Advanced Programming Languages — Assignment 3
CS 6110 S16 — Due Friday, 18 March 2016

Instructions: *You may complete this assignment with one other CS 6110 student. You must register with your partner on CMS before you begin working together on the assignment. With the exception of your CMS-registered partner you should not give or receive assistance on this assignment. If slip days are used, they must be used by both partners. In addition, please limit use of outside resources to the lecture notes and textbooks for this course. If you have any questions about what is allowed and what is not allowed, please contact the course staff.*

1. Static and Dynamic Scope

   What is the value of the following FL program under call by value with (a) static scoping and (b) dynamic scoping? Briefly justify your answers.

   $\mathsf{let}\ x = 3\ \mathsf{in}$
   $\mathsf{let}\ f = \lambda y.\, x + y\ \mathsf{in}$
   $\mathsf{let}\ x = 5\ \mathsf{in}$
   $\mathsf{let}\ g = \lambda z.\, \mathsf{let}\ x = f\, x\ \mathsf{in}\ f\, x\ \mathsf{in}$
   $\mathsf{let}\ x = 7\ \mathsf{in}$
   $\mathsf{let}\ y = 9\ \mathsf{in}$
   $g\, y + f\, x$

2. Mutual Fixpoints

   The FL language has mutually recursive functions defined by

   $$\mathsf{letrec}\ f_1 = \lambda x.\, e_1\ \mathsf{and}\ \cdots\ \mathsf{and}\ f_n = \lambda x.\, e_n\ \mathsf{in}\ e.$$

   Each of the bodies e_i may refer to any of the f_j. Thus, the functions may call one another recursively. To translate these definitions to the λ-calculus, we need mutual fixpoint operators Y_i^n, $1 \le i \le n$, such that for any $F_1, \ldots, F_n$, the terms $\mathsf{Y}_i^n\, F_1\, F_2\, \ldots\, F_n$, $1 \le i \le n$ are mutual fixpoints of the F_i.

   (a) Find λ-terms Y_1^2 and Y_2^2 such that for any λ-terms F and G, $\mathsf{Y}_1^2\, F\, G$ and $\mathsf{Y}_2^2\, F\, G$ satisfy:

   $$\begin{aligned} \mathsf{Y}_1^2\, F\, G &= F\, (\mathsf{Y}_1^2\, F\, G)\, (\mathsf{Y}_2^2\, F\, G) \\ \mathsf{Y}_2^2\, F\, G &= G\, (\mathsf{Y}_1^2\, F\, G)\, (\mathsf{Y}_2^2\, F\, G) \end{aligned}$$

   Briefly justify your answer.

   (b) **Karma Problem:** Generalize your solution to part to Y_i^n, $1 \le i \le n$.

3. State

   In lecture we claimed that FL! programs never generate dangling references during evaluation. In this exercise, you will prove this fact formally. Consider the fragment of FL! consisting of the following expressions and values:

   $$\begin{aligned} e &::= n \mid x \mid \mathsf{ref}\, e \mid\ !e \mid e_1 := e_2 \mid e_1\ ;\ e_2 \mid \mathsf{null} \mid \lambda x.\, e \mid e_0\, e_1 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid (e_1, e_2) \mid \#n\, e \\ v &::= n \mid (v_1, v_2) \mid \mathsf{null} \mid \lambda x.\, e \end{aligned}$$

   To define the small-step operational semantics for FL!, we need to augment the set of expressions and values with locations:

   $$\begin{aligned} e &::= \ldots \mid \ell \\ v &::= \ldots \mid \ell \end{aligned}$$

A store σ is a partial map from locations to values and we define the small-step semantics of FL! programs in terms of configurations $\langle e, \sigma \rangle$ where e is an augmented expression and σ is a store. We expect that locations will only be generated during the evaluation of a program (to enforce this expectation, the parser would reject a program if it contained an explicit location).

Let $loc(e)$ denote the set of locations in e. For example, $loc(!\ell_2\,(\lambda x.\,!\ell_1\,!(\mathsf{ref}\,\ell_3))) = \{\ell_1, \ell_2, \ell_3\}$. Thus, if e is a program then $loc(e) = \{\}$.

(a) Give an inductive definition of $loc(e)$.

(b) Consider the following FL! configuration:

$$\langle \lambda x.\,((!\ell_1)\,2)\,(\mathsf{ref}\,1),\ \{\}[\ell_1 \mapsto \lambda y.\,\mathsf{ref}\,y]\rangle$$

Give each step of evaluation starting from this configuration and, for each step $\langle e', \sigma' \rangle$ show $loc(e')$. You can use the notation $\{\}$ to denote the empty store and $\sigma[\ell \mapsto v]$ to denote the extended store that maps ℓ to v and otherwise behaves like σ.

(c) Prove the following fact: if e is an FL! program and $\langle e, \{\}\rangle \to^* \langle e', \sigma' \rangle$ then $loc(e') \subseteq loc(\sigma')$. If you use induction (and you should :-) identify the relation you are doing induction on and argue briefly that it is well-founded.

4. Control Flow

Writing programs that deal gracefully with failure can be difficult in the usual imperative style, especially in a distributed environment where failure is unavoidable. A failed subcomputation can leave the system in an unpredictable state where important invariants are violated. Transactions are one way to deal with this. Consider the following extension to FL!:

$$e ::= \ldots \mid \mathsf{transaction}\,e \mid \mathsf{abort}\,e$$

The idea is that a transaction expression evaluates e and if e evaluates successfully, the whole expression has the same result as e. However, if the evaluation of e leads to the evaluation of an expression $\mathsf{abort}\,e'$, then the most recently started transaction halts and its result is the result of evaluating e'.

Since this is an extension to FL!, in general, the state will be modified during evaluation. When a transaction completes successfully, the resulting state is the same as the state after evaluating e. However, when a transaction is aborted, the state reverts to the state that existed at the beginning of the transaction.

Transactions can be nested inside other transactions, so an abort only aborts the innermost ongoing transaction.

(a) Give an operational semantics for this language. You may uniformly lift the rules for constructs in FL and FL!. You may assume the fragment of FL! from Exercise 3.

(b) Give a state-passing translation from this extended language into FL.

(c) Writing a formal semantics for this mechanism exposes some ambiguities in the above description, which entails making some design choices. Which choices, if any, came up when you were formalizing the semantics? How did you resolve them, and where do these choices show up in your formal semantics? Discuss briefly.

5. Closure Conversion

In this programming exercise, you will demonstrate that nested λ-expressions are not essential to the expressiveness of FL. Compilers for functional languages often translate nested λ-expressions away because the intermediate languages they use as a target (e.g., C or assembly) do not support them.

We will define two versions of the FL language, a source version and a more restricted target version. The source language is defined by the following grammar:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x_1, \ldots, x_n . e \mid e_1\, e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ & & \mid\ \mathsf{letrec}\ f = e_1\ \mathsf{in}\ e_2 \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid n \mid \mathsf{true} \mid \mathsf{false} \\ & & \mid\ e_1 + e_2 \mid e_1 \leq e_2 \mid \ \ldots \text{ (other arithmetic and boolean expressions)} \end{array}$$

The target language only allows functions to be declared only at the top level of a program p, and only in a very restricted form.

$$\begin{array}{rcl} p & ::= & \mathsf{let}\ h = \lambda x_1, \ldots, x_n . e\ \mathsf{in}\ p \mid \mathsf{letrec}\ h = \lambda x_1, \ldots, x_n . e\ \mathsf{in}\ p \mid e \\ e & ::= & x \mid e_1\, e_2 \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid n \mid \mathsf{true} \mid \mathsf{false} \\ & & \mid\ e_1 + e_2 \mid e_1 \leq e_2 \mid \ \ldots \text{ (other arithmetic and boolean expressions)} \end{array}$$

Note that expressions e are λ-*free*—they may not contain λ-abstractions, let, or letrec. In effect, all λ-abstractions have been *lifted* to the outermost level. Moreover, these named λ-abstractions must be closed and their bodies λ-free.

Your job is to write an OCaml function to convert the source language into the target language. The basic idea is to convert each nested λ-abstraction, let expression, and letrec expression to a closed function F that takes the values of its free variables $a_1, \ldots, a_n$ as extra arguments. The function F will be bound to a fresh variable f in the top-level program. Each occurrence of the original function in e can be replaced by a function call $(f\, a_1 \cdots a_n)$. This transformation needs to be performed inside out (i.e., working from the smallest expressions first). It produces the translated expression as well as and an environment mapping the (fresh) names of functions f to λ-free expressions e' of the target language.

For example, consider the following FL program with some nested function definitions.

```
let add3 =
  let compose f g x = f (g x) in
  compose (fun x -> x + 1) (fun x -> x + 2) in
add3 10
```

Here is a sample session with our solution code.

```
FL version 2016.0
>> load examples/test.txt
>> list
let add3 = let compose f g x = (f (g x)) in
((compose fun x -> (x + 1)) fun x -> (x + 2)) in
(add3 10)
>> run
13
>> lift
>> list
let a = fun add3 -> (add3 10) in
let b = fun x -> (x + 1) in
let c = fun x -> (x + 2) in
let d = fun c b compose -> ((compose b) c) in
let e = fun f g x -> (f (g x)) in
(a (((d c) b) e))
>> run
13
>> quit
bye
```

The `lifting.zip` file contains an FL interpreter, some useful low-level helper functions, and some sample FL programs. We have provided tools for creating fresh variables avoiding another set of variables, for finding all free variables and all variables in an expression, and safe substitution if you need it. The `State` module represents environments and defines appropriate lookup and update functions.

There are two functions `convert : exp -> state -> exp * state` and `lift : exp -> exp` in the file `lifting.ml` that you will have to implement. The function `convert` should implement the closure conversion translation as described above, producing a λ-free expression e' and an environment σ. The function `lift` should invoke `convert` to obtain e' and σ, then construct the final output program in the target language. The resulting program should run under the FL interpreter provided.

6. Debriefing

   (a) How many hours did you spend on this assignment?

   (b) Would you rate it as easy, moderate, or difficult?

   (c) Did everyone in your study group participate?

   (d) How deeply do you feel you understand the material it covers (0%–100%)?

   (e) If you have any other comments, we would like to hear them! Please write them here or send email to `jnfoster@cs.cornell.edu`.