Lecture 21: Language $\lambda_F$: adding definitions incorrectly

Tiago Cogumbreiro
Today we will learn...

1. A primer on implementing inductive definitions
2. Extend $\lambda_E$ with $\text{define}$
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

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

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

\[
M(n) = n - 10 \quad \text{if } n > 100 \\
M(n) = M(M(n + 11)) \quad \text{if } n \leq 100
\]

Implementation

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

\[
(define (M n) \\
  (cond \\
    [(> n 100) (- n 10)] \\
    [(\leq n 100) (M (M (+ n 11)))]))
\]
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:

\[
\begin{align*}
M(n) &= n - 10 & \text{if } n > 100 \\
M(n) &= M(M(n + 1)) & \text{if } n \leq 100
\end{align*}
\]
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

\[
\begin{align*}
&M(n) = n - 10 & & \text{if } n > 100 \\
&M(n) = M(M(n + 11)) & & \text{if } n \leq 100
\end{align*}
\]

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

\[
\begin{align*}
\frac{n > 100}{M(n) = n - 10} & \quad \frac{M(n + 11) = x}{M(x) = y} & \quad n \leq 100 \\
M(n) &= y
\end{align*}
\]

- 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

\[
\begin{align*}
\frac{n > 100}{M(n) = n - 10} & \quad \frac{M(n + 11) = x}{M(x) = y} \quad \frac{n \leq 100}{M(n) = y}
\end{align*}
\]

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

```scheme
(define (M n)
  (cond
    [(> n 100) (- n 10)]
    [(\leq 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

\[
\begin{align*}
    n > 100 & \quad \Rightarrow \quad M(n) = n - 10 \\
    M(n + 11) = x & \quad M(x) = y & \quad n \leq 100
\end{align*}
\]
The equal sign is optional

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

\[
\begin{align*}
n > 100 & \quad \rightarrow \quad M(n) = n - 10 \\
M(n + 11) = x & \quad M(x) = y \quad n \leq 100
\end{align*}
\]

We can use any symbol!

Let us define the \( M \) function with the \( \mathrel{\triangleleft} \) symbol. The intent of notation is to aid the reader and reduce verbosity.

\[
\begin{align*}
n > 100 & \quad \rightarrow \quad n \mathrel{\triangleleft} n - 10 \\
n + 11 \mathrel{\triangleleft} x & \quad x \mathrel{\triangleleft} y \quad n \leq 100
\end{align*}
\]

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

\[ qs([ ]) = [ ] \]

\[ qs([x \mid x < p \land x \in l]) = l_1 \quad qs([x \mid x \geq p \land x \in l]) = l_2 \]

\[ qs(p :: l) = l_1 \cdot [p] \cdot l_2 \]
Pattern matching rules (implementation)

\[(\text{define } (\text{qs } l))\]
\[
(\text{cond} \begin{cases} 
(\text{empty? } l) \text{ empty} &; \text{qs}([]) = [] \\
\text{else} &; \text{Input: } p :: r
\end{cases})
\]
\[
(\text{define } p (\text{first } l))
(\text{define } r (\text{rest } l))
\]
\[
(\text{qs}([x | x < p / \ x \ in \ l]) = \text{l1})
(\text{define } \text{l1} (\text{qs} (\text{filter} (\text{lambda} (x) (x < p)) r)))
\]
\[
(\text{qs}([x | x \geq p / \ x \ in \ l]) = \text{l2})
(\text{define } \text{l2} (\text{qs} (\text{filter} (\text{lambda} (x) (x \geq p)) r)))
\]
\[
\text{l1} . p . \text{l2} \text{(append l1 (cons p l2))])
\]
Language $\lambda_F$

How do we add support for definitions?
How do we add support for definitions?

- We extend the our language ($\lambda_E$) with define
- We introduce the AST
- We discuss parsing our language
Syntax

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

\[
e ::= v \mid x \mid (e_1 \ e_2) \mid \lambda x.\ t \quad v ::= n \mid (E, \lambda x. t) \mid \text{void}
\]

- 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

```racket
;; 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 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

```racket
(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)
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)))))
```
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 \texttt{f:parse} can parse a sequence of terms, which corresponds to a Racket program.

\begin{verbatim}
(check-equal?  
 (f:parse '[(define (f x) (f 1))])  
 (f:define (f:variable 'f) (f:lambda (list (f:variable 'x)) (f:variable 'x))))
\end{verbatim}
The incorrect way of implementing define
\[ \lambda_F \text{ semantics} \]

The incorrect way of implementing

**Semantics** \( t \Downarrow_E \langle E, v \rangle \)

\[
\begin{align*}
  e & \Downarrow_E v \\
  e & \Downarrow_E \langle E, v \rangle 
\end{align*}
\]

\( (E-\text{exp}) \)

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

\[
\begin{align*}
  e & \Downarrow_E v \\
  (\text{define } x \ e) & \Downarrow_E \langle E[x \mapsto v], \text{void} \rangle 
\end{align*}
\]

\( (E-\text{def}) \)

\[
\begin{align*}
  t_1 & \Downarrow_{E_1} \langle E_2, v_1 \rangle \\
  t_2 & \Downarrow_{E_2} \langle E_3, v_2 \rangle \\
  t_1; t_2 & \Downarrow_{E_1} \langle E_3, v_2 \rangle 
\end{align*}
\]

\( (E-\text{seq}) \)
The Language $\lambda_F$

\[ v \downarrow_E v \quad \text{(E-val)} \]

\[ x \downarrow_E E(x) \quad \text{(E-var)} \]

\[ \lambda x.t \downarrow_E (E, \lambda x.t) \quad \text{(E-lam)} \]

\[ e_f \downarrow_E (E_b, \lambda x.t_b) \quad e_a \downarrow_E v_a \quad t_b \downarrow_{E_b[x\mapsto v_a]} v_b \]

\[ \frac{e_f \downarrow_E (E_b, \lambda x.t_b)}{(e_f e_a) \downarrow v_b} \quad \frac{e_a \downarrow_E v_a}{(e_f e_a) \downarrow v_b} \quad \frac{t_b \downarrow_{E_b[x\mapsto v_a]} v_b}{(e_f e_a) \downarrow v_b} \quad \text{(E-app)} \]

\[ e \downarrow_E v \]

\[ e \downarrow_E (E,v) \quad \text{(E-exp)} \]

\[ e \downarrow_E v \]

\[ (\text{define } x e) \downarrow_E (E[x \mapsto v], \text{void}) \quad \text{(E-def)} \]

\[ t_1 \downarrow_{E_1} (E_2, v_1) \quad t_2 \downarrow_{E_2} (E_3, v_2) \]

\[ t_1; t_2 \downarrow_{E_1} (E_3, v_2) \quad \text{(E-seq)} \]
Why $\lambda_F$ 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 $\lambda_F$ semantics!
Example 1: step 1

Input

Environment: []
Term: (define a 20)
Example 1: step 1

Input

Environment: []
Term: (define a 20)

Output

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

Evaluating
Example 1: step 1

Input

Environment: []
Term: (define a 20)

Output

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

Evaluating

\[
\begin{align*}
20 \downarrow &\{\} \ 20 & \quad \text{(E-val)} \\
\text{(define a 20)} \downarrow &\{\} \ (\{a : 20\}, \text{void}) & \quad \text{E-def}
\end{align*}
\]
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

\[
\lambda y.a \downarrow_{\{a:20\}} (\{a:20\}, \lambda y.a) \quad (E-lam)
\]

\[
(define\ b\ \lambda y.a) \downarrow_{\{a:20\}} (\{a:20,\ b:\ (\{a:20\},\ \lambda y.a)\},\ void) \quad E-def
\]
Example 1: step 3

Input

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

Term: (b 1)
Example 1: step 3

Input

Environment:

\[
\begin{align*}
(a & . 20) \\
(b & . (\text{closure } [(a . 20)] (\text{lambda } (y) a)))
\end{align*}
\]

Term: (b 1)

Output

Environment:

\[
\begin{align*}
(a & . 20) \\
(b & . (\text{closure } [(a . 20)] (\text{lambda } (y) a)))
\end{align*}
\]

Expression: 20
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

\[
E(b) = \{a : 20, \lambda y.a\} \quad \text{E-var} \quad \begin{array}{c}
1 \downarrow_E 1 \\
\end{array} \quad \text{E-val}
\]

\[
F(a) = 20 \quad \text{E-var} \quad \begin{array}{c}
a \downarrow_F 20 \\
\end{array} \quad \text{E-app}
\]

\[
(b 1) \downarrow_E 20 \quad \text{E-exp}
\]

\[
(b 1) \downarrow_E (E, 20)
\]

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

\[
\begin{align*}
&\text{(define } b \text{ (lambda } (x) a)) \\
&\text{(define } a \text{ 20)} \\
&b \text{ 1}
\end{align*}
\]

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

Let us try and evaluate this program with our $\lambda_F$ 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

\[
\lambda y.a \downarrow \{\} (\{\}, \lambda y.a) \quad (E-lam)
\]

\[
(define b \lambda y.a) \downarrow \{\} (\{b : (\{\}, \lambda y.a)\}, \text{void}) \quad E-def
\]
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>
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

\[
\begin{align*}
20 \downarrow_{b:(\{}\lambda y.a)} & \quad 20 \quad \text{(E-val)} \\
\text{(define a 20)} \downarrow_{b:(\{}\lambda y.a)} & \quad (\{b : (\{} , \lambda y.a), a : 20\}, \text{void}) \quad \text{E-def}
\end{align*}
\]
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 $\lambda_F$ is not enough! We need to introduce a notion of mutation.