Homework 8: Substitution and Evaluation of LFP

CIS 352: Programming Languages

22 March 2018, Version 1

Administrivia

- If you pick up an idea from someone or somewhere else, document it in your cover sheet.
- Turn in Part I in the CIS 352 submissions box.
- For the remaining problems, turn them in via Blackboard. Include: *(i)* the source files (`LFPbs.hs` with maybe a Problems 2+3+4 version and a Problem 5 version), *(ii)* the transcripts of test runs, and *(iii)* your cover sheet.

Grading Criteria

- The homework is out of 100 points.
- Unless stated otherwise, each programming problem is:
  - $\approx$ 70% correctness
  - $\approx$ 30% testing.
- Omitting your name(s) in the source code looses you 5 points.

Part I: Problems on Paper

❖ ***Problem 1 (20 points)*** ❖

For each of following LFP expression[1] mark each variable occurrence (correctly) as free or bound and for the bound ones, indicate what variable definition it is bound to.[2]

(a) $\textbf{let } y = y + 3 \textbf{ in } x + y$

(b) $\lambda f.(f\ (g\ x))$

(c) $\textbf{let } x = x + y \textbf{ in } (\textbf{let } y = y + 1 \textbf{ in } (\textbf{let } x = 3 + x \textbf{ in } x + y))$

(d) $\lambda w.\lambda x.(w((wx)w))$

(e) $\lambda x.\big(\textbf{let } p = (\lambda y.x + y) \textbf{ in } (\textbf{let } x = 3 + x \textbf{ in } (p\ x))\big)$

(f) $\lambda x.\big(\textbf{let } p = (\lambda x.x + y) \textbf{ in } (\textbf{let } x = 3 + x \textbf{ in } (p\ x))\big)$

(g) $\textbf{let } x = (\lambda x.x) \textbf{ in } (y\ (\lambda y.((x\ x)y)))$

(h) $\lambda g.((\lambda g.(g\ y))\ (\lambda y.(g\ y)))$

(i) $\textbf{let } y = (\textbf{let } x = x + 7 \textbf{ in } x + z) \textbf{ in } (\textbf{let } x = y \textbf{ in } (x + y))$

(j) $(\lambda x.((y\ (\lambda y.(y\ a))\ x)\ x))$

Example: $\textbf{let } x = y \textbf{ in } (\textbf{let } p = (\lambda y.\lambda z.x + y) \textbf{ in } (p\ x\ y))$

(i) label each variable definition with a unique number and

(ii) for each variable occurrence, either label it with the number of its definition, or else, if it is a free occurrence, label it *free*.

The result: $\textbf{let } x^1 = y^{free} \textbf{ in } \Big(\textbf{let } p^2 = (\lambda y^3.\lambda z^4.x^1 + y^3) \textbf{ in } (p^2\ x^1\ y^{free})\Big)$

[1] **N.B.** We'll use the standard LFP grammar for this problem.

[2] ***Hint:*** Drawing a parse tree of an expression is often helpful in figuring out the scopes of variable definitions.

Part II: Implementation Problems

Background

Here we implement the big-step semantics of Pitts' LFP language (Pitts, 2002), although we'll following Mayr and Stirling's (Mayr and Stirling, 2013a) version of LFP. We'll make use of the files:

`LFPas.hs`	LFP's abstract syntax & some utilities.
`State.hs`	the data structure for LC-states.
`LFPparser.hs`	a parser for LFP.
`LFPbs.hs`	a start at the implementation of the big-step interpreter. *Do all of your work in this file!*

Notes on the grammar To keep the parser simple, I've changed Pitts' grammar for LFP a bit.

- Locations now start with the *uppercase* character `'X'` followed by a number, e.g., `X0`, `X15`, etc.
- For λ-expressions, in place of $(\lambda x.e)$ we write `(fn x => e)`.
- Arithmetic expressions and comparisons are in prefix form rather than the traditional infix form.[3] For example:

infix	*prefix*
`12+9`	`(+ 12 9)`
`3<(4*2)`	`(< 3 (* 4 2))`

[3] The advantage of this is that *all* functions, whether built in (e.g., $+$, $*$, $<$) or defined by a user, have the same syntax for applications.

Note that, unlike LC, it is easy to write nonsense LFP expressions.[4]

[4] Which is part of the reason we have to forgo QuickCheck for this assignment.

Variables vs. names for variables Keep the following straight.

IN THE CONCRETE SYNTAX:

a variable is a nonempty string of lower case letters.

IN THE ABSTRACT SYNTAX OF `LFP.hs`:

a variable is is an `Expr`-expression `(Var x)`, where x is a Haskell `String` (of lowercase letters) that is the variable's name.

Problem 2's function `freeVar` returns a list of variable names. Problem 3's function `subst` has three arguments, the first and last being `Expr`-expressions and the middle one being a variable name (i.e., a `String`).

❖ ***Problem 2 (20 points)*** ❖

(a) ***(8 points)*** Complete the definition of the function

```
freeVars :: Expr -> [String]
```

that takes the abstract syntax of an LFP expression and returns a list of the names of the free variables of the expression. **Example:**

```
(freeVars (eparse "fn y => (* (+ x y) z)"))
```

should return `["x","z"]`.

Notes on `freeVars`

- The ordering of the variable names doesn't matter.
- For the definition of the set of free-variables of an LFP expression, see the slide "Free variables and closed expressions" in (Mayr and Stirling, 2013a) and (Pitts, 2002, Slide 39, page 51).
- The starting definition of `freeVars` in `LFPbs.hs` has all the boring cases done for you, just the exciting ones are left.
- `Data.List`'s "set"- and ordered-list-operations can be helpful here. See: `http://hackage.haskell.org/package/base-4.8.2.0/docs/Data-List.html#g:20`.

(b) ***(6 points)*** Test your definition by running each of `fvt1`, ..., `fvt5`, and see if you get the expected result.[5]

(c) ***(6 points)*** Come up with at least 3 original tests of `freeVars` and explain why the results you got are correct.

[5] You can run all of these at once via `runFvAll`.

❖ ***Problem 3 (20 points)*** ❖

(a) ***(8 points)*** Complete the definition of the function

```
subst :: Expr -> String -> Expr -> Expr
```

where `(subst e x p)` computes $e[p/x]$. For the λ-case, the definition of substitution is:

$$(\lambda y.E)[P/x] = \begin{cases} (\lambda y.E), & \text{if } x = y; \\ (\lambda z.E''), & \text{if } x \neq y, \text{ where} \\ & \quad z \text{ is a variable } \notin (FV(E) \cup FV(P) \cup \{ x \}), \\ & \quad E' = E[z/y], \text{ and} \\ & \quad E'' = E'[P/x]. \end{cases}$$

You can use `freeVars` from Problem 2 to help compute the $FV(\cdot)$'s. You can use `fresh` (defined in `LFPbs.hs`) to help find a suitable z. Also note that `subst` in `LFPbs.hs` all the boring cases done and you are left just the exciting ones.

(b) ***(6 points)*** Test your definition by running each of `st1`, ..., `st10`, and see if you get the expected result.[6]

(c) ***(6 points)*** Come up with at least 3 original tests of subst and explain why the results you got are correct.[7]

NOTATION. $e[p/x]$ means: the substitution of p for the variable with name x in expression e. This is defined in (Mayr and Stirling, 2013b) and (Pitts, 2002, page 52).

[6] You can run all of these at once via `runSubAll`. The tests `st1`, ..., `st10` have their expected results given in `LFPbs.hs`. Your answers may not exactly match these given results, but they should be *α-equivalent*.

Definition (α-equivalence): Two expressions are called α-equivalent iff any difference between the two expressions is only in the names of bound variables.

E.g., $\lambda a.\lambda b.(a(b\,c)) \equiv_\alpha \lambda x.\lambda y.(x(y\,c))$, but $\lambda a.\lambda b.(a(b\,c)) \not\equiv_\alpha \lambda a.\lambda b.((a\,b)c)$.

[7] See the *Notes on original tests* on the last page.

❖ ***Problem 4 (20 points)*** ❖

In this problem we implement the big-step, call-by-name semantics of LFP as given in (Mayr and Stirling, 2013a) and (Pitts, 2002, §5.2).

(a) ***(8 points)*** Complete the definition of the function

```
eval :: (Expr,State) -> (Expr,State)
```

where `(eval (e,s))` computes the big-step, call-by-name semantics of LFP. Note that `eval` in `LFPbs.hs` all the boring cases are done and you are left just the exciting ones—lucky you.

(b) ***(6 points)*** Test your definition by running each of `et1`, ..., `et10`, and see if you get results that are α-equivalent to the expected result.[8]

(c) ***(6 points)*** Come up with at least 3 original tests of `eval` and explain[9] why the results you got are correct.

[8] You can run all of these at once via `runEvaAll`.

[9] See the *Notes on original tests* below.

❖ ***Problem 5 (20 points)*** ❖

In this problem we implement the call-by-value version of LFP as described in (Mayr and Stirling, 2013a) and in (Pitts, 2002, §5.2).

Note: For this problem, just comment out your call-by-name definitions for the application- and let-cases of `eval` from the previous problem.

(a) ***(8 points)*** Modify the application- and let-cases of `eval` to implement the call-by-value version of LFP.

(b) ***(6 points)*** Test your definition by again running each of `et1`, ..., `et10`, and see if you get results that are α-equivalent to the expected result.[10]

(c) ***(6 points)*** Come up with at least 3 original tests of this new version of `eval` and explain[11] why the results you got are correct. *Your tests should produce different results from their call-by-name versions!!!*

[10] You can run all of these at once via `runEvaAll`.

[11] See the *Notes on original tests* below.

Notes on original tests.

- The examples of part (c) of Problems 3, 4, and 5 can be "along the lines of" the part (b) examples. Examples that are too close to given examples or too simple (e.g., (+ 1 1) evaluating to 2) will loose points.
- The explanations of why your part (c) results are correct should be put in your source code next to the example they are explaining.

References

R. Mayr and C. Stirling. CS3 language semantics and implementation: Slides from lectures 13–16. Technical report, School of Informatics, University of Edinburgh, 2013a. URL http://www.inf.ed.ac.uk/teaching/courses/lsi/13Lsi13-14-nup.pdf.

R. Mayr and C. Stirling. CS3 language semantics and implementation: Slides from lectures 11 and 12. Technical report, University of Edinburgh, 2013b. URL http://www.inf.ed.ac.uk/teaching/courses/lsi/13Lsi11-12-nup.pdf.

A. Pitts. Lecture notes on semantics of programming languages: For part IB of the Cambridge CS tripos. Technical report, University of Cambridge, 2002. URL http://www.cl.cam.ac.uk/teaching/2001/Semantics/.