay the binding of *S-exp* by replacing *Env* by

$$Env' = (Ident \multimap Int) \times (Ident \multimap S\text{-}exp \rightarrow S^\sharp)$$

and replacing $\mathcal{D}$ by a function $\mathcal{D}'$ of functionality $Pat \to Env'$.

An element of Env' may be thought of as a symbol table. Instead of having the function $\mathcal{D}$ which takes an S-expression and builds an environment containing S-expressions, we have a new function which takes a pattern and builds a symbol table containing functions from S-expressions to S-expressions. We call these functions *selectors*. A selector is later applied to the subject S-expression to extract the values for the variables. Thus the condition we want is that if $\mathcal{B}[\![p]\!]s$ is true, then for all i,

$$\langle (\mathcal{D}'[\![p]\!])_1 i, (\mathcal{D}'[\![p]\!])_2 is \rangle = \mathcal{D}[\![p]\!]si$$

This formulation also makes clear that the levels of the variables are independent of the subject S-expression to which the pattern is matched. (This is analogous to static chain analysis or type analysis).

It is easy to build $\mathcal{D}'$ as the product of two functions, which we call $\mathcal{D}'_1$ and $\mathcal{D}'_2$. Each of these is obtained by modifying the appropriate piece of the definition of $\mathcal{D}$, using the congruence condition above as a guide. Doing this, we get:

$$\mathcal{D}'_1[\![()]\!] = \emptyset$$
$$\mathcal{D}'_1[\![a]\!] = \{(a \mapsto 0)\}$$
$$\mathcal{D}'_1[\![(p_1 \ . \ p_2)]\!] = \mathcal{D}'_1[\![p_1]\!] \cup \mathcal{D}'_1[\![p_2]\!]$$
$$\mathcal{D}'_1[\![(p \ ...)]\!] = add1 \circ \mathcal{D}'_1[\![p]\!]$$

$$\mathcal{D}'_2[\![()]\!] = \emptyset$$
$$\mathcal{D}'_2[\![a]\!] = \{(a \mapsto \lambda s.s)\}$$
$$\mathcal{D}'_2[\![(p_1 \ . \ p_2)]\!] = \{(i \mapsto \lambda s.f(hd \ s)) \mid (i \mapsto f) \in \mathcal{D}'_2[\![p_1]\!]\}$$
$$\cup \{(i \mapsto \lambda s.f(tl \ s)) \mid (i \mapsto f) \in \mathcal{D}'_2[\![p_2]\!]\}$$
$$\mathcal{D}'_2[\![(p \ ...)]\!] = \{(i \mapsto \lambda s.map \ f \ s) \mid (i \mapsto f) \in \mathcal{D}'_2[\![p]\!]\}$$

Now we can proceed to modify $\mathcal{T}$ to work with new-style environments. Just as the analysis of $\mathcal{D}$ produced selectors, the analysis of $\mathcal{T}$ produces *constructors*. The new version of $\mathcal{T}$ will be

$$\mathcal{T}' \colon Pat \to Env' \to \textit{S-exp} \multimap \textit{S-exp}$$

Using the condition on $\mathcal{D}'$ above to illustrate how to translate new environments into old environments, we see that the condition we want $\mathcal{T}'$ to obey is

$$\mathcal{T}'[\![p]\!]\rho's = \mathcal{T}[\![p]\!](\lambda i.\langle (\rho')_1 i, (\rho')_2 is \rangle) \qquad (*)$$

With this condition, we can redefine $\mathcal{E}$ as:

$$\mathcal{E}[\![(lhs \ rhs)]\!] = \lambda s.(\mathcal{B}[\![lhs]\!]s \Rightarrow \mathcal{T}'[\![rhs]\!](\mathcal{D}'[\![lhs]\!])s,$$
$$error)$$

A simple calculation shows that this definition is equivalent to the old one.

As usual, it is easy to write the first three clauses for $\mathcal{T}'$:

$$
\begin{aligned}
\mathcal{T}'[\![\,(\,)\,]\!]\rho' &= \lambda s.(\,) \\
\mathcal{T}'[\![a]\!]\rho' &= (a \in \mathit{Dom}\,(\rho')_1) \Rightarrow \\
&\qquad ((\rho')_1 a = 0 \Rightarrow (\rho')_2 a,\, \mathit{error}), \\
&\qquad (\lambda s.a) \\
\mathcal{T}'[\![\,(p_1\ .\ p_2)\,]\!]\rho' &= \lambda s.(\mathit{cons}\,(\mathcal{T}'[\![p_1]\!]\rho's)\,(\mathcal{T}'[\![p_2]\!]\rho's))
\end{aligned}
$$

The analysis of $\mathcal{T}'[\![\,(p\ ...)\,]\!]$ is harder; we defer it to the next section.

The staging analysis shows that at compile-time we can produce a set of tests, selectors, and constructors that can be applied at macro-expansion time to an S-expression. In the second phase of the derivation, we choose appropriate representations for these functions. In keeping with slogan "target code as a representation of semantics" [Wand 82], we represent these functions in Scheme, our target language.

We begin with the representation for selectors. To determine the representation, we look at $\mathcal{D}'_2$ to see what are the basic selectors and what are the constructors which build selectors. This gives us a mini-language of selectors. We then choose representations for this language in Scheme. In particular, we represent these functions as Scheme expressions with a single free variable `s`. Then the function can be applied simply by evaluating the expression in a suitable environment.

Looking at the definition of $\mathcal{D}'_2$, we can see that every selector is either $\lambda s.s$ or of one of the forms $\lambda s.(M(\mathit{hd}\,s))$, $\lambda s.(M(\mathit{tl}\,s))$ and $\lambda s.(\mathit{map}\,M\,s)$, where M is another selector. Thus we can represent these by the Scheme expressions `s`, $M[$`(car s)`$/$`s`$]$, $M[$`(cdr s)`$/$`s`$]$, and `(map (lambda (s) M) s)`, respectively, where $M[N/v]$ means the substitution of N for the free occurrences of v in M. We can then do a small amount of peephole optimization on the representations (*e.g.,* replacing `(lambda (s) (map (lambda (s) s) s))` by `s`). This is easy since the language of selectors is small. We can analyze tests and constructors similarly, by looking at the

definitions of $\mathcal{B}$ and $\mathcal{T}'$. We summarize the representations as follows:

Tests:

$s = ()$	`(null? s)`
$true$	`true`
$\lambda s.pair?\,s$	`(and (pair? s)`
$\quad \wedge M(hd\,s)$	$\quad\quad M[$`(car s)`$/$`s`$]$
$\quad \wedge N(tl\,s)$	$\quad\quad N[$`(cdr s)`$/$`s`$])$
$\lambda s.list?\,s$	`(and (list? s)`
$\quad \wedge mapand\,M\,s$	`(mapand`
	`(lambda (s)` M`)`
	`s))`

Selectors:

$\lambda s.s$	`s`
$M(hd\,s)$	$M[$`(car s)`$/$`s`$]$
$M(tl\,s)$	$M[$`(cdr s)`$/$`s`$]$
$\lambda s.map\,M\,s$	`(map (lambda (s)` M`) s)`

Constructors:

$\lambda s.()$	`nil`
$\lambda s.a$	`a`
$\lambda s.(cons\,(Ms)\,(Ns))$	`(cons` M N`)`

Since the same representation is used for all these functions, they can be intermixed when necessary. For example, in $\mathcal{T}'$ there is no separate representation for $(\rho')_2 a$, since this is a selector whose representation is already determined.

Having decided on a representation, we then go back and modify the functions $\mathcal{B}$, $\mathcal{D}'_2$ and $\mathcal{T}'$ to produce these concrete representations rather than the functions. This gives us a compiler: a function which takes a pattern-transcription pair and produces a piece of Scheme code which performs the transliteration.

More precisely, we replace $\mathcal{E}$ by

$$\mathcal{E}''[\![(lhs\ rhs)]\!]$$
$$= \texttt{(lambda (s)}$$
$$\quad\quad \texttt{(if } \mathcal{B}''[\![lhs]\!]$$
$$\quad\quad\quad \mathcal{T}''[\![rhs]\!](\mathcal{D}''[\![lhs]\!])$$
$$\quad\quad \texttt{(error "pattern not matched")))}$$

where the doubly-primed functions produce the concrete representations specified in the table above. Each doubly-primed function is obtained by modifying the corresponding singly-primed function so that instead of returning a function, it produces the representation of that function.

The function $\mathcal{B}$ is replaced by:

$$\mathcal{B}''[\![()]\!] = (\texttt{null? s})$$
$$\mathcal{B}''[\![a]\!] = \texttt{true}$$
$$\mathcal{B}''[\![(p_1 \,.\, p_2)]\!] = (\texttt{and (pair? s)}$$
$$\mathcal{B}''[\![p_1]\!][(\texttt{car s})/\texttt{s}]$$
$$\mathcal{B}''[\![p_2]\!][(\texttt{cdr s})/\texttt{s}])$$
$$\mathcal{B}''[\![(p \,...)]\!] = (\texttt{and (list? s)}$$
$$(\texttt{mapand (lambda (s) } \mathcal{B}''[\![p]\!]) \texttt{ s})$$

$\mathcal{D}''[\![lhs]\!]$ produces a pair, consisting of a level-environment, $\mathcal{D}'_1[\![lhs]\!]$ as before, and an environment of concrete selectors in which the selector functions are replaced by their representations. Only the selectors need to be represented concretely, since these are the only things that $\mathcal{T}''$ inserts in its concrete output.

$$\mathcal{D}''[\![lhs]\!] = \langle \mathcal{D}'_1[\![lhs]\!], \mathcal{D}''_2[\![lhs]\!] \rangle$$

$$\mathcal{D}''_2[\![()]\!] = \emptyset$$
$$\mathcal{D}''_2[\![a]\!] = \{(a \mapsto \texttt{s})\}$$
$$\mathcal{D}''_2[\![(p_1 \,.\, p_2)]\!] =$$
$$\{(i \mapsto M[(\texttt{car s})/\texttt{s}]) \mid (i \mapsto M) \in \mathcal{D}''_2[\![p_1]\!]\}$$
$$\cup \{(i \mapsto M[(\texttt{cdr s})/\texttt{s}]) \mid (i \mapsto M) \in \mathcal{D}''_2[\![p_2]\!]\}$$
$$\mathcal{D}''_2[\![(p \,...)]\!] = \{(i \mapsto (\texttt{map (lambda (s) } M \texttt{) s}))$$
$$\mid (i \mapsto M) \in \mathcal{D}''_2[\![p]\!]\}$$

The function $\mathcal{T}$ becomes:

$$\mathcal{T}''[\![()]\!]\rho' = \texttt{nil}$$
$$\mathcal{T}''[\![a]\!]\rho' = (a \in Dom\,(\rho')_1) \Rightarrow$$
$$((\rho')_1 a = 0 \Rightarrow (\rho')_2 a, \, error),$$
$$a$$
$$\mathcal{T}''[\![(p_1 \,.\, p_2)]\!]\rho' = (\texttt{cons } \mathcal{T}''[\![p_1]\!]\rho' \; \mathcal{T}''[\![p_2]\!]\rho')$$

Note that all this code is obtained by modifying the output of the previous functions, not by doing any *ab initio* coding or analysis. In Scheme, this modification may be accomplished by simply introducing backquotes in the appropriate places.

This completes the compiler, except for the transcription of ellipses.

5. Transcribing Ellipses

Transcribing ellipses is a harder problem for this compilation strategy, since in general there is no way to avoid building intermediate structures akin to environments.

Our strategy is to deal first with the most common special case, which luckily does not require any intermediate structures. It is easy to confirm that the following clause satisfies the congruence relation $(*)$, given in the previous section, which must hold between $\mathcal{T}$ and $\mathcal{T}'$:

$$\mathcal{T}'[\![(a\ ...)]\!]\rho' = ((\rho')_1 a = 1) \Rightarrow (\rho')_2 a,\ error$$

This takes care of what seems to be the most common use of ellipses. In general, however, we need to attack the congruence relation for $\mathcal{T}'[\![(p\ ...)]\!]$ directly. The congruence relation requires that

$$\mathcal{T}'[\![(p\ ...)]\!]\rho' s$$
$$= \mathcal{T}[\![(p\ ...)]\!](\lambda i.\langle(\rho')_1 i, (\rho')_2 is\rangle)$$
$$= map\,(\mathcal{T}[\![p]\!])$$
$$(decompose\,(\lambda i \in fv(p).\langle(\rho')_1 i, (\rho')_2 is\rangle))$$

Let us write ρ for the argument to $decompose$ above. We need to build some representation of ρ and of $decompose\,\rho$. That representation should allow us to build as much as possible of the environment before knowing the value of s. Once that is done, we can attack the problem of replacing the $\mathcal{T}$ in the mapping expression by $\mathcal{T}'$.

Let $i^* = [i_0, \cdots, i_k]$ be an enumeration of the free variables of p in a list. (We use the square brackets $[\cdots]$ or $[\cdots l \cdots \mid l \in L]$ to denote lists). We represent all functions whose domain is $fv(p)$ as lists, indexed by i^*. Thus a function f can be represented by the list $[f(i_0), \cdots, f(i_k)]$.

Since an element of Env' is a pair of functions, we represent it as a pair of lists. We refer to these two lists as the *level* component and the *value* component of the environment. Hence $decompose\,\rho$ becomes

$$decompose\,[(\rho')_1 i \mid i \in i^*]\,[(\rho')_2 is \mid i \in i^*]$$

The output from $decompose$ is a list of environments, each of which has the same level component. Hence we can represent this list of environments as a single level component and a list of value components. Thus we can represent $decompose(\lambda i \in fv(p).\langle(\rho')_1 i, (\rho')_2 is\rangle)$ as

$$\langle decompose\text{-}levels\,n^*,\ decompose\text{-}values\,n^*\,s^*\rangle$$

where
$$n^* = [(\rho')_1 i \mid i \in i^*]$$
$$s^* = [(\rho')_2 is \mid i \in i^*]$$
$$decompose\text{-}levels = \lambda n^*.[((n = 0) \Rightarrow 0, n - 1) \mid n \in n^*]$$
$$decompose\text{-}values$$
$$= \lambda n^* s^*.UnequalLengths?\,s^* \Rightarrow error,$$
$$StopNow?\,\rho \Rightarrow (),$$
$$cons\,(split^*\,hd\,n^*\,s^*)$$
$$decompose\text{-}values\,n^*\,(split^*\,tl\,n^*\,s^*)$$
$$split^* = \lambda f n^* s^*.[(n = 0) \Rightarrow s, fs \mid (n, s) \in (n^*, s^*)]$$

13

Here the definition of $split^*$ describes mapping over two lists of equal length.

By these manipulations, we have given ourselves the possibility of precomputing the levels. We take advantage of this by currying the functionality of $\mathcal{T}'$, changing it from

$$\mathcal{T}' \colon Pat \to ((Ident \multimap Int) \times (Ident \multimap S\text{-}exp \to S^\sharp))$$
$$\to S\text{-}exp \multimap S\text{-}exp$$

to

$$\mathcal{T}' \colon Pat \to (Ident \multimap Int) \to (Ident \multimap S\text{-}exp \to S^\sharp)$$
$$\to S\text{-}exp \multimap S\text{-}exp$$

We may also make a similar change in the functionality of $\mathcal{T}$ to reflect our new representation of environments:

$$\mathcal{T} \colon Pat \to (Ident \multimap Int) \to (Ident \multimap S^\sharp) \to S\text{-}exp$$

With these changes, the congruence condition between $\mathcal{T}$ and $\mathcal{T}'$ may be restated as:

$$\mathcal{T}'[\![p]\!]\rho_1\rho_2 s = \mathcal{T}[\![p]\!]\rho_1(\lambda i.\rho_2 is) \qquad (**)$$

Let ρ^* be the selector environment $\mathcal{D}_2'[\![i^*]\!]$, that is, the environment $\{(i_0 \mapsto \lambda s.(hd\,s)), (i_1 \mapsto \lambda s.(hd\,(tl\,s))), \cdots\}$. Hence, if we have an environment represented by the pair (n^*, s^*), we can use ρ^* with $\mathcal{T}'$ to retrieve elements from s^*. More precisely, we may deduce from the congruence condition $(**)$ that for any pattern p', we have

$$\mathcal{T}'[\![p']\!]n^*\rho^*s^* = \mathcal{T}[\![p']\!]n^*s^*$$

where the first s^* denotes a list which is the subject of the transcription and the second denotes the same list in its role as the representation of a value environment.

We can use this identity to replace the $\mathcal{T}$ in the definition of $\mathcal{T}'[\![(p\,...)]\!]$ by $\mathcal{T}'$. For clarity, we assume for the moment that the pattern p is controllable in level-environment ρ_1. We may now derive the new definition:

$$\mathcal{T}'[\![(p\,...)]\!]\rho_1\rho_2 s$$
$$= \mathcal{T}[\![(p\,...)]\!]\rho_1(\lambda i.\rho_2 is)$$
$$= map\,(\mathcal{T}[\![p]\!])\,(decompose\,(\lambda i \in fv(p).\langle\rho_1 i, \rho_2 is\rangle))$$
$$= \text{let } i^* = fv(p);\ n^* = [\rho_1 i \mid i \in i^*];$$
$$\quad m^* = decompose\text{-}levels\,n^* \text{ in}$$
$$\quad \lambda s.map\,(\mathcal{T}[\![p]\!]m^*)(decompose\text{-}values\,n^*\,[\rho_2 is \mid i \in i^*])$$
$$= \text{let } i^* = fv(p);\ n^* = [\rho_1 i \mid i \in i^*];$$
$$\quad m^* = decompose\text{-}levels\,n^*;\ \rho^* = \mathcal{D}_2'[\![i^*]\!] \text{ in}$$
$$\quad \lambda s.map\,(\mathcal{T}'[\![p]\!]m^*\rho^*)$$
$$\qquad\qquad (decompose\text{-}values\,n^*\,[\rho_2 is \mid i \in i^*])$$

14

This version of the definition allows us to write $\mathcal{T}'$ as a pure structural recursion, so that we can transcribe it into a concrete representation as well:

$$\mathcal{T}''[\![(p \ ...)]\!]\rho_1\rho_2$$
$$= \text{let } i^* = fv(p); \ n^* = [\rho_1 i \mid i \in i^*];$$
$$m^* = decompose\text{-}levels \, n^*; \ \rho^* = \mathcal{D}'_2[\![i^*]\!] \text{ in}$$
$$\texttt{(map (lambda (s)} \ \mathcal{T}'[\![p]\!]m^*\rho^*\texttt{)}$$
$$\texttt{(decompose-values (quote} \ n^*\texttt{)}$$
$$\texttt{(list} \ \rho_2 i_0 \ \cdots \ \rho_2 i_k\texttt{)))}$$

For the last line, recall that ρ_2 contains concrete selectors, which, when evaluated with s bound to s, will return the value of $\rho_2 is$. Hence, if $i^* = [i_0, \cdots, i_k]$, then the `list` expression will evaluate correctly to $[\rho_2 is \mid i \in i^*]$.

This version of the target code includes an explicit call to `decompose-values`. It is possible to apply the same methods to *decompose-values*, using staging to take advantage of the fact that its first argument is known. The resulting system generates target code that includes a local `letrec` loop in place of the `decompose-values`, and in which the run-time level tests are eliminated. We leave this development as an exercise for the reader.

6. Results

The derivation, including false starts and debugging of the resulting code, took well under one man-week. The production compiler, by contrast, embodies several man-months of work.

How good is the resulting compiler? It seems to produce code which is comparable with the production version. For `let`, it produces the following code for the transcription:

```
(lambda (s)
  (cons
   (cons 'lambda
         (cons (map (lambda (s) (car s))
                    (car (cdr s)))
               (cdr (cdr s))))
   (map (lambda (s) (car (cdr s)))
        (car (cdr s)))))
```

which is clearly equivalent (given a reasonably optimizing compiler) to the production version code given above. It also produces comparable code for the tests. In view of this performance, we regard the derivation as a success.

7. Conclusions

We have presented a "macro-by-example" facility for Lisp-like languages. The facility allows the user to specify syntactic extensions in a natural, non-procedural

manner. These specifications are expanded into transformations which automatically perform pattern-matching, error-checking, and mapping. We have given a formal semantics for the specification language and have shown how the semantics can be converted into a compiler by the use of staging and suitable choices of representations for the semantic functions.

Dan Friedman originally suggested the idea of a new macro declaration tool. We gratefully acknowledge his contributions to this work. We also thank Matthias Felleisen for helping implement the compiler and proof-read this paper.

References

[Dybvig 87]

Dybvig, K. *The Scheme Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Felleisen 85]

Felleisen, M. "Transliterating Prolog into Scheme," Indiana University Computer Science Department Technical Report No. 182, October, 1985.

[Foderaro, Sklower, & Layer 83]

Foderaro, J.K., Sklower, K.L., and Layer, K. *The Franz Lisp Manual,* June, 1983.

[Felleisen & Friedman 86]

Felleisen, M., and Friedman, D.P., "A Closer Look at Export and Import Statements," *Computer Languages 11* (1986), 29–37.

[Friedman, Haynes, & Wand 86]

Friedman, D.P., Haynes, C.T., and Wand, M. "Obtaining Coroutines with Continuations," *J. of Computer Languages 11,* No. 3/4 (1986), 143–153

[Jorring & Sherlis 86]

Jorring, U., and Sherlis, W.L. "Compilers and Staging Transformations," *Conf. Rec. 13th Annual ACM Symposium on Principles of Programming Languages* (1986), 86–96.

[Kohlbecker 86]

Kohlbecker, E., *Syntactic Extensions in the Programming Language Lisp*, PhD dissertation, Indiana University, August, 1986.

[Kohlbecker, *et al.* 86]

Kohlbecker, E., Friedman, D.P., Felleisen, M., and Duba, B. "Hygienic Macro

Expansion," *Proceedings 1986 ACM Conference on Lisp and Functional Programming*, 151–161.

[Rees, Clinger, *et al.* 86]

Rees, J., and Clinger, W., eds. "Revised[3] Report on the Algorithmic Language Scheme," *SIGPLAN Notices 21,* 12 (December, 1986), 37–79

[Steele & Sussman 78]

Steele, G.L. and Sussman, G.J. "The Revised Report on SCHEME," Mass. Inst. of Tech. Artif. Intell. Memo No. 452, Cambridge, MA (January, 1978).

[Wand 82]

Wand, M. "Deriving Target Code as a Representation of Continuation Semantics," *ACM Trans. on Prog. Lang. and Systems 4*, 3 (July, 1982) 496–517.

[Wand 84]

Wand, M. "A Semantic Prototyping System," *Proc. ACM SIGPLAN '84 Compiler Construction Conference* (1984), 213–221.

[Wand 85]

Wand, M. "The Semantics of Backtracking," Brandeis University Computer Science Department Colloquium, January, 1985 (unpublished).