Meta-circular evaluation/compilation and first-class macros

Matthew Might
University of Utah
matt.might.net
www.ucombinator.org

Topics

- Architecture(s) of meta-circular evaluator
- Converting evaluators into compilers
- How to implement first-class macros

Warning

Trailing parentheses in examples may not be accurate.

The eval/apply model

- `eval  : exp   env                => value`
- `apply : value list[value] => value`

Expressions

```
<exp> ::= <number>
        |  <symbol>
        |  <boolean>
        |  <string>
        |  <character>
        |  <symbol>
        |  <primitive>
        |  (quote <s-exp>)
        |  (if <exp> <exp> <exp>)
        |  (lambda (<var> ...) <exp>)
        |  (letrec ((<var> <exp>) ...) <exp>)
        |  (<exp>1 ... <exp>n)
```

Options for env

- Direct encoding
- Structured encoding

Direct encoding

```
env = symbol => value
```

Direct encoding

```
env = symbol boolean value => value
```

Structured encoding

```
env = alist[symbol,value]
```

Structured encoding

```
env = balanced-map[symbol,value]
```

Options for `value`

- Direct encoding
- Wrapped encoding
- Mixture of both

Direct encoding

```
value = s-exp
```

S-Expressions

```
<s-exp> ::= (<s-exp>₁ ... <s-exp>ₙ)
          |  (<s-exp>₁ ... <s-exp>ₙ . <s-exp>)
          |  #(<s-exp>₁ ... <s-exp>ₙ)
          |  #t | #f
          |  <number>
          |  <string>
          |  <character>
          |  <symbol>
```

S-Expressions

```
<s-exp> ::= (<s-exp>₁ ... <s-exp>ₙ)
          |  (<s-exp>₁ ... <s-exp>ₙ . <s-exp>)
          |  #(<s-exp>₁ ... <s-exp>ₙ)
          |  #t | #f
          |  <number>
          |  <string>
          |  <character>
          |  <symbol>
          |  <procedure>
          |  <struct>
```

Wrapped encoding

```
<value> ::= (nil)
          |  (number    <number>)
          |  (boolean   <boolean>)
          |  (string    <string>)
          |  (symbol    <symbol>)
          |  (character <char>)
          |  (pair      <value> <value>)
          |  (vector    <value>₁ ... <value>ₙ)
          |  (primitive <symbol>)
          |  (closure   <exp> <env>)
          |  (struct    <symbol>
                       (<symbol> <value>) ...)
```

Mixed encoding

```
<value> ::= '()
          |  <number>
          |  <boolean>
          |  <string>
          |  <symbol>
          |  <char>
          |  (pair      <value> <value>)
          |  (vector    <value>₁ ... <value>ₙ)
          |  (primitive <symbol>)
          |  (closure   <exp> <env>)
          |  (struct    <symbol>
                       (<symbol> <value>) ...)
```

Macros & wrapped values

```
extract : wrapped-value => s-exp
```

eval

```
(define (eval exp env)
 (cond
  ((exp-type? exp)  (eval-exp-type exp env))
  ...))
```

eval

```
(define (eval exp env)
 (cond
  ((number? exp)  (eval-self    exp))
  ((string? exp)  (eval-self    exp))
  ...
  ((lambda? exp)  (eval-lambda exp env))
  ((app? exp)     (eval-app    exp env))))
```

eval-self (direct)

```
(define (eval-self exp) exp)
```

eval-self (wrapped)

```
(define (eval-self exp)
 (cond
  ((number? exp) (list ‘number exp))
  ((string? exp) (list ‘string exp))
  ...))
```

eval-lambda (wrapped)

```
(define (eval-lambda exp env)
  (list ‘closure exp env))
```

eval-lambda (direct)

```
(define (eval-lambda exp env)
 (let ((formals (cadr exp))
       (body    (caddr exp)))
  (if (symbol? formals)
   (lambda arg-values
    (eval body (env-extend formals arg-values)))
   (lambda arg-values
    (eval body (env-extend* formals arg-values))))))
```

eval-app (wrapped)

```
(define (eval-app exp env)
 (let ((f    (eval  (app->f    exp) env))
       (vals (eval* (app->args exp) env)))
  (cond
   ((primitive? f)   (apply-prim f vals)
   (else             (apply-proc f vals)))))
```

apply-proc

```
(define (apply-proc f vals)
  (let* ((lam     (cadr  f))
         (env     (caddr f))
         (formals (cadr  lam))
         (body    (caddr lam)))
    (if (symbol? formals)
     (eval body (env-extend  formals vals))
     (eval body (env-extend* formals vals)))))
```

eval-app (direct)

```
(define (eval-app exp env)
  (let ((f    (eval  (app->f    exp) env))
        (vals (eval* (app->args exp) env)))
    (apply f vals)))
```

From evaluator to compiler

Evaluator *v.* compiler

```
compile : exp      => (env => value)
eval    : exp env => value
```

Equivalence

```
(eval exp env) = ((compile exp) env)
(compile exp)  = (lambda (env) (eval exp env))
```

compile

```
(define (compile exp)
 (lambda (env)
  (eval exp env)))
```

compile

```
(define (compile exp)
 (lambda (env)
  (cond
   ((number? exp)  (eval-self   exp))
   ((string? exp)  (eval-self   exp))
   ...
   ((lambda? exp)  (eval-lambda exp env))
   ((app? exp)     (eval-app    exp env)))))
```

compile

```
(define (compile exp)
 (cond
  ((number? exp)
   (lambda (env) (eval-self   exp)))
 ((string? exp)
  (lambda (env) (eval-self   exp)))
 ...
 ((lambda? exp)
  (lambda (env) (eval-lambda exp env)))
 ((app? exp)
  (lambda (env) (eval-app    exp env)))))
```

compile

```
(define (compile exp)
 (cond
  ((number? exp) (compile-self   exp))
  ((string? exp) (compile-self   exp))
  ...
  ((lambda? exp) (compile-lambda exp))
  ((app? exp)    (compile-app    exp))))
```

compile-self (direct)

```
((compile-self exp) env) = (eval-self exp)
```

compile-self (direct)

```
(define (compile-self exp)
 (lambda (env)
  (eval-self exp)))
```

compile-self (direct)

```
(define (compile-self exp)
 (lambda (env) exp))
```

compile-lambda (direct)

```
((compile-lambda exp) env) = (eval-lambda exp env)
```

compile-lambda (direct)

```
(define (compile-lambda exp)
 (lambda (env)
  (eval-lambda exp env)))
```

compile-lambda (direct)

```
(define (compile-lambda exp)
 (lambda (env)
  (let ((formals (cadr exp))
        (body    (caddr exp)))
   (if (symbol? formals)
    (lambda arg-values
     (eval body (env-extend formals arg-values)))
    (lambda arg-values
     (eval body (env-extend* formals arg-values)))))))
```

compile-lambda (direct)

```
(define (compile-lambda exp)
 (lambda (env)
  (let* ((formals (cadr exp))
         (body    (caddr exp))
         (exe     (compile body)))
   (if (symbol? formals)
    (lambda arg-values
     (exe (env-extend formals arg-values)))
    (lambda arg-values
     (exe (env-extend* formals arg-values)))))))
```

compile-lambda (direct)

```
(define (compile-lambda exp)
 (let* ((formals (cadr exp))
        (body    (caddr exp))
        (exe     (compile body))
  (if (symbol? formals)
   (lambda (env)
    (lambda arg-values
     (exe (env-extend formals arg-values))))
   (lambda (env)
    (lambda arg-values
     (exe (env-extend* formals arg-values)))))))
```

compile-app (direct)

```
((compile-app exp) env) = (eval-app exp env)
```

compile-app (direct)

```
(define (compile-app exp)
 (lambda (env)
  (eval-app exp env)))
```

compile-app (direct)

```
(define (compile-app exp)
 (lambda (env)
  (let ((f    (eval  (app->f    exp) env))
        (vals (eval* (app->args exp) env)))
   (apply f vals))))
```

compile-app (direct)

```
(define (compile-app exp)
 (lambda (env)
  (let ((exe  (compile (app->f exp)))
        (exes (map compile (app->args exp))))
   (let ((f    (exe env))
         (vals (map exes env)))
    (apply f vals)))))
```

compile-app (direct)

```
(define (compile-app exp)
 (let ((exe  (compile (app->f exp)))
       (exes (map compile (app->args exp))))
  (lambda (env)
   (let ((f    (exe env))
         (vals (map exes env)))
    (apply f vals)))))
```

Can we do better?

Yes.

First-class macros

Expressions

```
<exp> ::= <number>
        ...
      |  <boolean>
      |  (<exp> <exp> ...)
      |  (<keyword> <s-exp> ...)
```

Wrapped encoding

```
<value> ::= (nil)
          |  (number              <number>)
          |  (boolean             <boolean>)
          |  (string              <string>)
          |  (symbol              <symbol>)
          |  (character           <char>)
          |  (pair                <value> <value>)
          |  (vector              <value>₁ ... <value>ₙ)
          |  (primitive           <symbol>)
          |  (closure             <exp> <env>)
          |  (struct              <symbol>
                                   (<symbol> <value>) ...)
          |  (macro               <procedure>)
          |  (syntax-primitive    <procedure>)
```

eval

```
(define (eval exp env)
 (cond
  ((number? exp)  (eval-self    exp))
  ((string? exp)  (eval-self    exp))
  ...
  ((lambda? exp)  (eval-lambda exp env))
  ((app? exp)     (eval-app    exp env))))
```

eval

```
(define (eval exp env)
 (cond
  ((number? exp)  (eval-self   exp))
  ((string? exp)  (eval-self   exp))
  ...

  ((app? exp)     (eval-app    exp env))))
```

eval-app (wrapped)

```
(define (eval-app exp env)
 (let ((f    (eval  (app->f    exp) env))
       (vals (eval* (app->args exp) env)))
  (cond
   ((primitive? f)   (apply-prim f vals)
   (else             (apply-proc f vals)))))
```

eval-app (wrapped)

```
(define (eval-app exp env)
 (let ((f    (eval  (app->f    exp) env)))
  (cond
   ((macro? f)
    (eval (apply-macro f exp) env))
   ((syntax-primitive? f)
    ((prim->eval f) exp env))
   (else
    (apply-proc
         f
         (eval* (app->args exp) env)))))
```

Initial environment (1)

```
[lambda => (syntax-primitive ...),
 set!   => (syntax-primitive ...),
 letrec => (syntax-primitive ...),
 let    => (syntax-primitive ...),
 ...]
```

Initial environment (II)

```
[lambda => (syntax-primitive ...),
 set!   => (syntax-primitive ...),
 letrec => (macro ...),
 let    => (macro ...),
 ...]
```

Questions?