Lecture 18 — Interpreter for MiniOCaml

- **To begin a deeper study of functional programming, we'll start with interpreters in the style of MPs 6 and 7. First, we'll discuss an interpreter based on** *substitution*. **After the midterm, we'll discuss an** *environment-based* **approach.**

- **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: pass as arguments, put into lists and tuples, etc.
  - ***We will concentrate on the latter for the next two weeks.***

Anonymous functions

- **OCaml has a syntactic way of defining functions without giving them a name:**

  fun x -> exp

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

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

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

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

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

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

Exercises with anonymous functions

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

```
let funlis =
 [                          ;                          ;                  ]

in (                             ) 4
```

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

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

Substitution semantics

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

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

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

Substitution semantics examples

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

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

Multi-argument functions

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

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

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

Multi-arg functions unnecessary

- **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)
```

Simplifying OCaml

- What we have just shown is that *multi-argument functions are unnecessary* — they're syntactic sugar. Similarly, applying functions to multiple arguments (except for built-in operations like addition) is unnecessary.

- Anonymous functions allow another simplification: Rewrite `let` f x `=` e `in` e' as `let` f `=` `fun` x `->` e `in` e'. Now *every* `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 expressions. There is no set of "values" distinct from expressions (as there was in the interpreters for MiniJava). Instead, expressions that are simple enough play the role of values.**
- **To be more precise, these are the expressions considered "simple enough" to be values.**
  - **All constants — ints, floats, strings,** `true`, `false`
  - **Lists and tuples —** $[v_1, \ldots, v_n]$ **or** $(v_1, \ldots, v_n)$ **—** ***so long 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*** **expression.)**

Values (cont.)

- Non-values are terms that should be further simplified: `if` and `let` expressions, applications, binary or unary terms with built-in operators (+, `hd`, etc.).
- What about variables — how are they evaluated?

SOS rules for evaluation

$i \Downarrow$ $\quad$ (i an integer constant)

$e_1 \; + \; e_n \Downarrow$

$\texttt{List [}e_1\texttt{; ...; }e_n\texttt{]} \Downarrow$

Rules for evaluation by substitution (cont.)

`if` e_1 `then` e_2 `then` e_3`)` $\Downarrow$

`if` e_1 `then` e_2 `then` e_3`)` $\Downarrow$

`fun` x `->` $e \Downarrow$

Rules for evaluation by substitution (cont.)

$e_1\ e_2 \Downarrow$

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

Substitution model evaluation rules

(Const) Const x $\Downarrow$ Const x

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

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

(δ) $e\ op\ e' \Downarrow v\ OP\ v'$

$e \Downarrow v$

$e' \Downarrow v'$

(δ) $op\ e \Downarrow OP\ v$

$e \Downarrow v$

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

$e_1 \Downarrow$ True

$e_2 \Downarrow v$

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

$e_1 \Downarrow$ False

$e_3 \Downarrow v$

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

$e_1 \Downarrow v_1$

$\vdots$

$e_n \Downarrow v_n$

(App) $e\ e' \Downarrow v$

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

$e' \Downarrow v'$

$e''[v'/a] \Downarrow v$

(Let) Let(a,e,e') $\Downarrow v'$

$e \Downarrow v$

$e'[v/a] \Downarrow v'$

Examples of evaluation by substitution

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

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

Examples of evaluation by substitution

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

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

Substitution — $e[v/x]$

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

Recursion

- The rules we've given do not handle `let rec`. We will introduce a new type of expression (not actually in OCaml) to handle this:

  $$\texttt{rec}\ f\ e$$

  where e is always a `fun` expression.

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

  $$\texttt{let}\ f\ \texttt{=}\ \texttt{rec}\ f\ \texttt{(fun}\ x\ \texttt{->}\ e\texttt{)}\ \texttt{in}\ e'$$

- Add an SOS axiom for `rec`:

  $$\texttt{rec}\ f\ \texttt{(fun}\ x\ \texttt{->}\ e\texttt{)} \Downarrow \texttt{fun}\ x\ \texttt{->}\ e\texttt{[rec}\ f\ \texttt{(fun}\ x\ \texttt{->}\ e\texttt{)/}f\texttt{]}$$

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]
```

MP9: Interpret MiniOCaml

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

- **Concrete syntax of MiniOCaml:**

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

  def $\rightarrow$ **IDENT args EQ exp**

  args $\rightarrow \epsilon$ | **args IDENT**

  binary-operation $\rightarrow$ **COMMA | EQ | LT | GT | NEQ | ANDAND |** *etc.*

  unary-operation $\rightarrow$ **NOT | HD | TL | FST | SND |** *etc.*

  explis1 $\rightarrow$ *list of exp's separated by semicolons*
  explis2 $\rightarrow$ *list of exp's separated by commas*

Abstract syntax of MiniOCaml

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

```
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 abstract syntax
- You will write function `reduce` that maps ASTs to ASTs, following the SOS rules.
- As in earlier MPs, we will provide precise evaluation rules. (In MP9, SOS rules are given in terms of abstract syntax.)
- You will interpret a *dynamically-typed* version. of OCaml.

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 languages work, and in particular, understanding *higher-order functions*.
- **In Thursday's class, we will:**
  - Discuss higher-order functions
- **What to do now:**
  - MP9