Lecture 7-8: Solutions of the Exercises

prof. Marco Bellia, Dip. Informatica, Università di Pisa

March 25, 2014

- Let $\mathtt{F} \equiv \lambda \mathtt{f}.\lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then\ 1\ else\ n} * \mathtt{f}(\mathtt{n}-1)$ be a functional.
  Notationa remark: $\mathtt{Y\ F}$ is a contraction for: $\mathtt{Yf}.\mathtt{F}$
- **To compute $\mathtt{Y}$:**
  Intensional computation. $\forall \mathtt{x} : \mathtt{Y\ F(x)} = \mathtt{F\ (Y\ F)(x)}$
  Extensional computation. $\mathtt{Y\ F} = \mathtt{Lim}_{\mathtt{n}\in\mathtt{Nat}}(\mathtt{Y\ F})^{\mathtt{n}}$ where:
  $(\mathtt{Y\ F})^0 = \mathtt{F}(\bot)$
  $(\mathtt{Y\ F})^{\mathtt{n}} = \mathtt{F}((\mathtt{Y\ F})^{\mathtt{n}-1})$

Example

Complete the extensional c. of $\mathtt{Y\ F}\ (2)$ and show how the result has been obtained

Intensional:

$\mathtt{Y\ F}\ (2) = \mathtt{if}(2=0)\mathtt{then\ 1\ else\ 2} * \mathtt{Y\ F}\ (1)$
$= \ldots = 2 * 1 * 1$

Extensional:

$\mathtt{Y\ F}^0 = \lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then\ 1\ else}\ \bot$
$\mathtt{Y\ F}^1 = \lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then\ 1\ else}\ (\mathtt{if}(\mathtt{n}=1)\mathtt{then\ 1\ else}\bot)$
$\mathtt{Y\ F}^2 = \ldots$

Lecture7-8-2014: Slide 6 : Solved

- Let $\mathtt{F} \equiv \lambda \mathtt{f}.\lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else}\ \mathtt{n} * \mathtt{f}(\mathtt{n}-1)$ be a functional.
  Notationa remark: $\mathtt{Y}\ \mathtt{F}$ is a contraction for: $\mathtt{Y}\mathtt{f}.\mathtt{F}$
- Extensional computation. $\mathtt{Y}\ \mathtt{F} = \mathtt{Lim}_{\mathtt{n}\in\mathtt{Nat}}(\mathtt{Y}\ \mathtt{F})^{\mathtt{n}}$ where:

$$(\mathtt{Y}\ \mathtt{F})^0 = \mathtt{F}(\bot)$$
$$(\mathtt{Y}\ \mathtt{F})^{\mathtt{n}} = \mathtt{F}((\mathtt{Y}\ \mathtt{F})^{\mathtt{n}-1})$$

Example

Complete the extensional c. of $\mathtt{Y}\ \mathtt{F}$ (2) and show how the result has been obtained
Extensional: First of all, we need compute $\mathtt{Y}\ \mathtt{F}^2$. Then we apply it to 2

$$\mathtt{Y}\ \mathtt{F}^0 = \lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else}\ \bot$$
$$\mathtt{Y}\ \mathtt{F}^1 = \lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else}\ (\mathtt{if}(\mathtt{n}=1)\mathtt{then}\ 1\ \mathtt{else}\bot)$$
$$\begin{aligned}
\mathtt{Y}\ \mathtt{F}^2 &= (\lambda \mathtt{f}.\lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else}\ \mathtt{n} * \mathtt{f}(\mathtt{n}-1))(\mathtt{Y}\ \mathtt{F}^1) \\
&= \lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else}\ \mathtt{n} * (\mathtt{Y}\ \mathtt{F}^1)(\mathtt{n}-1) \\
&= \lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else} \\
&\qquad \mathtt{n} * (\lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else}\ (\mathtt{if}(\mathtt{n}=1)\mathtt{then}\ 1\ \mathtt{else}\bot))(\mathtt{n}-1) \\
&= \lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else} \\
&\qquad \mathtt{n} * (\mathtt{if}(\mathtt{n}-1=0)\mathtt{then}\ 1\ \mathtt{else}\ (\mathtt{if}(\mathtt{n}-1=1)\mathtt{then}\ 1\ \mathtt{else}\bot)) \\
&= \lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else} \\
&\qquad (\mathtt{if}(\mathtt{n}-1=0)\mathtt{then}\ \mathtt{n}*1\ \mathtt{else}\ (\mathtt{if}(\mathtt{n}-1=1)\mathtt{then}\ \mathtt{n}*1\ \mathtt{else}\bot)) \\
&= \lambda \mathtt{n}.\mathtt{if}(\mathtt{n}=0)\mathtt{then}\ 1\ \mathtt{else} \\
&\qquad (\mathtt{if}(\mathtt{n}=1)\mathtt{then}\ 1*1\ \mathtt{else}\ (\mathtt{if}(\mathtt{n}=2)\mathtt{then}\ 2*1\ \mathtt{else}\bot))
\end{aligned}$$
$$\mathtt{Y}\ \mathtt{F}^2\ (2) = 2*1$$

Lecture7-8-2014: Slide 12-13: Text

let $\mathcal{D}[\![\texttt{Proc I() C}]\!]_\rho = \texttt{bind}(\texttt{I}, \mathcal{M}[\![\texttt{C}]\!]_\rho, \rho)$.
What is the relation between Static Chain and the ρ?

Lecture7-8-2014: Slide 12-13 - solved

let $\mathcal{D}[\![\texttt{Proc I() C}]\!]_\rho = \texttt{bind}(\texttt{I}, \mathcal{M}[\![\texttt{C}]\!]_\rho, \rho)$.
What is the relation between Static Chain and the ρ?

Answer: ρ is (one-to-one with) the sequence of the frames (i.e. local ENV) of the AR's of the Static Chain.

Lecture7-8-2014: Slide 14-15: Text

Complete in the case of Dynamic Scope:
$\mathcal{D}[\![\texttt{Proc I() C}]\!]_\rho = \ldots$

and answer:

- is Dynamic Chain known at the time of the proc. declaration (i.e. compile time)?
- is Dynamic Chain of a procedure known before the proc. invocation?
- is Dynamic Chain of a procedure the same for all the proc. invocation?

Lecture7-8-2014: Slide 14-15 - solved

Complete in the case of Dynamic Scope:
$\mathcal{D}[\![\texttt{Proc I() C}]\!]_\rho = \texttt{bind}(\texttt{I}, \lambda\delta.\mathcal{M}[\![\texttt{C}]\!]_\delta, \rho)$

and answer:

- is Dynamic Chain known at the time of the proc. declaration (i.e. compile time)?

  NO
- is Dynamic Chain of a procedure known before the proc. invocation?

  NO
- is Dynamic Chain of a procedure the same for all the proc. invocation?

  NO

Example

Exercise 1.
The formula written for parallel declaration is:

$$\mathcal{D}[\![\texttt{Mut D}_1\ \texttt{D}_2\ \texttt{Ally}]\!]_\rho = \texttt{Y}\mu.\mathcal{D}[\![\texttt{D}_2]\!](\mathcal{D}[\![\texttt{D}_1]\!](\rho)(\mu))$$

This writing contains a small bug.
(a) Can you find it?
(b) Do you know how to correct it?
(c) Which consequences in letting the formula unchanged?

Exercise 2.
(a) Do You recognize the language used in the interactive sessions below?
(b)
let rec x = fun u → u + y and y = 5 in x(3);;
- ... what will be printed here?
(c)
let rec x = fun u → y(u) and y = fun u → x(u) in x;;
- ... what will be printed here?
(d)
let rec onetwo = 1::twoone and twoone=2::onetwo in List.nth onetwo 5;;
- ... what will be printed here?
(e)
let rec onetwo = 1::twoone and twoone=2::onetwo in twoone;;
- ... what will be printed here?

Lecture7-8-2014: Slide 17-18 - Solved1

Example

Exercise 1.
The formula written for parallel declaration is:

$$\mathcal{D}[\![\texttt{Mut D}_1\ \texttt{D}_2\ \texttt{Ally}]\!]_\rho = \texttt{Y}\mu.\mathcal{D}[\![\texttt{D}_2]\!](\mathcal{D}[\![\texttt{D}_1]\!](\rho)(\mu))$$
$$\text{where } \mathcal{D}[\![\texttt{D}_1]\!] = \lambda\sigma.\lambda\mu.\texttt{bind}(\texttt{I}_1, \mathcal{F}[\![d_1]\!]_\mu, \sigma)$$
$$\mathcal{D}[\![\texttt{D}_2]\!] = \lambda\sigma.\lambda\mu.\texttt{bind}(\texttt{I}_2, \mathcal{F}[\![d_2]\!]_\mu, \sigma)$$

This writing contains a small bug.
(a) Can you find it?
(b) Do you know how to correct it?
(c) Which consequences in letting the formula unchanged?

Answers.
(a) Yes. It is the use of a term $\mathcal{D}[\![\texttt{D}_1]\!](\rho)(\mu)$ having μ unbound as the argument of a term $\mathcal{D}[\![\texttt{D}_2]\!]$ having μ bound: i.e.
$\mathcal{D}[\![\texttt{D}_1]\!](\rho)(\mu) = \texttt{bind}(\texttt{I}_1, \mathcal{F}[\![d_1]\!]_\mu, \rho)$ has μ free
$\mathcal{D}[\![\texttt{D}_2]\!] = \lambda\sigma.\lambda\mu.\texttt{bind}(\texttt{I}_2, \mathcal{F}[\![d_2]\!]_\mu, \sigma)$ has μ and σ bound
$\mathcal{D}[\![\texttt{D}_2]\!](\mathcal{D}[\![\texttt{D}_1]\!](\rho)(\mu)) = \lambda\mu.\texttt{bind}(\texttt{I}_2, \mathcal{F}[\![d_2]\!]_\mu, \texttt{bind}(\texttt{I}_1, \mathcal{F}[\![d_1]\!]_\mu, \rho))$ the wrong substitution "captures" the "free" μ but the first occurrence of μ should not be the same of the second one.
(b) Yes.
$\mathcal{D}[\![\texttt{Mut D}_1\ \texttt{D}_2\ \texttt{Ally}]\!]_\rho = \texttt{Y}\mu.\lambda\mu.(\mathcal{D}[\![\texttt{D}_2]\!](\mathcal{D}[\![\texttt{D}_1]\!](\rho)(\mu))(\mu))$
(c) The formula was wrong defined, i.e. in Lambda Calculus the correct substitution returns:
$\mathcal{D}[\![\texttt{D}_2]\!](\mathcal{D}[\![\texttt{D}_1]\!](\rho)(\mu)) = \lambda\mu.\texttt{bind}(\texttt{I}_2, \mathcal{F}[\![d_2]\!]_\mu, \mathcal{D}[\![\texttt{D}_1]\!](\rho)(\mu'))$ which is not a closed term (since μ' is unbound and) hence computes nothing

Example

Exercise 2.
(a) Do You recognize the language used in the interactive sessions below?
(b)
let rec x = fun u → u + y and y = 5 in x(3);;
- ... what will be printed here?
(c)
let rec x = fun u → y(u) and y = fun u → x(u) in x;;
- ... what will be printed here?
(d)
let rec onetwo = 1::twoone and twoone=2::onetwo in List.nth onetwo 5;;
- ... what will be printed here?
(e)
let rec onetwo = 1::twoone and twoone=2::onetwo in twoone;;
- ... what will be printed here?

Answers.
(a) Ocaml (b) - : int = 8
(c) : 'a →'b = <fun>
(d) -: int = 5
(e) -: int List = [2;1;2;1;...] – depends on the output setting for "infinite values" (i.e. how many approximation steps are set for the "presentation" of infinite values)

Lecture7-8-2014: Slide 19: Text

- Apply the definitions to the declaration below, in the example. To do it:
  - Correct: (a) formula for g and (b) formula for $\mathtt{Y}\ \mathtt{H}^0$;
  - Complete the text.

Example

Let A and B two identifiers. Show the bindings of A and B that the following fragment defines:

```
Mut
  Proc A() {Call B();}
  Proc B() {Call A();}
Ally
```

$\mathtt{g} \equiv \mathtt{Y}\mu.\lambda\sigma.\lambda\mu.\mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_\mu, \sigma)(\lambda\sigma.\lambda\mu.\mathtt{bind}(\mathtt{A}, ...)(\rho)(\mu))$

Compute the first 3 approximations to the solution of the functional:

$\mathtt{H} \equiv \lambda\mu.\mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_\mu, \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\{\mathtt{Call\ B();}\}]\!]_\mu, \rho))$

At the starting step:
$$\begin{aligned} \mathtt{Y}\ \mathtt{H}^0 &= \mathtt{H}(\bot) \\ &= \mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_\mu, \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\{\mathtt{Call\ B();}\}]\!]_\mu, \rho)) \\ \mathtt{Y}\ \mathtt{H}^1 &= \mathtt{H}(\mathtt{Y}\ \mathtt{H}^0) \\ &= ... \\ \mathtt{Y}\ \mathtt{H}^2 &= \mathtt{H}(\mathtt{Y}\ \mathtt{H}^1) \\ &= ... \end{aligned}$$

Lecture7-8-2014: Slide 19: Solved

Example

```
Mut
  Proc A() {Call B();}
  Proc B() {Call A();}
Ally
```

$\mathtt{g} \equiv \mathtt{Y}\mu.\lambda\mu.(\lambda\sigma.\lambda\mu.\mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_\mu, \sigma)(\lambda\sigma.\lambda\mu.\mathtt{bind}(\mathtt{A}, ...)(\rho)(\mu))(\mu))$

Compute the first 3 approximations to the solution of the functional:

$\mathtt{H} \equiv \lambda\mu.\mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_\mu, \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\{\mathtt{Call\ B();}\}]\!]_\mu, \rho))$

At the starting step: $\mathtt{Y\ H}^0 = \mathtt{H}(\bot)$

$= \mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_\bot, \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\{\mathtt{Call\ B();}\}]\!]_\bot, \rho))$

$= \mathtt{bind}(\mathtt{B}, \mathtt{find}(\bot, \mathtt{A}), \mathtt{bind}(\mathtt{A}, \mathtt{find}(\bot, \mathtt{B}), \rho))$

$= \mathtt{bind}(\mathtt{B}, \bot_{\mathtt{Den}}, \mathtt{bind}(\mathtt{A}, \bot_{\mathtt{Den}}, \rho))$

$\mathtt{Y\ H}^1 = \mathtt{H}(\mathtt{Y\ H}^0)$

$= \mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\{\mathtt{Call\ A();}\}]\!]_{\mathtt{H}(\bot)}, \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\{\mathtt{Call\ B();}\}]\!]_{\mathtt{H}(\bot)}, \rho))$

$= \mathtt{bind}(\mathtt{B}, \mathtt{find}(\mathtt{Y\ H}^0, \mathtt{A}), \mathtt{bind}(\mathtt{A}, \mathtt{find}(\mathtt{Y\ H}^0, \mathtt{B}), \rho))$

$= \mathtt{bind}(\mathtt{B}, \bot_{\mathtt{Den}}, \mathtt{bind}(\mathtt{A}, \bot_{\mathtt{Den}}, \rho))$

$= \mathtt{Y\ H}^0$

$\mathtt{Y\ H}^2 = \mathtt{Y\ H}^0$

Example

Show the environment Env when the semantics of mutually applies to the fragment

```
Mut
  final int x = y;
  final int y = 3;
Ally
```

Lecture7-8-2014: Slide 20-21: Solved

Example

Show the environment Env when the semantics of mutually applies to the fragment

```
Mut
  final int x = y;
  final int y = 3;
Ally
```

...

$$\mathtt{g} \equiv \mathtt{Y}\mu.\lambda\mu.(\lambda\sigma.\lambda\mu.\mathtt{bind}(\mathtt{y}, \mathcal{E}[\![3]\!]_\mu, \sigma)(\lambda\sigma.\lambda\mu.\mathtt{bind}(\mathtt{x}, \mathcal{E}[\![\mathtt{y}]\!]_\mu, \sigma)(\rho)(\mu))(\mu))$$

$$\mathtt{H} \equiv \lambda\mu.\mathtt{bind}(\mathtt{y}, \mathcal{E}[\![3]\!]_\mu, \mathtt{bind}(\mathtt{x}, \mathcal{E}[\![\mathtt{y}]\!]_\mu, \rho))$$

At the starting step:

$$\begin{aligned}
\mathtt{Y\ H}^0 &= \mathtt{H}(\bot) \\
&= \mathtt{bind}(\mathtt{y}, \mathcal{E}[\![3]\!]_\bot, \mathtt{bind}(\mathtt{x}, \mathcal{E}[\![\mathtt{y}]\!]_\bot, \rho)) \\
&= \mathtt{bind}(\mathtt{y}, \mathtt{intoVal}(3), \mathtt{bind}(\mathtt{x}, \mathtt{find}(\bot, \mathtt{y}), \rho)) \\
&= \mathtt{bind}(\mathtt{y}, \mathtt{intoVal}(3), \mathtt{bind}(\mathtt{x}, \bot_{\mathtt{Den}}, \rho)) \\
\mathtt{Y\ H}^1 &= \mathtt{H}(\mathtt{Y\ H}^0) \\
&= \mathtt{bind}(\mathtt{y}, \mathcal{E}[\![3]\!]_{\mathtt{YH}^0}, \mathtt{bind}(\mathtt{x}, \mathcal{E}[\![\mathtt{y}]\!]_{\mathtt{YH}^0}, \rho)) \\
&= \mathtt{bind}(\mathtt{y}, \mathtt{intoVal}(3), \mathtt{bind}(\mathtt{x}, \mathtt{find}(\mathtt{YH}^0, \mathtt{y}), \rho)) \\
&= \mathtt{bind}(\mathtt{y}, \mathtt{intoVal}(3), \mathtt{bind}(\mathtt{x}, \mathtt{intoVal}(3), \rho)) \\
\mathtt{Y\ H}^2 &= \mathtt{bind}(\mathtt{y}, \mathcal{E}[\![3]\!]_{\mathtt{YH}^1}, \mathtt{bind}(\mathtt{x}, \mathcal{E}[\![\mathtt{y}]\!]_{\mathtt{YH}^1}, \rho)) \\
&= \mathtt{bind}(\mathtt{y}, \mathtt{intoVal}(3), \mathtt{bind}(\mathtt{x}, \mathtt{find}(\mathtt{YH}^1, \mathtt{y}), \rho)) \\
&= \mathtt{bind}(\mathtt{y}, \mathtt{intoVal}(3), \mathtt{bind}(\mathtt{x}, \mathtt{intoVal}(3), \rho)) \\
\mathtt{Y\ H}^2 &= \mathtt{Y\ H}^1
\end{aligned}$$

Lecture 9-10: Solutions of the Exercises

prof. Marco Bellia, Dip. Informatica, Università di Pisa

March 28, 2014

Example

The following Haskell expression, `h 3 (f 5)`, when `h` e `f` are:

```
h = \x y -> if (x\=0) then x else y
f n = f(n+1)
```

evaluates to 3.

Can You rephrase it in OCaml, C or Java?

Lecture 9-10: slide 10: Solved

Example

The following Haskell expression, `h 3 (f 5)`, when `h` e `f` are:

```
h = \x y -> if (x\=0) then x else y
f n = f(n+1)
```

evaluates to 3.

Can You rephrase it in OCaml, C or Java?

Yes. We can do it in Ocaml by using the lazy-force emulation mechanism introduced in the language

```
# let u = lazy(3+5);;
val u:int lazy_t = <lazy>
# u;;
- :int lazy_t = <lazy>
# Lazy.force(u);;
- :int = 8
# let h = fun x y -> if (x!=0) then x else Lazy.force(y);; -- solution begins here
val h:int -> int Lazy.t -> int = <fun>
# let rec f n = f(n+1);;
val f:int -> 'a = <fun>
# h 3 (lazy(f 5));; -- solution ends here
- :int = 3
```

Example

The Haskell expression `g[4,(f 5)]`, when `h` and `f` are:

```
g u = if ((head u)==0) then 3 else 7
f n = f(n+1)
```

computes 7.

Can You rephrase it in OCaml, C or in Java?

Lecture 9-10: slide 14: Text

Example

Finitely Approximated, Infinite Values:

```
nat n = n:nat(n+1)
naturali = nat 0
v = take 3 naturali
```

In Haskell, the 3 expressions above, compute one function, the infinite list of naturals, the list of the first 3 naturals.
Can You rephrase it in Caml, C or Java?

Example

Finitary Infinite values:

`data Tree a = T(a,Tree a)` - it defines a polymorphic type of Haskell

`treeM = T(3,treeM)` - a value of Haskell

treeM computes a infinite tree that can be finitely represented (with pointers !?)
Can You rephrase it in Caml, C or Java?

Lecture 9-10: slide 14: Solved

Example

Finitely Approximated, Infinite Values:

```
  nat n = n:nat(n+1)
  naturali = nat 0
  v = take 3 naturali
```

In Haskell, the 3 expressions above, compute one function, the infinite list of naturals, the list of the first 3 naturals.
Can You rephrase it in Caml, C or Java?
Yes. We can do it in Ocaml but at some non maginal cost

```
# let rec nat n = n::(nat (n+1));;
val nat :  int -> int list = <fun>
# let naturali = nat 0;;
Stack overflow during evaluation (looping recursion?).
# let naturali = lazy(nat 0);;
val naturali :  int list lazy_t = <lazy> -- is it enough for our computation?
------------------- well!  we provide a definition of take in Ocaml
# exception TakeIllegalArgs;;
exception TakeIllegalArgs
# let take = fun n l -> if (List.length l) < n then (raise TakeIllegalArgs)
        else let rec itake n l = if (n=1) then (List.hd (Lazy.force l))
                                          else (itake (n-1) (lazy(List.tl (Lazy.force l))))
             in (itake n l);;
# take 3 naturali;;
Stack overflow during evaluation (looping recursion?).
-------------------- oh!  It is not enough.
-------------------- We need be able to partially compute the list constructor.  E.g.:
# type 'a myList = Nil | Cons of 'a * 'a myList lazy_t;;
type 'a myList = Nil | Cons of 'a * 'a myList lazy_t
# let rec nat n = Cons(2,lazy(nat(n+1)));;
val nat :  int -> int myList = <fun> -- we are finally, moving in the right way....
```

Example

Consider the following C expression:

```
z = x = y
```

The abstract syntax of it, resulting from the compiler or interpreter front-end, in the notation adopted in the provious slides is:

```
Val(Den(z) = Val(Den(x) = Val(y)))
```

Do the same with the following C expression:

```
A[*v+j] = x = y + A[*v+1]
....
```

Lecture9-10: Slide 19: Text

Example

Show the environment Env when the semantics of mutually applies to the fragment

```
Mut
  int x = y;
  final int y = 3;
Ally
```

Check definitions for bugs: (a) fix them; (b) motivate

Example

Semantic Functions

$$\mathcal{D}[\![\mathtt{D}]\!]_\rho : \mathtt{Store} \to (\mathtt{Env} \times \mathtt{Store})_\perp$$

$$\begin{aligned}
&\mathcal{D}[\![\mathtt{Var\ I;}]\!]_\rho \\
&\quad = \lambda s.\ \mathtt{Let}\{(l, s_l) = \mathit{allocate}(s)\}\ (\mathtt{bind}(\mathtt{I}, l, \rho), s) \\
&\mathcal{D}[\![\mathtt{Var\ I = E;}]\!]_\rho(s) \\
&\quad = \mathtt{Let}\{(v_e, s_e) = \mathcal{E}[\![\mathtt{E}]\!]_\rho(s)\}\ (\mathtt{bind}(\mathtt{I}, v_e, \rho), s_e)
\end{aligned}$$

Check definitions for bugs: (a) fix them; (b) motivate

Example

Semantic Functions

$$\mathcal{D}[\![\mathtt{D}]\!]_\rho : \mathtt{Store} \to (\mathtt{Env} \times \mathtt{Store})_\perp$$

$$\begin{aligned}
&\mathcal{D}[\![\mathtt{Var\ I};]\!]_\rho \\
&\quad = \lambda s.\ \mathtt{Let}\{(l, s_l) = \mathit{allocate}(s)\}\ (\mathtt{bind}(\mathtt{I}, l, \rho), s) \\
&\mathcal{D}[\![\mathtt{Var\ I} = \mathtt{E};]\!]_\rho(s) \\
&\quad = \mathtt{Let}\{(v_e, s_e) = \mathcal{E}[\![\mathtt{E}]\!]_\rho(s)\}\ (\mathtt{bind}(\mathtt{I}, v_e, \rho), s_e)
\end{aligned}$$

Lecture 11: Solutions of the Exercises

prof. Marco Bellia, Dip. Informatica, Università di Pisa

March 28, 2014

1. Complete with the suitable definitions in order to run the following Ocaml codes

Example

```
let x = ref 0 in
 let pippo xr =
      function n -> xr := !xr + n in
  let pippo1 = pippo(x) in
   pippo1(3);
   print(!x);
   (let x = ref 0 in
      pippo1(3);
      print(!x));
   print(!x);
```

```
let x = ref 0 in
 let pippo xr =
      function n -> xr := !xr + n in
  pippo(x)(3);
  print(!x);
  (let x = ref 0 in
     pippo(x)(3);
     print(!x));
  print(!x);
```

2. Give definitions for the domain Env: Values and Operations
3. Give definitions for the domain Store: Values and Operations

Lecture11: Slide 11: Solved 2

```
bind:  Ide x Den x Env -> Env
   abstract view:  bind(i,d,e) = fun u -> if (u=i) then d else e(u)  a function
 find:  Ide x Env -> (Den + Ide)
   abstract view:  find(i,e) =e(i)  a function application
empty:  () -> Env
   abstract view:
empty() = fun u -> u  identity

-- top down development in Ocaml
# type ide = Ide;;
type ide = Ide
# type den = Den | ID of ide;;
type den = Den | ID of ide
# type env = ide -> den;;
type env = ide -> den
# let empty = fun () -> fun (x:ide) -> ID x;;
val empty :  () -> ide -> den = <fun>
-- wrong
    # exception DenUndef;;
    exception DenUndef
    # let empty = fun () -> fun (x:ide) -> raise DenUndef;;
    val empty :  unit -> ide -> 'a = <fun> -- see the type
-- wrong because of the type (i.e.  is a generic 'a) and of the use context (i.e.  semantics)
# let bind (i,d,e) = fun u -> if (u=i) then d else e(u);;
val bind :  'a * 'b * ('a -> 'b) -> 'a -> 'b = <fun>
# let find = fun (i,e) -> e(i);;
val find :  'a * ('a -> 'b) -> 'b = <fun>
```