Scheme Continuations and Macros

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

October 6, 2021

Table of Contents

Table of Contents

Continuations and macros form some of Scheme's more advanced features. While they are not strictly necessary to be productive in Scheme, they are very powerful features.

Table of Contents

Continuations

The R6RS standard says, "Whenever a Scheme expression is evaluated there is a continuation wanting the result of the expression. The continuation represents an entire (default) future for the computation."
Whenever the interpreter executes a subexpression, there is a continuation waiting on the result of that subexpression.
For example, if I execute the expression 5, there is a continuation waiting on 5 to evaluate.

Continuation Example

Given the expression

```
(+ 1 3)
```

The continuation of 3 in the above expression is its parent expression, which will add 1 to it.

Continuation Definition

Definition (Continuation)

The continuation of an expression E is the expression that wants the evaluated result of E.

Continuations

- Usually we don't need to think about continuations when we code in Scheme; we normally don't think too deeply about expressions wanting the results of subexpressions.

Continuations

- Usually we don't need to think about continuations when we code in Scheme; we normally don't think too deeply about expressions wanting the results of subexpressions.
- However, there are occasions where we want to be able to deal with continuations manually.
  - For example, when handling errors we may be keenly interested in the result of the expression that caused the error. Thus, the continuation of the error-prone expression E is of interest.

call-with-current-continuation

- Often abbreviated `call/cc` in many Scheme implementations (but sadly not DrRacket), though we can abbreviate it ourselves using `(define call/cc call-with-current-continuation)`.
- Syntax: `(call/cc fn)`
- `call/cc` calls `fn`, a function that has one parameter, with the argument being an *escape procedure*.
- According to the R6RS standard, "The escape procedure can then be called with an argument that becomes the result of the call to `call/cc`. That is, the escape procedure abandons its own continuation and reinstates the continuation of the call to `call/cc`."

Example of call-with-current-configuration

```
(let ((fn (lambda (escape)
            (+ 2 (escape 3)))))
  (+ 1 (call/cc fn)))
```

Example of `call-with-current-configuration`

```
(let ((fn (lambda (escape)
            (+ 2 (escape 3)))))
  (+ 1 (call/cc fn)))
```

The above code evaluates to 4, because `(escape 3)` is performing $1 + 3$, and because the continuation of `call/cc fn)` is $+1$.

Notes about `call-with-current-configuration`

- `fn` in `(call/cc fn)` always takes one argument: the escape procedure.
- The escape procedure's parameters are the same number as the continuation of the call to `call/cc`.
- The escape procedure is a closure that can be called at any time. It can be passed along like any other value, and it can be stored like any other value.

More about Continuations

Because the escape procedure can be called at any time by any expression that has access to it, `call/cc` can be used to implement a wide variety of custom control-flow operations, even `goto`-like arbitrary jumps! Thus, it is very important to treat `call/cc` with care.

Table of Contents

Motivation for Macros

We can extend Lisp by defining functions that are based on existing functions:

```
(define (sqrt n)
  (expt n 0.5))

(sqrt 2)
```

Limitations of Function Definitions

However, there are some constructs we would like to provide that would be difficult to express as functions.

Implementing `let`

Suppose we were implementing our own `let` function. Below is a reasonable attempt:

```
(define (my-let bindings body)
  (if (empty? bindings)
      body
      ((lambda (first (first bindings))
         (my-let (rest bindings) body))
       (second (first bindings)))))
```

Implementing `let`

Suppose we were implementing our own `let` function. Below is a reasonable attempt:

```
(define (my-let bindings body)
  (if (empty? bindings)
      body
      ((lambda (first (first bindings))
         (my-let (rest bindings) body))
       (second (first bindings)))))
```

What is wrong with this approach?

The problem is that all function arguments must be evaluated unless they are quoted.

Implementing `let`

We could sidestep the problem by requiring the quoting of arguments we don't want evaluated and then using the `eval` function inside of `my-let` in order to provide more fine-grained control over evaluation:

```
(my-let ((’x 5)
         (’y 10))
  ’(+ x y))
```

Implementing `let`

We could sidestep the problem by requiring the quoting of arguments we don't want evaluated and then using the `eval` function inside of `my-let` in order to provide more fine-grained control over evaluation:

```
(my-let ((’x 5)
         (’y 10))
  ’(+ x y))
```

However, it would be inconvenient for programmers if they were required to quote so many arguments like this.

Solution to Implementing `let`: Macros

The solution is to use macros, which will give us finer control over how a Scheme expression is evaluated without resorting to quoting. Macros can be thought of as a way to substitute terms on an AST.

Solution to Implementing `let`

```
; Solution is from Veit Heller's blog at
; blog.veitheller.de/Scheme_Macros_III:_Defining_let.html
(define-syntax my-let
  (syntax-rules ()
    ((my-let ((var val) ...) body ...)
     ((lambda (var ...) body ...) val ...))))
```

This works in R5RS Scheme. The ellipses (`...`) are used to express variable length arguments and is part of the pattern language that `syntax-rules` supports. Please see the R5RS Scheme specification for more details about this pattern language.

Another Example of a Macro: `my-when`

Note that this also works in R5RS Scheme.

```
(define-syntax my-when
  (syntax-rules ()
    ((my-when test branch ...) (if test (begin branch ...)))
```

Another Example of a Macro: swap

```
(define-syntax swap
  (syntax-rules ()
    ((swap x y) (begin (let ((tmp x))
                         (set! x y)
                         (set! y tmp))))))
```

A Basic Pattern for Implementing Macros

```
(define-syntax MACRO-NAME
  (syntax-rules ()
    ((MACRO-NAME ARG1 ARG2 ...) EXPR)))
```

Hygienic Macros

- A hygienic macro is one where the AST substitutions are performed in such a way to avoid variable capture, the phenomenon in which a macro tramples over variables defined in the environment.
- All macros defined by `syntax-rules` are hygienic per the R5RS Scheme standard.
- However, it is possible to create non-hygienic macros in Scheme using quasiquotation.
- For examination purposes, I'm not going to test you on quasiquotation syntax, but please be expected to be able to write hygienic macros and to know the difference between hygienic and unhygienic macros.

A word of caution: each Scheme implementation has its own mechanism for defining macros. While the R5RS standard has `define-syntax` that uses `syntax-rules` to specify hygenic macros, different implementations of Scheme offer their own implementation-dependent ways of defining macros. For example, in the textbook *Teach Yourself Scheme in Fixnum Days*, the examples all use `define-macro`, which is used in some Scheme implementations, but (unfortunately) not DrRacket.

Macros as Inline Functions

Another advantage of macros is that because they serve as syntactic substitutions, they can be used as alternatives to defining functions, since a macro substitution is not the same thing as a function call.

Summary of Macros

- Macros are necessary for implementing certain language features in Scheme without excessive quoting.
- Macros can be used to implement inline functions.
- There are many different mechanisms for implementing macros in Scheme based on the implementation, but in R5RS Scheme the standard way of implementing macros is through `define-syntax`. All macros defined using `syntax-rules` are hygienic.
- Hygienic macros do not trample over bindings, while unhygienic macros potentially could.