

CS450

Structure of Higher Level Languages

Lecture 13: Difficulty in adding definitions

Tiago Cogumbreiro

Implementing inductive definitions

A primer

Implementing inductive definitions

A primer

Disciplining an ambiguous presentation medium to communicate a precise mathematical meaning (**notation** and **convention**)

- Implementing algorithms written in a mathematical notation
- Discuss recursive functions (known as inductive definitions)
- Present various design choices
- We are restricting ourselves to the specification of functions
(If $M(x) = y$ and $M(x) = z$, then $y = z$)

Equation notation

- Function $M(n)$ has one input n and one output after the equals sign.
- Each rule declares some pre-conditions
- The result of the function is only returned if the pre-conditions are met

Formally

$$\begin{aligned} M(n) &= n - 10 && \text{if } n > 100 \\ M(n) &= M(M(n + 11)) && \text{if } n \leq 100 \end{aligned}$$

Implementation

- Each branch of the `cond` represents a rule
- The condition of each branch should be the pre-condition



Equation notation

- Function $M(n)$ has one input n and one output after the equals sign.
- Each rule declares some pre-conditions
- The result of the function is only returned if the pre-conditions are met

Formally

$$\begin{aligned} M(n) &= n - 10 && \text{if } n > 100 \\ M(n) &= M(M(n + 11)) && \text{if } n \leq 100 \end{aligned}$$

Implementation

- Each branch of the `cond` represents a rule
- The condition of each branch should be the pre-condition

```
(define (M n)
  (cond
    [(> n 100) (- n 10)]
    [(≤ n 100) (M (M (+ n 11)))]))
```



Fraction notation

- We can use the "fraction"-based notation to represent pre-conditions
- Above is a pre-condition, below is the result of the function
- The result is only available if the pre-condition holds

Formally

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{n \leq 100}{M(n) = M(M(n + 11))}$$

Fraction notation

- We can use the "fraction"-based notation to represent pre-conditions
- Above is a pre-condition, below is the result of the function
- The result is only available if the pre-condition holds

Formally

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{n \leq 100}{M(n) = M(M(n + 11))}$$

Implementation

```
(define (M n)
  (cond
    [(> n 100) (- n 10)]
    [(≤ n 100) (M (M (+ n 11)))]))
```

Multiple pre-conditions in fraction-notation

- Fraction-based notation admits multiple pre-conditions
- The result only happens if **all** pre-conditions are met (logical conjunction)
- We are only interested in function calls that do always succeed (ignore errors)
- Since we are defining functions, only one output is possible at any time

$$\frac{n > 100}{M(n) = n - 10} \quad \frac{M(n + 11) = x \quad M(x) = y \quad n \leq 100}{M(n) = y}$$

- In the second rule, note the implicit dependency between variables
- The dependency between variables, specifies the implementation order (eg, x must be defined before y)

Multiple pre-conditions in fraction-notation

- Fraction-based notation admits multiple pre-conditions
- The result only happens if **all** pre-conditions are met (logical conjunction)
- We are only interested in function calls that do always succeed (ignore errors)
- Since we are defining functions, only one output is possible at any time

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{M(n + 11) = x \quad M(x) = y \quad n \leq 100}{M(n) = y}$$

- In the second rule, note the implicit dependency between variables
- The dependency between variables, specifies the implementation order (eg, `x` must be defined before `y`)

```
(define (M n)
  (cond
    [(> n 100) (- n 10)]
    [(≤ n 100)
     (define x (M (+ n 11)))
     (define y (M x))
     y]))
```

The equal sign is optional

- The distinction between input and output should be made clear by the author of the formalism

$$\frac{n > 100}{M(n) = n - 10}$$

$$\frac{M(n + 11) = x \quad M(x) = y \quad n \leq 100}{M(n) = y}$$

The equal sign is optional

- The distinction between input and output should be made clear by the author of the formalism

$$\frac{n > 100}{M(n) = n - 10} \qquad \frac{M(n + 11) = x \quad M(x) = y \quad n \leq 100}{M(n) = y}$$

We can use any symbol!

Let us define the M function with the 📻 symbol. The intent of notation is to aid the reader and reduce verbosity.

$$\frac{n > 100}{n \text{ 📻 } n - 10} \qquad \frac{n + 11 \text{ 📻 } x \quad x \text{ 📻 } y \quad n \leq 100}{n \text{ 📻 } y}$$

How do we write $M(M(n + 11))$?

Pattern matching rules

- The pre-condition is implicitly defined according to the **structure** of the input
- **First rule:** can only be applied if the list is empty
- **Second rule:** can only be applied if there is at least one element in the list

$$\text{qs}([]) = []$$

$$\frac{\text{qs}([x \mid x < p \wedge x \in l]) = l_1 \quad \text{qs}([x \mid x \geq p \wedge x \in l]) = l_2}{\text{qs}(p :: l) = l_1 \cdot [p] \cdot l_2}$$

Pattern matching rules (implementation)

```
(define (qs l)
  (cond [(empty? l) empty] ; qs([]) = []
        [else
         ; Input: p :: r
         (define p (first l))
         (define r (rest l))
         ; qs([ x | x < p /\ x \in l]) = l1
         (define l1 (qs (filter (lambda (x) (< x p)) r)))
         ; qs([ x | x ≥ p /\ x \in l]) = l2
         (define l2 (qs (filter (lambda (x) (≥ x p)) r)))
         ; l1 . p . l2
         (append l1 (cons p l2))]))
```

Homework assignment 4

- **Exercise 1.** Function $e[x := v]$ is `(s:subst exp var val)`, where e is `exp`, x is `var`, and v is `val`.
- **Exercise 2.** Function $e \Downarrow v$ is `(s:eval subst exp)`, where e is `exp`, v is the return value (not displayed in the function signature).

In the exercise, parameter `subst` represents the substitution function (local tests use your own implementation, remote tests use a correct implementation of `subst`).

- **Exercise 3.** Function $e \Downarrow_E v$ is `(e:eval env exp)`, where e is `exp`, E is `env`, v is the return value (not displayed in the function signature).



Language λ_F

How do we add support for definitions?

Language λ_F

How do we add support for definitions?

- We extend the our language (λ_E) with `define`
- We introduce the AST
- We discuss parsing our language

λ_F : Understanding definitions

Syntax

$$t ::= e \mid t; t \mid (\text{define } x \ e)$$
$$e ::= v \mid x \mid (e_1 \ e_2) \mid \lambda x. t \qquad v ::= n \mid (E, \lambda x. t) \mid \text{void}$$

- New grammar rule: **terms**
- A program is now a non-empty sequence of terms
- Since we are describing the **abstract** syntax, there is no distinction between a basic and a function definition
- Since evaluating a definition returns a void, we need to update values

Values

■ We add `void` to values.

$$v ::= n \mid (E, \lambda x.t) \mid \text{void}$$

Racket implementation

```
;; Values  
(define (f:value? v) (or (f:number? v) (f:closure? v) (f:void? v)))  
(struct f:number (value) #:transparent)  
(struct f:closure (env decl) #:transparent)  
(struct f:void () #:transparent)
```

Expressions

Expressions remain unchanged.

$$e ::= v \mid x \mid (e_1 \ e_2) \mid \lambda x. t$$

Racket implementation

```
(define (f:expression? e) (or (f:value? e) (f:variable? e) (f:apply? e) (f:lambda? e)))  
(struct f:variable (name) #:transparent)  
(struct f:apply (func args) #:transparent)  
(struct f:lambda (params body) #:transparent)
```

Terms

■ We implement terms below.

$$t ::= e \mid t; t \mid (\text{define } x \ e)$$

Racket implementation

```
(define (f:term? t) (or (f:expression? t) (f:seq? t) (f:define? t)))  
(struct f:seq (fst snd) #:transparent)  
(struct f:define (var body) #:transparent)
```

The body of a function declaration is a single term

The body is no longer a list of terms!

■ A sequence is not present in the concrete syntax, but it simplifies the implementation and formalism (see reduction)



Parsing datum into AST terms

- Our parser handles multiple terms in the body of a function declaration.
- Function `f:parse1` parses a single term.

```
(check-equal?  
  (f:parse1 '(lambda (x) x y z))  
  (f:lambda (list (f:variable 'x))  
    (f:seq (f:variable 'x)  
      (f:seq (f:variable 'y) (f:variable 'z))))))
```

Parsing datum into AST terms

- The body of a function can have one or more definitions, values, or function calls.

```
(check-equal?
  (f:parse1 '(lambda (x) (define x 3) x))
  (f:lambda (list (f:variable 'x))
    (f:seq (f:define (f:variable 'x) (f:number 3)) (f:variable 'x))))
```

Parsing datum into AST terms

- Parsing supports function definitions.
- Function `f:parse` can parse a sequence of terms, which corresponds to a Racket program.

```
(check-equal?  
  (f:parse '[(define (f x) x)])  
  (f:define (f:variable 'f) (f:lambda (list (f:variable 'x)) (f:variable 'x))))
```

λ_F semantics

The incorrect way of implementing `define`

λ_F semantics

The incorrect way of implementing

Semantics $t \Downarrow_E \langle E, v \rangle$

$$\frac{e \Downarrow_E v}{e \Downarrow_E \langle E, v \rangle} \quad (\text{E-exp})$$

- Evaluating a define **extends** the environment with a new binding
- Sequencing must thread the environments

$$\frac{e \Downarrow_E v}{(\text{define } x \ e) \Downarrow_E \langle E[x \mapsto v], \text{void} \rangle} \quad (\text{E-def})$$

$$\frac{t_1 \Downarrow_{E_1} \langle E_2, v_1 \rangle \quad t_2 \Downarrow_{E_2} \langle E_3, v_2 \rangle}{t_1; t_2 \Downarrow_{E_1} \langle E_3, v_2 \rangle} \quad (\text{E-seq})$$

The Language λ_F

$$v \Downarrow_E v \quad (\mathbf{E}\text{-val})$$

$$x \Downarrow_E E(x) \quad (\mathbf{E}\text{-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \quad (\mathbf{E}\text{-lam})$$

$$\frac{e_f \Downarrow_E (E_b, \lambda x.t_b) \quad e_a \Downarrow_E v_a \quad t_b \Downarrow_{\mathbf{E}_b[x \mapsto v_a]} v_b}{(e_f e_a) \Downarrow v_b} \quad (\mathbf{E}\text{-app})$$

$$\frac{e \Downarrow_E v}{e \Downarrow_E (E, v)} \quad (\mathbf{E}\text{-exp})$$

$$\frac{e \Downarrow_E v}{(\mathbf{define} \ x \ e) \Downarrow_E (E[x \mapsto v], \mathbf{void})} \quad (\mathbf{E}\text{-def})$$

$$\frac{t_1 \Downarrow_{E_1} (E_2, v_1) \quad t_2 \Downarrow_{E_2} (E_3, v_2)}{t_1; t_2 \Downarrow_{E_1} (E_3, v_2)} \quad (\mathbf{E}\text{-seq})$$

Why λ_F is incorrect?

Evaluating define

Example 1

■ Consider the following program

```
(define a 20)
(define b (lambda (x) a))
(b 1)
```

■ What is the output of this program?

Evaluating define

Example 1

■ Consider the following program

```
(define a 20)
(define b (lambda (x) a))
(b 1)
```

■ What is the output of this program? The output is: 20

Let us try and evaluate this program with our λ_F semantics!

Example 1: step 1

Input

```
Environment: []  
Term: (define a 20)
```

Example 1: step 1

Input

```
Environment: []  
Term: (define a 20)
```

Evaluating

Output

```
Environment: [ (a . 20) ]  
Value: #<void>
```

Example 1: step 1

Input

```
Environment: []  
Term: (define a 20)
```

Output

```
Environment: [ (a . 20) ]  
Value: #<void>
```

Evaluating

$$\frac{20 \Downarrow_{\{\}} 20 \quad (\text{E-val})}{(\text{define } a \ 20) \Downarrow_{\{\}} (\{a : 20\}, \text{void})} \text{E-def}$$

Example 1: step 2

Input

```
Environment: [ (a . 20) ]  
Term: (define b (lambda (y) a))
```

Example 1: step 2

Input

```
Environment: [ (a . 20) ]  
Term: (define b (lambda (y) a))
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Value: #<void>
```

Example 1: step 2

Input

```
Environment: [ (a . 20) ]  
Term: (define b (lambda (y) a))
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Value: #<void>
```

Evaluating

$$\frac{\lambda y.a \Downarrow_{\{a:20\}} (\{a : 20\}, \lambda y.a) \quad (\text{E-lam})}{(\text{define } b \lambda y.a) \Downarrow_{\{a:20\}} (\{a : 20, b : (\{a : 20\}, \lambda y.a)\}, \text{void})} \text{E-def}$$

Example 1: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Term: (b 1)
```

Example 1: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Term: (b 1)
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Value: 20
```

Evaluation

Example 1: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Term: (b 1)
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [(a . 20)] (lambda (y) a)))  
]  
Value: 20
```

Evaluation

$$\frac{\frac{E(b) = (\{a : 20\}, \lambda y. a)}{b \Downarrow_E (\{a : 20\}, \lambda y. a)} \text{E-var} \quad \frac{}{1 \Downarrow_E 1} \text{E-val} \quad \frac{F(a) = 20}{a \Downarrow_F 20} \text{E-var}}{(b \ 1) \Downarrow_E 20} \text{E-app}$$
$$\frac{(b \ 1) \Downarrow_E 20}{(b \ 1) \Downarrow_E (E, 20)} \text{E-exp}$$

where

$$E = \{a : 20, b : (\{a : 20\}, \lambda y. a)\}$$

$$F = \{a : 20\}[y \mapsto 1] = \{a : 20, y : 1\}$$



Evaluating define

Example 2

Evaluating define

Example 2

Consider the following program

```
(define b (lambda (x) a))  
(define a 20)  
(b 1)
```

What is the output of this program?

Evaluating define

Example 2

■ Consider the following program

```
(define b (lambda (x) a))  
(define a 20)  
(b 1)
```

■ What is the output of this program? The output is: 20

Let us try and evaluate this program with our λ_F semantics!

Example 2: step 1

Input

Environment: []

Term: (define b (lambda (y) a))

Example 2: step 1

Input

```
Environment: []  
Term: (define b (lambda (y) a))
```

Output

```
Environment: [  
  (b . (closure [] (lambda (y) a)))  
]  
Value: #<void>
```

Evaluation

Example 2: step 1

Input

```
Environment: []  
Term: (define b (lambda (y) a))
```

Output

```
Environment: [  
  (b . (closure [] (lambda (y) a))  
)]  
Value: #<void>
```

Evaluation

$$\frac{\lambda y.a \Downarrow_{\{\}} (\{\}, \lambda y.a) \quad (\text{E-lam})}{(\text{define } b \lambda y.a) \Downarrow_{\{\}} (\{b : (\{\}, \lambda y.a)\}, \text{void})} \text{E-def}$$

Example 2: step 2

Input

```
Environment: [  
  (b . (closure [] (lambda (y) a))  
]  
Term: (define a 20)
```

Example 2: step 2

Input

```
Environment: [  
  (b . (closure [] (lambda (y) a))  
  ]  
Term: (define a 20)
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
  ]  
Value: #<void>
```

Evaluation

Example 2: step 2

Input

```
Environment: [  
  (b . (closure [] (lambda (y) a))  
]  
Term: (define a 20)
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
]  
Value: #<void>
```

Evaluation

$$\frac{20 \Downarrow_{\{b: (\{\}, \lambda y. a)\}} 20 \quad (\text{E-val})}{(\text{define } a \ 20) \Downarrow_{\{b: (\{\}, \lambda y. a)\}} (\{b: (\{\}, \lambda y. a), a: 20\}, \text{void})} \text{E-def}$$

Example 2: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
]  
Term: (b 1)
```


Example 2: step 3

Input

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
]  
Term: (b 1)
```

Output

```
Environment: [  
  (a . 20)  
  (b . (closure [] (lambda (y) a))  
]  
Value: error! a is undefined
```

Insight

When creating a closure we copied the existing environment, and therefore any future updates are forgotten.

The semantics of λ_F is not enough! We need to introduce a notion of **mutation**.

