Lecture 18 — Interpreter for MiniOCaml

- To begin a deeper study of functional programming start with interpreters in the style of MPs 6 and 7. we'll discuss an interpreter based on *substitution*. Aft midterm, we'll discuss an *environment-based* approa
- Topics we will cover are:
  - Anonymous functions
  - Language simplifications
  - Substitution model (SOS rules)
  - MP9

Functional programming

- Two essential features of functional programming:
  - Recursion over lists and trees
  - Higher-order functions, a.k.a. first-class functions
    - Treat functions the same as other values: p
      arguments, put into lists and tuples, etc.
  - *We will concentrate on the latter for the nex*
    weeks.

Anonymous functions

- **OCaml has a syntactic way of defining functions w giving them a name:**

  `fun` x `->` exp

- **Denotes a function that takes x to the value of exp presumably contains x).**

```
(fun a -> a*a) 4
```
16

```
let addnl = fun s -> s ^ "\n" in addnl "abc"
```
"abc\n"

```
let funlist = [fun a -> a+1; fun a -> a*a]
in (hd (tl funlis)) 3
```
9

```
let funpair = (fun x -> x+1, fun y -> y+3)
in (fst funpair) 4 + (snd funpair) 5
```
13

```
let rec fac = fun x -> if x=0 then 1 else x * fac (x-1) in fac 5
```
120

Exercises with anonymous functi

- **Define a list of three functions: the first adds 1 to any integer; the second do integer; and the third triples any integer. Then apply the second function to 4**

```
let funlis =
  [ fun x ->  x+1      ; fun x -> x * 2      ; fun x -> x+3
in ( hd (tl funlis)          ) 4
```

- **Transform the following expression to the form** `let rec length = ...` **, an pattern-matching:**

```
let rec length lis = match lis with [] -> 0 | h::t -> 1 + lengt
in length [1;2;3]
```

let rec length = fun lis -> if lis = [] then 0
else 1 + length (tl lis)

Substitution semantics

- **The application of an anonymous function to its arg can be described easily in an SOS rule:**

$$\frac{\texttt{(fun } x \texttt{ -> } e\texttt{)} \;\; e' \;\Downarrow v}{\begin{array}{l} e' \Downarrow v' \\ e[v'/x] \Downarrow v \end{array}}$$

where $e[v'/x]$ means "replace all free occurrences of x v'." ("Free" means not introduced by a `let` or `fun` co within e)

Substitution semantics example

- `(fun x -> x+x) (3+4)` ⇓ 14

  3 + 4 ⇓ 7

  7 + 7 ⇓ 14

  $(x + x)[7/x]$

- `(fun x -> hd x + 1) (tl [3;4;5])` ⇓ 5

  tl [3;4;5] ⇓ [4;5]

  hd [4;5] + 1 ⇓ 5

  $(hd\ x + 1)[[4;5]/x]$

Multi-argument functions

- **Here is the obvious extension of the application r functions of two arguments:**

$$\texttt{(fun } x \ y \texttt{ -> } e\texttt{)} \ e' \ e'' \Downarrow v$$
$$e' \Downarrow v'$$
$$e'' \Downarrow v''$$
$$e[v'/x][v''/y] \Downarrow v$$

`(fun x y -> x*y) (3+4) (1+2)` $\Downarrow$ 21

$3+4 \Downarrow 7$

$1+2 \Downarrow 3$

$7*3 \Downarrow 21$

$(x*y)[7/x][3/y]$

Multi-arg functions unnecessar

- **Instead of the 2–argument rule, use two rules:**

$$e\ e' \Downarrow v$$
$$e \Downarrow \texttt{fun}\ x\ \texttt{->}\ e'$$
$$e' \Downarrow v'$$
$$e'[v'/x] \Downarrow v$$

$$\texttt{fun}\ x\ \texttt{->}\ e \Downarrow \texttt{fun}\ x\ \texttt{->}\ e$$

- **Now evaluate:**

```
((fun x -> (fun y -> x*y)) (3+4)) (1+2)
```

⇓ 21

(fun x → (fun y → x*y)) (3+4) ⇓ fun y → 7*y

fun x → fun y → x*y ⇓ fun x → fun y → x*y

3+4 ⇓ 7

fun y → 7*y ⇓ fun y → 7*y

(fun y → x*y)[7/x]

1+2 ⇓ 3

7*3 ⇓ 21

(7*y)[3/y]

Simplifying OCaml

- What we have just shown is that *multi-argument* *tions are unnecessary* — they're syntactic sugar. Si applying functions to multiple arguments (except for b operations like addition) is unnecessary.
- Anonymous functions allow another simplification: R `let` f x `=` e `in` e' as `let` f `=` `fun` x `->` e `in` e'. No *ery* `let` *expression has the form* `let` x `=` e `in` e'.
- (We will have to deal with recursion separately.)

Evaluation by substitution

- In this model of evaluation, we just simplify expre
  There is no set of "values" distinct from expressions (as
  was in the interpreters for MiniJava). Instead, expre
  that are simple enough play the role of values.
- To be more precise, these are the expressions cons
  "simple enough" to be values.
  - All constants — ints, floats, strings, `true`, `false`
  - Lists and tuples — $[v_1, \ldots, v_n]$ or $(v_1, \ldots, v_n)$ — *s*
    as $v_1, \ldots, v_n$ *are values.*
  - `fun` x `->` e — regardless of what e is
- If $e \Downarrow v$, then v is a value. (e must be a *closed* expres

Values (cont.)

- **Non-values are terms that should be further simplifi**
  and `let` expressions, applications, binary or unary term
  built-in operators (+, `hd`, etc.).

- **What about variables — how are they evaluated?**

  We never evaluate a variable because they are replaced by closed terms before they need to be evaluated. $e \Downarrow v$ only applies when e is closed.

SOS rules for evaluation

$i \Downarrow$ (i an integer constant)

$e_1 + e_n \Downarrow$

See slide 15, or MP9

`List [`e_1`; ...; `e_n`]` $\Downarrow$

Rules for evaluation by substitut (cont.)

if e_1 then e_2 then e_3) $\Downarrow$

if e_1 then e_2 then e_3) $\Downarrow$

See slide 15, or MP 9

fun x -> e $\Downarrow$

Rules for evaluation by substitut (cont.)

$e_1 \ e_2 \Downarrow$

See slide 15, or MP9

$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' \Downarrow$

Substitution model evaluation ru

(Const) Const x $\Downarrow$ Const x

(Rec) Rec(f,Fun(a,e)) $\Downarrow$ Fun(a,e[Rec(f,Fun(a,e))/f]

(δ) $e\ op\ e' \Downarrow v\ OP\ v'$
$e \Downarrow v$
$e' \Downarrow v'$

(If) If(e_1, e_2, e_3) $\Downarrow v$
$e_1 \Downarrow$ True
$e_2 \Downarrow v$

(List) $[e_1, \ldots, e_n] \Downarrow [v_1, \ldots, v_n]$
$e_1 \Downarrow v_1$
$\vdots$
$e_n \Downarrow v_n$

(Let) Let(a,e,e') $\Downarrow v'$
$e \Downarrow v$
$e'[v/a] \Downarrow v'$

(Fun) Fun(a,e) $\Downarrow$ Fun(a

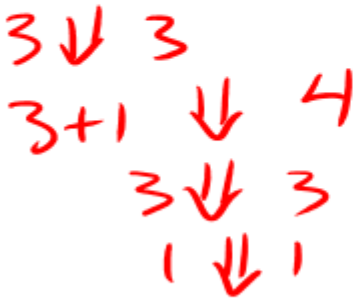
(δ) $op\ e \Downarrow OP\ v$
$e \Downarrow v$

(If) If(e_1, e_2, e_3) $\Downarrow v$
$e_1 \Downarrow$ False
$e_3 \Downarrow v$

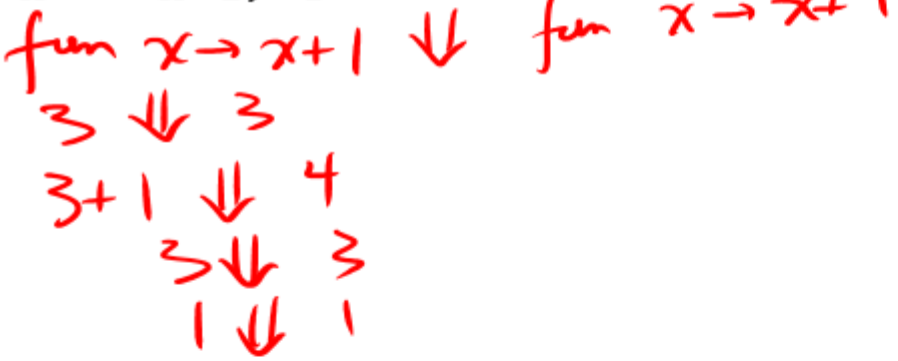
(App) $e\ e' \Downarrow v$
$e \Downarrow$ Fun(a, e'')
$e' \Downarrow v'$
$e''[v'/a] \Downarrow v$

Examples of evaluation by substitution

```
let x = 3 in x+1
```




```
(fun x -> x+1) 3
```



From here on, to save space, we omit evaluation steps of the form $e \Downarrow e$, and just write a line to indicate that a step is omitted.

Examples of evaluation by substitution

```
((fun x -> (fun y -> x+y)) 3) 4
```

$\Downarrow 7$

$(\text{fun } x \to (\text{fun } y \to x+y))\ 3 \Downarrow \text{fun } y \to 3+y$

$3+4 \Downarrow 7$

```
let f = fun x -> fun y -> x+y
in let g = f 1
   in g 2
```

$\text{let } g = (\text{fun } x \to \text{fun } y \to x+y)\ 1 \text{ in } g\ 2 \Downarrow 3$

$(\text{fun } x \to \text{fun } y \to x+y)\ 1 \Downarrow \text{fun } y \to 1+y$

$(\text{fun } y \to 1+y)\ 2 \Downarrow 3$

$1+2 \Downarrow 3$

Substitution — $e[v/x]$

- **Definition of substitution needs to be given precisely MP9. However, its meaning is intuitively clear: $e[v/x]$ to replace all occurrences of x in e by v. Just remembe if e contains a subexpression that binds x (a `let` or `fun` leave that subexpression alone.**

Recursion

- **The rules we've given do not handle `let rec`. W introduce a new type of expression (not actually in O to handle this:**

  `rec` f e

  where e is always a `fun` expression.

- **Rewrite `let rec` f x `=` e `in` e' as**

  `let` f `=` `rec` f `(fun` x `->` e`)` `in` e'

- **Add an SOS axiom for `rec`:**

  `rec` f `(fun` x `->` e`)` $\Downarrow$ `fun` x `->` e`[rec` f `(fun` x `->`

Example of evaluation by substitution

```
let map = fun f -> Rec(map1, fun lis -> if lis=[] then []
                                       else (f (hd lis)) :: map1 (tl lis))
in map (fun x->x+1) [1;2;3]
```

⇓ [2;3;4]

(fun f → Rec(map1, ...)) (fun x → x+1) [1;2;3] ⇓ [2;3;4]

(fun f → Rec(...)) (fun x → x+1) ⇓ fun lis → if lis=[] then []
else ((fun x→x+1)(hd lis))
:: (Rec(...)(tl lis))

Rec(map1, fun lis → if lis=[] then []
else ((fun x→x+1)(hd lis)) ⇓ fun lis → ...
:: map1 (tl lis)

⇓ [2;3;4]

if [1;2;3]=[] ... else
((fun x→x+1)(hd [1;2;3])) ::
(Rec(...)(tl [1;2;3]))

[1;2;3]=[] ⇓ false
((fun x→x+1)(hd [1;2;3])) :: (Rec(...)(tl [1;2;3])) ⇓ [2;3;4]

ETC.

MP9: Interpret MiniOCaml

- **Like OCaml, but without: pattern-matching; mutual sion ("and"); type definitions (only built-in types — floats, bools, strings, pairs, lists)**

- **Concrete syntax of MiniOCaml:**

  exp → **exp binary-operation exp** | **unary-operation exp**
  | **INTEGER_LITERAL** | **FLOAT_LITERAL** | **STRING_LITERAL** | **TRUE**
  | **LBRACK explis1 RBRACK** | **LPAREN explis2 RPAREN** | **LET def IN e**
  | **IDENT** | **IF exp THEN exp ELSE exp** | **LET REC def IN exp** | **exp exp**
  | **FUN IDENT RIGHTARROW exp**

  def → **IDENT args EQ exp**

  args → ϵ | **args IDENT**

  binary-operation → **COMMA** | **EQ** | **LT** | **GT** | **NEQ** | **ANDAND** | *etc.*

  unary-operation → **NOT** | **HD** | **TL** | **FST** | **SND** | *etc.*

  explis1 → *list of exp's separated by semicolons*
  explis2 → *list of exp's separated by commas*

Abstract syntax of MiniOCam

- **The abstract syntax incorporates the simplifications we**

```
type exp =
      Operation of exp * binary_operation * exp
    | UnaryOperation of unary_operation * exp
    | Var of string | StrConst of string | IntConst of int
    | FloatConst of float | True | False
    | List of exp list | Tuple of exp list
    | If of exp * exp * exp | App of exp * exp
    | Let of string * exp * exp
    | Fun of string * exp
    | Rec of string * exp

and binary_operation = Semicolon | Comma | Equals | LessThan
    | GreaterThan | NotEquals | Assign | And | Or
    | IntPlus | IntMinus | IntDiv | IntMult
    | FloatPlus | FloatMinus | FloatDiv | FloatMult
    | StringAppend | ListAppend

and unary_operation = Not | Head | Tail | Fst | Snd
```

MP 9 — Interpreter for dynamically-typed MiniOCaml

- We will provide lexer/parser and translation to al syntax
- You will write function `reduce` that maps ASTs to following the SOS rules.
- As in earlier MPs, we will provide precise evaluation (In MP9, SOS rules are given in terms of abstract syn
- You will interpret a *dynamically-typed* version. of O

Wrap-up

- **Today we discussed:**
  - Simplifying OCaml using anonymous functions
  - Evaluating expressions by substitution
  - MP9
- **We discussed them because:**
  - They are important in understanding how functional language
    and in particular, understanding *higher-order functions*.
- **In Thursday's class, we will:**
  - Discuss higher-order functions
- **What to do now:**
  - MP9