CS 6110 Spring 2017 — Assignment 3

Advanced Programming Languages — Due: Wednesday, 22 March 2017

Instructions: You may complete this assignment with another CS 6110 student. You must form a group with your partner on CMS before you begin working together on the assignment. With the exception of the course staff and your CMS partner, you should not give or receive assistance on this assignment. In addition, please limit use of resources to the lecture notes 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.

Consider the following FL program under call-by-value evaluation:

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

(a) What is the value of this program with static scoping? Briefly (and informally) justify your answer.

(b) What is the value of this program with dynamic scoping? Again, briefly justify your answer.

2. Mutual fixed points.

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, so the functions can call one another recursively.

To translate ordinary, non-mutual recursion, we can use a fixed-point combinator such as Y or its call-by-value variant, Z. To translate these letrec definitions to the λ-calculus, however, we need *mutual* fixed-point operators. The idea is to write Y^n_i, $1 \le i \le n$, such that for any $F_1, \ldots, F_n$, the terms $\mathsf{Y}^n_i\, F_1\, F_2\, \ldots\, F_n$, $1 \le i \le n$ are mutual fixed points of the F_i terms. Each F_i should be a function that takes n other functions as arguments; when Y^n_i is applied to them, it should produce the "recursive version" of F_i.

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

$$\mathsf{Y}^2_1\, F\, G =_\beta F\, (\mathsf{Y}^2_1\, F\, G)\, (\mathsf{Y}^2_2\, F\, G)$$
$$\mathsf{Y}^2_2\, F\, G =_\beta G\, (\mathsf{Y}^2_1\, F\, G)\, (\mathsf{Y}^2_2\, F\, G)$$

(Here, $=_\beta$ denotes "full" β-equivalence, so you do not need to assume any particular evaluation order.) Briefly justify your answer.

(b) **Karma problem (zero points):** Generalize your solution to Y^n_i, $1 \le i \le n$.

3. State.

In lecture, we defined a language FL! that adds references to FL. In the semantics, the store σ is a partial function that maps locations ℓ to values. If a dereference expression $!\ell$ were to run when $\ell \notin \mathsf{dom}(\sigma)$, evaluation would get stuck. In this exercise, you will prove that such "dangling references" are impossible.

Consider this simplified version of FL!:

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

We define the small-step semantics of FL! programs in terms of configurations $\langle e, \sigma \rangle$ where e is an expression and σ is a store. There is a context rule:

$$\frac{\langle e, \sigma \rangle \to \langle e', \sigma' \rangle}{\langle E[e], \sigma \rangle \to \langle E[e'], \sigma' \rangle}$$

along with computation rules (all axioms) that can be found in the notes for Lecture 11 (base FL, where all the rules can be naturally extended to work on configuration pairs) and Lecture 13 (FL!).

We will assume that "source" programs will not contain locations ℓ, i.e., the parser would reject them. Locations only arise during the small-step evaluation.

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

Our goal is to prove that, when a source program executes to any configuration $\langle e, \sigma \rangle$, it always holds that $\mathrm{loc}(e) \subseteq \mathsf{dom}(\sigma)$. In other words, executing a source program can never create a dangling reference anywhere in the expression.

(a) Give an inductive definition of $\mathrm{loc}(e)$.

(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 $\mathrm{loc}(e')$. You can use the notation $\{\ell_1 \mapsto v_1, \ell_2 \mapsto v_2, \dots\}$ to denote stores σ.

(c) As is often the case when proving properties about executions, we will need to *strengthen* our criterion that defines safe configurations. Specifically, we need to rule out dangling references that occur *in the store* as well as in the expression.

To this end, we define $\mathrm{loc}(\sigma) \stackrel{\triangle}{=} \{\sigma(\ell) \mid \ell \in \mathsf{dom}(\sigma)\} \cap \mathit{Loc}$ to get the locations stored *as values* in an environment. For example, $\mathrm{loc}(\{\ell_1 \mapsto 42, \ell_2 \mapsto \ell_3\}) = \{\ell_3\}$.

For any FL! configuration $\langle e, \sigma \rangle$, let's call the configuration *safe* if and only if $\mathrm{loc}(e) \cup \mathrm{loc}(\sigma) \subseteq \mathsf{dom}(\sigma)$. In other words, safe configurations are free of dangling references. Prove this lemma, which says that small-step evaluation preserves safety:

Lemma. *If* $\mathrm{loc}(e) \cup \mathrm{loc}(\sigma) \subseteq \mathsf{dom}(\sigma)$ *and* $\langle e, \sigma \rangle \xrightarrow{1} \langle e', \sigma' \rangle$, *then* $\mathrm{loc}(e') \cup \mathrm{loc}(\sigma') \subseteq \mathsf{dom}(\sigma')$.

Hint: You will want to use structural induction over the small-step semantics. In the cases for this proof, only four will be "interesting." You may argue briefly (but convincingly!) why the others are trivial rather than reasoning about each individually.

You may assume these three facts in your proof:

1. There is a definition of $\mathrm{loc}(E[\cdot])$ for evaluation contexts such that $\mathrm{loc}(E[e]) = \mathrm{loc}(E[\cdot]) \cup \mathrm{loc}(e)$.
2. $\mathrm{loc}(\sigma(l)) \subseteq \mathrm{loc}(\sigma)$, i.e., the locations for a single value in a store are contained within the set of all locations in the entire store.
3. If $\langle e, \sigma \rangle \xrightarrow{1} \langle e', \sigma' \rangle$, then $\sigma \subseteq \sigma'$. In other words, evaluation steps never delete labels from the store (because we have no garbage collection!).

(d) Now use the lemma to prove this theorem, which says that source programs (which may not contain locations) can never create dangling references.

Theorem. *If* $\mathrm{loc}(e) = \varnothing$ *and* $\langle e, \{\} \rangle \xrightarrow{*} \langle e', \sigma' \rangle$, *then* $\mathrm{loc}(e') \subseteq \mathsf{dom}(\sigma')$.

Hint: With the lemma already in place, this proof is considerably shorter. You can prove it by mathematical induction on the number of small steps. Specifically, read $\langle e, \sigma \rangle \xrightarrow{*} \langle e', \sigma' \rangle$ as $\langle e, \sigma \rangle \xrightarrow{n} \langle e', \sigma' \rangle$ for all $n \geq 0$ and induct on n.

4. 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 target language (e.g., C or assembly) does 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 this grammar:

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

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

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

In the target language, 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.

To do the translation, we first 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 at the top level. The occurrence of the original function in e is replaced by a function call $f\, a_1 \cdots a_n$. This computation is done from the inside out (i.e., smallest subterms first). The result of the first step costs of a set of function bindings σ and a λ-free expression e' in the target language.

Here are the formal rules that govern this translation for variables, application, and λ-abstraction with one variable (the remaining constructs are up to you). Here, $\mathcal{L}[\![e]\!]\sigma$ represents the result of translating e with prior bindings σ. The value of this expression is a pair $\langle e', \sigma' \rangle$, where e' is e with all nested functions extracted and σ' is the new environment with the new function bindings appended to σ.

$$\mathcal{L}[\![x]\!]\sigma = \langle x, \sigma \rangle \qquad \frac{\mathcal{L}[\![e_1]\!]\sigma = \langle e_1', \sigma' \rangle \qquad \mathcal{L}[\![e_2]\!]\sigma' = \langle e_2', \sigma'' \rangle}{\mathcal{L}[\![e_1\, e_2]\!]\sigma = \langle e_1'\, e_2', \sigma'' \rangle}$$

$$\frac{\mathcal{L}[\![e]\!]\sigma = \langle e', \sigma' \rangle}{\mathcal{L}[\![\lambda x . e]\!]\sigma = \langle f\, a_1 \cdots a_n, \sigma'[\lambda a_1, \ldots, a_n, x . e'/f] \rangle}\ (FV(\lambda x . e') = \{a_1, \ldots, a_n\},\ f \text{ is fresh})$$

Most of the action is in the last rule. Given a λ-abstraction $\lambda x . e$, we recursively extract the functions in the body e, leaving the λ-free term e'. These functions are bound to fresh variables and the bindings are appended to σ to get σ'. That happens in the premise $\mathcal{L}[\![e]\!]\sigma = \langle e', \sigma' \rangle$. Then, we need to lift $\lambda x . e'$ itself. We find the free variables $a_1, \ldots, a_n$ and add them to the front of the function's argument list to form $\lambda a_1, \ldots, a_n, x . e'$, which is now a closed term. This we bind to a fresh variable f and insert into the environment to get $\sigma'[\lambda a_1, \ldots, a_n, x . e'/f]$. In place of the original λ-expression, we put $f\, a_1 \cdots a_n$.

Once all the functions are converted, the final step is to construct a complete program in the target language. The final program will consist of a series of let expressions reflecting the bindings in σ followed by the λ-free term e.

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 2017.0
>> load examples/test.fl
>> 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 that avoid another set of variables, for finding all free variables or 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 you need to implement in `lifting.ml`:

- `convert : exp -> state -> exp * state`. Here, you will implement the closure conversion translation described above, producing a λ-free expression e' and an environment σ.
- `lift : exp -> exp`. This function should invoke `convert` to obtain e' and σ, then construct the final output program in the target language by creating `let` expressions for all the bindings. The resulting program should run under the FL interpreter provided.

Karma extension (worth zero points): Add support for `let rec` expressions to support recursion. You will also need to *emit* `let rec` bindings in the output for these recursive definitions. The parser and evaluator already have support for recursion, and a few of the examples use `let rec`; you just need to implement the cases for `Letrec` AST nodes in your `convert` function and produce `Letrec` bindings appropriately in your `lift` function.

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 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 `asampson@cs.cornell.edu`.