CS450

Structure of Higher Level Languages

Lecture 24: Dynamic scoping/generic methods/macros

Tiago Cogumbreiro

Today we will learn about...

- Dynamic scoping in Racket
- Generic methods vs pattern-match
- Macros

Dynamic scoping in Racket

`parameterize`

Static versus dynamic scoping

Static Scoping

> **Static binding:** variables are captured at creation time

```
(define x 1)

(define (f y) (+ y x))

(define (g)
  (define x 20)
  (define y 3)
  (f (+ x y)))

(check-equal? (g) (+ 23 1))
```

Dynamic Scoping

> **Dynamic binding:** variables depends on the calling context

```
(define x 1)

(define (f y) (+ y x))

(define (g)
  (define x 20)
  (define y 3)
  (f (+ x y)))
; NOT VALID RACKET CODE
(check-equal? (g) (+ 23 20))
```

Why dynamic scoping?

1. A controlled way to represent global variables
2. A technique to make code testable

Dynamic scoping example

Dynamic scoping In Racket

```
(define x (make-parameter 1))
(define (f y) (+ y (x)))

(define (g)
  (parameterize ([x 20])
    (define y 3)
    (f (+ (x) y))))

(check-equal? (g) (+ 23 20))
```

Pseudo-Racket dynamic scoping

```
(define x 1)

(define (f y) (+ y x))

(define (g)
  (define x 20)
  (define y 3)
  (f (+ x y)))
; NOT VALID RACKET CODE
(check-equal? (g) (+ 23 20))
```

- Function `make-parameter` returns a reference to a dynamically scoped memory-cell
- Calling a parameter without parameter returns the contents of the memory-cell
- Use `parameterize` to overwrite the memory-cell

Dynamic binding

Globals

Dynamic binding: controlled globals

We can define different globals in different contexts.

```
(define buff (open-output-string))
(parameterize ([current-output-port buff])
  ; In this context, the standard output is a string buffer.
  (display "hello world!"))
(check-equal? (get-output-string buff) "hello world!")
```

Racket uses parameters to allow extending the behavior of many features:

- command line parameters
- standard output stream (known as a port)
- formatting options (eg, default implementation to print structures)

Dynamic binding

Testing

Dynamic binding: making code testable

> Consider an excerpt of Homework 5. We would like to be able to test each function independently. How?

```
(define (s:eval-exp mem env exp)
  (define (on-app mem env exp)
    ;; ...
    ;; Eb \Downarrow Eb vb
    (s:eval-term mem3 Eb (s:lambda-body lam)))
  (cond
    ;; ...
    [(s:apply? exp) (on-app mem env exp)]

(define (s:eval-term mem env term)
  (cond
    ; ...
    [else (s:eval-exp mem env term)]))
```

Dynamic binding: making code testable

- In Homework 4, we added a function parameter to test `r:eval` independently from `r:subst`.
- This extra function parameter was confusing to some students.
- This choice made the function interface more verbose than needed.
- More arguments, more chance of mistakes! Do we call `subst` or `s:subst`?

How can we use dynamic binding

to improve the testing design of `r:eval`?

UMass Boston

Dynamic binding: making code testable

- Create a parameter per global function that you want to make testable
- Internal calls should target the ***parameter*** and not the global variable

Before

```
(define (r:eval subst exp)
  (cond
    [...
     (define eb' (subst eb x va))
     ...]))
```

Dynamic binding: making code testable

- Create a parameter per global function that you want to make testable
- Internal calls should target the ***parameter*** and not the global variable

Before

```
(define (r:eval subst exp)
  (cond
    [...
     (define eb' (subst eb x va))
     ...]))
```

After

```
(define r:subst-impl
    (make-parameter r:subst))

(define (r:eval exp)
  (cond
    [...
     (define eb' ((r:subst-impl) eb x va))
     ...]))
```

UMass Boston

Dynamic binding: making code testable

> Consider an excerpt of Homework 8. We would like to be able to test each function independently. How?

```
(define (s:eval-exp mem env exp)
  (define (on-app mem env exp)
    ; ...
    ((s:eval-term-impl) mem3 Eb (s:lambda-body lam)))
  (cond ; ...
    [(s:apply? exp) (on-app mem env exp)]
(define s:eval-exp-impl (make-parameters s:eval-exp))

(define (s:eval-term mem env term)
  (cond ; ...
    [else ((s:eval-exp-impl) mem env term)]))
(define s:eval-term-impl (make-parameters s:eval-term))
```

Dynamic binding: making code testable

- Usage example:

```
(parameterize ([s:eval-expr-impl (lambda (mem env expr) (s:number 10))])
  ; Now x is evaluated to (s:number 10) and y evaluates to (s:number 10)
  (eval-term? '[x y] 10))
```

We can test eval-term without implementing eval-exp!

This testing technique is known as ***mocking***.

Generic methods versus match

Example: serialization

Let us implement a serialization function

```
#lang racket
(require rackunit)
(require racket/generic)
(provide (all-defined-out))
;; Values
(define (r:value? v) (r:number? v))
(struct r:number (value) #:transparent)
;; Expressions
(define (r:expression? e) (or (r:value? e) (r:variable? e) (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)
```

Specification

```
(check-equal? (r:quote (r:apply (r:variable '+) (list (r:number 1) (r:number 2))))  '(+ 1 2
```

Implementing `r:quote` with `match`

File: `example1.rkt`

> Copy/paste the AST and implement `r:quote`.

Solution

```
(define (r:quote exp)
```

Implementing `r:quote` with `match`

File: `example1.rkt`

> Copy/paste the AST and implement `r:quote`.

Solution

```
(define (r:quote exp)

  (match exp
    [(r:number n) n]
    [(r:variable x) x]
    [(r:apply ef ea) (cons (r:quote ef) (map r:quote ea))]))
```

Revisiting racket/generic

File: example2.rkt

> We can use racket/generic to represent abstract interfaces that are satisfied dynamically by the argument. A generic interface may have one or more functions.

```
(define-generics quotable
   (r:quote quotable))

(define (r:value? v) (r:number? v))
(struct r:number (value) #:transparent
  #:methods gen:quotable
  [(define (r:quote n) (r:number-value n))])

(check-equal? (r:quote (r:number 10)) 10)
```

`racket/generic` and recursive calls

> When a method needs to do a ***generic*** recursive call, we need to access the "***main***" generic method, and not the current method. To do so, we need to use `define/generic` to access the main generic method.

```
(struct r:apply (func args) #:transparent
  #:methods gen:quotable
  [
    (define/generic rec-quote r:quote)
    (define (r:quote app)
      (cons (rec-quote (r:apply-func app))
            (map rec-quote (r:apply-args app))))])
```

In contrast with

```
    [(r:apply ef ea) (cons (r:quote ef) (map r:quote ea))]))
```

Generic interface summary

`define-generics` defines an interface

- A generic interface has a name, in this example it is `fruit`
- We specify which methods are generic and provide the list of formal parameters. Exactly one parameter must have the name of the interface.

```
(define-generics fruit
  (pick x fruit)
  (pluck fruit x))
; (foo fruit fruit) <-- incorrect because fruit shows up more than once
; (bar x y)         <-- incorrect because fruit does not show up
```

More

- `define/generic` accesses the generic method
- We can check if a value is of a given interface with `(fruit? x)`

UMass Boston

Introducing booleans

Introducing booleans

```
;; Values
(define (r:value? v) (or (r:number? v) (r:bool? v)))
(struct r:number (value) #:transparent)
(struct r:bool (value) #:transparent)

(check-equal? (r:quote (r:apply (r:variable 'and) (list (r:bool #t) (r:bool #f))))
  '(and #t #f))
```

What is the impact of adding a new kind of AST node?

Match version

File: `example1-v2.rkt`

> We must go through each function that has a `match` and add a branch to handle our new AST node.

```
(define (r:quote exp)
  (match exp
    [(r:number n) n]
    [(r:variable x) x]
    [(r:bool b) b]
    [(r:apply ef ea) (cons (r:quote ef) (map r:quote ea))]))
```

Generic version

File: `example2-v2.rkt`

- We must update our AST to implement the generic interface.

```
(struct r:bool (value) #:transparent
  #:methods gen:quotable
  [(define (r:quote b) (r:bool-val b))])
```

Generic is open-ended

File: `example3.rkt`

> A benefit of `generic` is that it is dynamically extensible. With `match` you may need to change a 3rd-party code.

```
#lang racket
(require rackunit)
(require "example2.rkt")

(struct r:bool (val) #:super struct:r:value
  #:methods gen:quotable
  [(define (r:quote b) (r:bool-val b))])

(check-equal? (r:quote (r:apply (r:variable 'and) (list (r:bool #t) (r:bool #f))))
  '(and #t #f))
```

Contrasting `match` with `generic`

> What are the main differences between `match` and `generic`?

Code impact in adding a new kind of node

Contrasting `match` with `generic`

> What are the main differences between `match` and `generic`?

Code impact in adding a new kind of node

Match

- Code is centralized in a function

Dispatch

- Code is split across structs

Extension points

UMass Boston

Contrasting `match` with `generic`

> What are the main differences between `match` and `generic`?

Code impact in adding a new kind of node

Match

- Code is centralized in a function

Dispatch

- Code is split across structs

Extension points

Match

- Not possible

Dispatch

- Any code may add a branch

Quiz: `match` versus `dispatch`

Q1: Which of the code is centralized?

Q2: Each of which allows for extension points?

Implementing generic

Implementing `generic`

1. Declare a generic function

```
(define-generic quotable (r:quote quotable))
```

2. Register an instance of said function

```
  #:methods gen:quotable
  [(define (r:quote b) (r:bool-val b))])
```

3. Call a generic function

```
(r:apply (r:variable 'and) (list (r:bool #t) (r:bool #f)))
```

What is implicit here?

1. Declare a generic function

```
(define-generic quotable (r:quote quotable))
```

What is implicit here?

1. Declare a generic function

```
(define-generic quotable (r:quote quotable))
```

Nothing implicit.

2. Register an instance of said function

```
  #:methods gen:quotable
  [(define (r:quote b) (r:bool-val b))])
```

What is implicit here?

1. Declare a generic function

```
(define-generic quotable (r:quote quotable))
```

Nothing implicit.

2. Register an instance of said function

```
  #:methods gen:quotable
  [(define (r:quote b) (r:bool-val b))])
```

The **registry** of `quotable` is implicit!

3. Call a generic function

```
(r:apply (r:variable 'and) (list (r:bool #t) (r:bool #f)))
```

What is implicit here?

1. Declare a generic function

```
(define-generic quotable (r:quote quotable))
```

Nothing implicit.

2. Register an instance of said function

```
  #:methods gen:quotable
  [(define (r:quote b) (r:bool-val b))])
```

The **registry** of `quotable` is implicit!

3. Call a generic function

```
(r:apply (r:variable 'and) (list (r:bool #t) (r:bool #f)))
```

The **registry** of `quotable` is implicit!

What is the registry?

What is the registry?

A map from types to functions (instances)

1. Declare a generic function

> Declaring a generic function should return a registry. We will assume only **one** generic function. We must allow the selection of which argument to dispatch on.

2. Register an instance of said function

What is the registry?

A map from types to functions (instances)

1. Declare a generic function

> Declaring a generic function should return a registry. We will assume only **one** generic function. We must allow the selection of which argument to dispatch on.

2. Register an instance of said function

> Registering an instance should add one entry to the registry. It should register the type as the key.

3. Call a generic function

> Calling a generic function should lookup the registry for the right instance according to the type.

1. Declaring a generic function

- Which argument is being dispatched on?
- How many arguments does the function have?
- What is an instance?

1. Declaring a generic function

- Which argument is being dispatched on?
- How many arguments does the function have?
- What is an instance?
  - The keys are predicates
  - The values are functions as values

1. Declaring a generic function

- Which argument is being dispatched on?
- How many arguments does the function have?
- What is an instance?
  - The keys are predicates
  - The values are functions as values

```
(struct generic (index instances))
(define (make-generic index)
  (generic index (list)))
(struct instance (type? func))
```

Example

```
(define g
  (generic 0 ; dispatch on the first argument
    (list (instance r:bool? (lambda (b) (r:bool-val b))))))
```

Original

```
#:methods gen:quotable
[(define (r:quote b)
   (r:bool-val b))])
```

2. Registering an instance

Registration takes a predicate and a function, and updates a generic.

```
(define (generic-register gen prec? func)
```

2. Registering an instance

Registration takes a predicate and a function, and updates a generic.

```
(define (generic-register gen prec? func)
  (generic
    (generic-index gen)
    (cons (instance prec? func) (generic-instances gen))))
```

3. Call a generic function

- We want to implement `(generic-apply gen . args)`

3. Call a generic function

> We want to implement `(generic-apply gen . args)`

1. Let the list of instances be `l`
2. Let the the index being dispatched be `n`
3. Load the `n`-th argument
4. Let the the instance that matches the `n`-th argument be `f`
5. Call `f` with arguments `args`

Implementing instance lookup

> Given a `generic` and a value, return the instance callback. Function `(memf f l)` finds an element using `f`; an element is found when `f` applied to the element returns a true value.

Implementing instance lookup

Given a `generic` and a value, return the instance callback. Function `(memf f l)` finds an element using `f`; an element is found when `f` applied to the element returns a true value.

```
(define (generic-lookup gen elem)
  (memf
    (lambda (inst) ((instance-type? inst) elem))
    (generic-instances gen)))
```

Implementing `generic-apply`

- We can load the `n`-th element of a list with function `(list-ref list index)`.

```
(define (generic-apply gen . args)
```

Implementing `generic-apply`

- We can load the `n`-th element of a list with function `(list-ref list index)`.

```
(define (generic-apply gen . args)

  (define elem (list-ref args (generic-index gen)))
  (apply (generic-lookup gen elem) args))
```

Example

```
(define g
  (generic 0 ; dispatch on the first argument
    (list (instance r:bool? (lambda (b) (r:bool-val b))))))
(check-true (generic-apply g (r:bool #t)))
```

Limitations

- Lookup is linear with the number of instances
- No error reporting:
  - Instance with 1 arguments, but we are dispatching on the 2nd argument
  - Do we want to enforce that all instances have the same number of arguments?

Today we will...

- Why macros are needed?
- Where are macros used?
- Safe versus unsafe macros
- The problems of using macros
- Macros in Racket
- Macros: side-effects
- Macros: controlling evaluating
- Macros: types in macros
- Macros: pattern matching

> Acknowledgment: Today's lecture is inspired by Professor Dan Grossman's wonderful lecture in CSE341 from the University of Washington.

Macro systems

What is a macro

> A macro is a technique to perform reusable source-code transformations with the objective to extend the language semantics.

- **Macro definition:** describes how the transformation occurs
- **Macro system:** the language used to describe transformations
- **Macro expansion:** the process of transforming the syntax according to some macro

> Macro expansion occurs before the program is run (and compiled).

Macros in Racket

> Macros in Racket are used as function calls, however evaluation does **not** proceed as it does with a function application.

Example 1

Expands a do-macro that accepts special keywords/symbols

```
(do x <- (push 10) (pop))
```

into

```
(bind (push 10) (lambda (x) (pop)))
```

Example 2

Omit some expressions of the macro

```
(comment-out (/ x 0) 10)
```

expands into

```
10
```

Example uses

Macros can vastly transform the Racket language

Macros can:

- encode infix notation
- encode alternate evaluation methods (such as lazy evaluation)
- generate boilerplate code (repetitive code)
- encode different programming models (succinct syntax for monads, OOP, etc)

Macros uses in practice

- Most Racket's language features are built with macros!
  Examples: `cond`, promises, OOP system, etc
- Automatic JSON/XML serialization in OCaml
- Boilerplate generation (bridges) from OCaml to JavaScript, and from Rust to GLib (C-based OOP runtime)

The perils of macros

The perils of macros

- **Unclear computational model:**
  How are the parameters evaluated? Does the macro produce side effects?
- **Limited composability:**
  Is the result of a macro a value? can it be passed around?
- **Stack-trace obfuscation:**
  The emitted code may generate a non-obvious stack trace, which hinders debugging.
- **Non-terminating compilation:**
  Most macros-systems are Turing complete, which means they may not terminate. They may slow down compilation times, a problem at scale.

Declare macros sparingly and with caution

Following we will learn...

- Manipulating syntactic elements (tokens, parentheses, scope)
- Defining macros
- Controlling expression evaluation
- Introduce macro ***hygiene***

Macros manipulate syntactic terms

- A macro system usually operates on the **concrete** syntax
- Recall our exercises on datums, a macro system operates at the datums level.
- In the concrete syntax, there will be some notion of a literal, an identifier, a sequence, a datum, maybe control-flow data structures
- Generally, a macro system does **not** operate at the lexical level
  For example, a macro system cannot declare a new parsing rule to recognize, say, binary number literals.

Macro expansion

How macro systems generate code?

> Does the macro system support structured data?

Unstructured expansion

> The C macro system operates at the textual level, there is no notion of structure, and simply allows for free-text transformation.

```
#define ADD(x,y) x+y
```

Expression `ADD(1, 2) * 3` expands to `1 + 2 * 3` and not to `(1 + 2) * 3`.

Structured expansion

> The Racket macro system operates at the concrete syntax level, so code transformations retain their structure.

```
(define-syntax-rule (ADD x y) (+ x y))
(check-equal? (* (ADD 1 2) 3) 9)
```

UMass Boston

C: The perils of unstructured macros

> "What is the worst real-world macros/pre-processor abuse you've ever come across?" Stack Overflow.

```c
int foo(state_t *state) {
    int a, b, rval;

    $
    if (state->thing == whatever) {
        $
        do_whatever(state);
    }
    // more code

    $
    return rval;
}
```

```c
#if DEBUG
#define $ log("%s %d", __FILE__, __LINE__);
#else
#define $
#endif
```

Source: Frank Szczerba

The infamous UNIX Bourne Shell

```
#define IF   if (
#define THEN ) {
#define ELSE } else {
#define ELIF } else if (
#define FI   ; }

VOID    free(ap)
    BLKPTR      ap;
{
    REG BLKPTR  p;

    IF (p=ap) ANDF p<bloktop
    THEN    Lcheat((--p)->word) &= ~BUSY;
    FI
}
```

> The source code of the UNIX Bourne shell (1970) used macros to make C code more similar to Algol 68. Source code available online: macros defined in `mac.h`, example program `blok.c`.

Source: Jim Ferrans

The Love/Hate Relationship with the C Preprocessor

The Love/Hate Relationship with the C Preprocessor: An Interview Study. Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. ECOOP, 2015.

Why use macros

- portability: support different operating systems with little change
- variability: removing parts of the library to reduce the binary code size

```
if (b_ffname != NULL
#ifdef FEAT_NETBEANS
  && netbeansReadFile
#endif
) {
  // code
}
```

```
mfp = open(mf_fname
#ifdef UNIX
, (mode_t)0600
#endif
#if defined(MSDOS)
, S_IREAD | S_IWRITE
#endif
);
```

```
#if defined(GUI_W32)
void msgNetbeansW32(
#else
void msgNetbeans(Xt client,
#endif
XtInputId *id) {
  // code
}
```

Code snippets from the Vim editor.

Macros in Racket

A macro example

- Use `define-syntax-rule` as you would use a `define`.

```
(define-syntax-rule (ADD x y)
  (+ x y))
(check-equal? (* (ADD 1 2) 3) 9)
```

Side effects

> keeping in mind that its contents are ***not*** evaluated. The contents of the macro are therefore **inlined**.

Example

```
(define-syntax-rule (SQR x)
  (* x x))
```

Beware of side-effects!

```
; Prints !!
(define (f) (display "!") 3)
(SQR (f))
```

Spec

```
(check-equal?
  (SQR (* 2 3))
  (* (* 2 3) (* 2 3))) ; expands x twice!
```

Solution

```
(define-syntax-rule (SQR x)
  ((lambda (new-x) (* new-x new-x))
   x))
; Or, use the let construct
(define-syntax-rule (SQR x)
  (let ([new-x x]) (* new-x new-x)))
```

Why would you want to control evaluation?

Controlling evaluation: example 1

> Macros allow us to control evaluation, which lets us delay evaluation. Here is an implementation of an `if` command.

```
(define-syntax-rule (IF cnd then-branch else-branch)
  (or (and cnd then-branch) else-branch))
; Sanity tests; in case of eager evaluation it should crash
(check-equal? (IF #t 1 (/ 1 0)) 1)
(check-equal? (IF #f (/ 1 0) 2) 2)
```

Controlling evaluation: example 2

> When creating a testing library, we may need to show the user which code is failing. We can quote a macro variable and print the datum.

```
(define-syntax-rule (assert x)
  (IF x (void) (error "Condition failed: " (quote x))))

(assert (and #f 10))
 ; Condition failed:  (and #f 10) [,bt for context]
```

UMass Boston

Controling evaluation: example 3

```
(define-syntax-rule (letin x v e)
  ((lambda (x) e) v))

(check-equal? (letin x (+ 10 50) x) 60)
```

Adding types to macros

Restricting what appears where

> The macro construct `define-simple-macro` allows restricting what ***kind*** of parameter is expected, which improves the error messages.

Version 1

```
(require syntax/parse/define)
(define-simple-macro (fn x body)
  (lambda (x) body))

(check-equal? ((fn x x) 10) 10)
; (fn 11 10)
; lambda: not an identifier, identifier wit
; default, or keyword
;   at: 11
;   in: (lambda (11) 10)
; [,bt for context]
```

Version 2

```
(require syntax/parse/define)
(define-simple-macro (fn x:id body:expr)
  (lambda (x) body))

(check-equal? ((fn x x) 10) 10)
; (fn 11 10)
; fn: expected identifier
;   at: 11
;   in: (fn 11 10)
; [,bt for context]
```

Introducing syntactic literals

```
(define-simple-macro (fn x (~literal ->) expr)
  (lambda (x) expr))

(check-equal? ((fn x -> x) 10) 10)
```

Pattern matching in macros

Revisiting the do notation

```
(define-syntax do
  (syntax-rules (<-) ; here we declare reserved syntactic  tokens
    ; Only one monadic-op, return it
    [(_ mexp) mexp]  ; alternatively, we could write (do mexp)
    ; A binding operation
    [(_ var <- mexp rest ...) (bind mexp (lambda (var) (do rest ...)))]
    ; No binding operator, just ignore the return value
    [(_ mexp rest ...) (bind mexp (lambda (_) (do rest ...)))]))
```