CS450

Structure of Higher Level Languages

Lecture 22: Pattern matching / dynamic dispatch

Tiago Cogumbreiro

Pattern matching

Pattern matching

Operation `match` can perform pattern matching on the given argument. Think of it as a `switch` statement on steroids.

Without

```
(define (r:eval-builtin sym)
  (cond [(equal? sym '+) +]
        [(equal? sym '*) *]
        [(equal? sym '-) -]
        [(equal? sym '/) /]
        [else #f]))
```

With `match`

```
(define (r:eval-builtin sym)
  (match sym
    ['+ +]
    ['* *]
    ['- -]
    ['/ /]
    [_ #f]))
```

The underscore operator `_` means any pattern.

No-match exception

> Operation `match` raises an exception when no pattern is matched, unlike `cond` that returns `#<void>`.

```
(match 1
  [10 #t]) ; Expecting 10, but given 1, so no match
 ; match: no matching clause for 1 [,bt for context]
```

Matching lists

With `cond`

```
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

With `match`

Matching lists

With `cond`

```
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

With `match`

```
(define (factorial n)
  (match n
    [0 1]
    [_ (* n (factorial (- n 1)))]))
```

Introducing define/match

> The define and match pattern is so common that there is a short-hand version. ***Notice the parenthesis!***

With define/match

```
(define/match (factorial n)
  [(0) 1]
  [(_) (* n (factorial (- n 1)))])
```

With match

```
(define (factorial n)
  (match n
    [0 1]
    [_ (* n (factorial (- n 1)))]))
```

With cond

```
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

List patterns

- Lists are so common that they deserve a special range of patterns

```
(define (f l)
  (match l
    [(list) #f]              ; Matches the empty list
    [(list 1 2) #t]          ; Matches a list with exactly 1 and 2
    [(list x y) (+ x y)]     ; Matches a list with any two elements
    [(list h t ...) t]))     ; Matches a nonempty list

(check-equal? (f (list)) ???)
(check-equal? (f (list 1) ???)
(check-equal? (f (list 1 2) ???)
(check-equal? (f (list 2 3) ???)
```

List patterns

- Lists are so common that they deserve a special range of patterns

```
(define (f l)
  (match l
    [(list) #f]
    [(list 1 2) #t]
    [(list x y) (+ x y)]
    [(list h t ...) t]))

(check-equal? (f (list)) #f)
(check-equal? (f (list 1) (list))
(check-equal? (f (list 1 2) #t)
(check-equal? (f (list 2 3) (+ 2 3))
```

Example map

With cond

```
(define (map f l)
  (cond [(empty? l) l]
        [else (cons (f (first l)) (map f (rest l)))]))
```

With match

Example `map`

With `cond`

```
(define (map f l)
  (cond [(empty? l) l]
        [else (cons (f (first l)) (map f (rest l)))]))
```

With `match`

```
(define (map f l)
  (match l
    [(list) l]
    [(list h t ...) (cons (f h) (map f t))]))
```

The `#:when` clause

With `match`

```
(define (member x l)
  (match l
    [(list) #f]
    [(list h _ ...) #:when (equal? x h) #t]
    [(list _ t ...) (member x t)]))
```

With `cond`

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

- Use the `#:match` clause to add a condition to the pattern

struct patterns

- Match also supports structs

```
(struct foo (bar baz))
(define (f x)
  (match x
    [(foo a b) (+ a b)]))
(check-equal? (f (foo 1 2)) 3)
```

Exercise r:eval-exp

With cond

```
(define (r:eval-exp exp)
  (cond
    ; 1. When evaluating a number, just return that number
    [(r:number? exp) (r:number-value exp)]
    ; 2. When evaluating an arithmetic symbol, return the respective arithmetic function
    [(r:variable? exp) (r:eval-builtin (r:variable-name exp))]
    ; 3. When evaluating a function call evaluate each expression and apply
    ;    the first expression to remaining ones
    [(r:apply? exp)
     ((r:eval-exp (r:apply-func exp))
      (r:eval-exp (first (r:apply-args exp)))
      (r:eval-exp (second (r:apply-args exp))))]
    [else (error "Unknown expression:" exp)]))
```

Example `r:eval-exp`

```
(define/match (r:eval-exp exp)
  ; 1. When evaluating a number, just return that number
  [((r:number n)) n]
  ; 2. When evaluating an arithmetic symbol, return the respective arithmetic function
  [((r:variable x)) (r:eval-builtin x)]
  ; 3. When evaluating a function call evaluate each expression and apply
  ;    the first expression to remaining ones
  [((r:apply ef (list ea1 ea2))) ((r:eval-exp ef) (r:eval-exp ea1) (r:eval-exp ea2))]
  [(_) (error "Unknown expression:" exp)])
```

Formalism

$$n \Downarrow n \qquad x \Downarrow \text{builtin}(x) \qquad \frac{e_f \Downarrow v_f \qquad e_{a_1} \Downarrow v_{a_1} \qquad e_{a_2} \Downarrow v_{a_2} \qquad v = v_f(v_{a_1}, v_{a_2})}{(e_f\ e_{a_1}\ e_{a_2}) \Downarrow v}$$

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)`

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

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?