CS450
Structure of Higher Level Languages
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 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

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

Implementation

- Each branch of the cond represents a rule
- The condition of each branch should be the pre-condition
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)]
    [(<= 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

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

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

Implementation

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

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

- 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)

```
(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

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

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

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

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.

\[
\begin{align*}
\text{if } n > 100 & \quad \text{then } n \oplus n - 10 \\
n + 11 \oplus x & \quad x \oplus y \quad \text{if } n \leq 100
\end{align*}
\]

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)

\[
\begin{align*}
\text{(define (qs l)} & \text{ (cond [(empty? l) empty]; qs([]) = [])} \\
& \text{[else}} \text{ Input: } p :: r \text{)} \\
& \text{(define p (first l))} \\
& \text{(define r (rest l))} \\
& \text{qs([ x | x < p \land x \in l]) = l1} \\
& \text{(define l1 (qs (filter (lambda (x) (< x p)) r)))} \\
& \text{qs([ x | x \geq p \land x \in l]) = l2} \\
& \text{(define l2 (qs (filter (lambda (x) (\geq x p)) r))} \\
& \text{ (append l1 (cons p l2))])}
\end{align*}
\]
How do we add support for definitions?
Language $\lambda_F$

How do we add support for definitions?

- We extend the our language ($\lambda_F$) with define
- We introduce the AST
- We discuss parsing our language
**λ_F**: Understanding definitions

**Syntax**

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

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

- 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

Expressions remain unchanged.

\[ e ::= v \mid x \mid (e_1 \; e_2) \mid \lambda x . t \]

Racket implementation

```racket
(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)
```
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)))))
```
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.

```scheme
(check-equal? (f:parse '[[define (f x) x]])
(f:define (f:variable 'f) (f:lambda (list (f:variable 'x)) (f:variable 'x))))
```
$\lambda_F$ semantics

The incorrect way of implementing define
**λ_F semantics**

The incorrect way of implementing

**Semantics** $t \Downarrow_{E} \langle E, v \rangle$

$$e \Downarrow_{E} v \quad (E\text{-exp})$$

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

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

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

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

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

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

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

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

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

\[
\frac{t_1 \Downarrow_{E_1} (E_2, v_1) \quad t_2 \Downarrow_{E_2} (E_3, v_2)}{t_1; t_2 \Downarrow_{E_1} (E_3, v_2)} \quad \text{(E-seq)}
\]
Why $\lambda_F$ is incorrect?
Evaluating define

Example 1

Consider the following program

```lisp
(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

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

\[ \lambda y.a \downarrow_{\{a:20\}} (\{a:20\}, \lambda y.a) \quad \text{(E-lam)} \]
\[ (\text{define } b \; \lambda y.a) \downarrow_{\{a:20\}} (\{a:20\}, b : (\{a:20\}, \lambda y.a), \text{void}) \quad \text{E-def} \]
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
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

\[ E(b) = \{a : 20, \lambda y.a\} \quad E-var \]

\[ b \downarrow_E (\{a : 20\}, \lambda y.a) \]

\[ 1 \downarrow_E 1 \quad E-val \]

\[ (b 1) \downarrow_E 20 \quad E-app \]

\[ (b 1) \downarrow_E (E, 20) \quad E-exp \]

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 $\lambda_F$ 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))

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

\[
\begin{align*}
\lambda y.a & \downarrow \{\}, \lambda y.a \quad \text{(E-lam)} \\
\text{(define } b \lambda y.a) & \downarrow \{\} \quad \{b : (\{\}, \lambda y.a)\}, \text{void} \quad \text{(E-def)}
\end{align*}
\]
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>
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

\[
\begin{align*}
20 & \Downarrow_{\{b:()\}, \lambda y.a}\, 20 & \text{E-val} \\
\text{(define a 20)} & \Downarrow_{\{b:()\}, \lambda y.a}\, \{b : (\{} \}, \lambda y.a), a : 20\}, \text{void} & \text{E-def}
\end{align*}
\]
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)))
]

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 $\lambda_F$ is not enough! We need to introduce a notion of **mutation**.