CS450

Structure of Higher Level Languages

Lecture 21: Language λ_F: adding definitions incorrectly

Tiago Cogumbreiro

Today we will learn...

1. A primer on implementing inductive definitions
2. Extend λ_E with `define`
3. Extend the semantics **incorrectly** (naive approach)
4. Give an example of why it is incorrect

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

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

- 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

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

- 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

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

The equal sign is optional

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

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

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.

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

> 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

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

$$\frac{\mathrm{qs}([x \mid x < p \land x \in l]) = l_1 \qquad \mathrm{qs}([x \mid x \geq p \land x \in l]) = l_2}{\mathrm{qs}(p :: l) = l_1 \cdot [p] \cdot l_2}$$

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))]))
```

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

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

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

- 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.

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

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.

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

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 1)])
  (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 $t \Downarrow_E \langle E, v \rangle$

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

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

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

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

The Language λ_F

$$v \Downarrow_E v \qquad (\texttt{E-val})$$

$$x \Downarrow_E E(x) \qquad (\texttt{E-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \qquad (\texttt{E-lam})$$

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

$$\frac{e \Downarrow_E v}{e \Downarrow_E (E, v)} \qquad (\texttt{E-exp})$$

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

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

Why $\boldsymbol{\lambda}_F$ 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

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

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)))
]
Expression: #<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)))
]
Expression: #<void>
```

Evaluating

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

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)))
]
Expression: 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)))
]
Expression: 20
```

Evaluation

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

where

$$E = \{a : 20, b : (\{a : 20\}, \lambda y.a)\}$$
$$F = E[y \mapsto 1] = \{a : 20, b : (\{a : 20\}, \lambda y.a), y : 1\}$$

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))
]
Expression: #<void>
```

Evaluation

Example 2: step 1

Input

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

Output

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

Evaluation

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

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))
]
Expression: #<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))
]
Expression: #<void>
```

Evaluation

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

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))
]
Expression: 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**.