CS 4110 – Programming Languages and Logics
Homework #9

ChangeLog

- Version 1 (Wednesday, November 19): Initial release.
- Version 2 (Saturday, November 22): Fixed typo in description of languages in Exercise 2: void changed to () and null removed.

Due Tuesday, November 25, 2014 by 11:59pm

Instructions This assignment may be completed with one partner. You and your partner should submit a single solution on CMS. Please do not offer or accept any other assistance on this assignment. Late submissions will not be accepted.

Exercise 1. Naive implementations of recursive functions often redundantly compute the same results many times. For example, to evaluate *fib* 5, the value *fib* 2 is computed 3 times!

One idea for making recursive functions more efficient is to cache and reuse previously-computed results. This technique is known as *memoization*. To memoize *fib* we can build a function that takes an argument n and checks if *fib* n is in the cache. If so, it returns the result immediately. Otherwise it computes the result, stores it in the cache, and returns it. As an example, here is a memoized version of Fibonacci in OCaml that uses an hashtable for the cache:

```
let cache : (int,int) Hashtbl.t = Hashtbl.create 43
let rec fib n =
  try Hashtbl.find cache n with Not_found ->
    let r =
      if n = 0 then 1
      else if n = 1 then 1
      else fib (n-1) + fib (n-2) in
    Hashtbl.add cache n r;
    r
```

In this exercise you will build a memoized version of your implementation of *fib* using references. Because we do not have hashtables, you will have to implement the cache using references: the first time a result is computed, you should overwrite the reference at the top-level with one containing a new function that returns that result if invoked on the same argument in the future and otherwise computes *fib* n.

We have provided some code to help you get started. Note that there are two levels of references—one to encode recursion and one to help you encode the cache.

```
let $fib$ : $\mathbf{int} \to \mathbf{int}$ =
  let $f$ : $((\mathbf{int} \to \mathbf{int})\ \mathbf{ref})\ \mathbf{ref}$ = ref (ref ($\lambda n\!:\!\mathbf{int}.\ 42$)) in
  $f$ := ref ($\lambda n\!:\!\mathbf{int}.$
              let $r\!:\!\mathbf{int}$ =
                if $n = 0$ then $1$
                else if $n = 1$ then $1$
                else $!(!f)\ (n-1)\ +\ !(!f)\ (n-2)$ in
                $\boxed{\textit{(* Fill in missing code here *)}}$
  );
  $!(!f)$
```

Exercise 2. In this exercise, we will show that sum types $\tau + \sigma$ can be encoded using product and universal types in System F. The propositions-as-type principle will help in constructing the appropriate translations. In particular, we will use one of De Morgan's laws: $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$.

As the source language, we will use the simply-typed λ-calculus extended with sums and unit:

$$\begin{aligned} e &::= () \mid x \mid e_1\, e_2 \mid \lambda x : \tau.\, e \mid \mathsf{inl}_{\tau_1+\tau_2} e \mid \mathsf{inr}_{\tau_1+\tau_2} e \mid \mathsf{case}\ e_0\ \mathsf{of} e_1 \ \mid\ e_2 \\ \tau &::= \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \end{aligned}$$

As the target language, we will use System F extended with products and unit (but not sums!):

$$\begin{aligned} e &::= () \mid x \mid e_1\, e_2 \mid \lambda x : \tau.\, e \mid (e_1, e_2) \mid \#1\, e \mid \#2\, e \mid \Lambda\alpha.\, e \mid e\,[\tau] \\ \tau &::= \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \alpha \mid \forall\alpha.\tau \end{aligned}$$

(a) Give a formula that is equivalent to $\phi \vee \psi$, but which only contains logical operators for which there are corresponding types in the fragment of System F above. (Hint: use a universal type to encode "negation".)

(b) Define a translation $\mathcal{T}[\![\cdot]\!]$ on types that takes a type in the fragment of simply-typed λ-calculus listed above and produces a type in the fragment of System F extended with products.

(c) Define the corresponding translation $\mathcal{E}[\![\cdot]\!]$ on expressions in the fragment of simply-typed λ calculus listed above. You only need to give the cases for $\mathsf{inl}_{\tau_1+\tau_2}$, $\mathsf{inr}_{\tau_1+\tau_2}$, and $\mathsf{case}\ e_0\ \mathsf{of}\ e_1 \ \mid\ e_2$.

You may find it helpful to think about types. In particular, your translations should be type preserving in the sense that they should map well-typed terms to well-typed terms. However, you do not need to prove this property.

Exercise 3. Suppose we extend Featherweight Java with support for simple exceptions:

$$e \quad ::= \quad x \mid e.f \mid e.m(\overline{e})\texttt{new}\ C(\overline{e}) \mid (C)\ e \mid \texttt{throw e} \mid \texttt{try}\ \{e\}\ \texttt{catch}\ (C\ x)\ \{\ e\ \}$$

The typing rules for these new expressions are as follows:

$$\text{T-THROW}\ \frac{\Gamma \vdash e : C \qquad C \leq \texttt{Exception}}{\Gamma \vdash \texttt{throw}\, e : D} \qquad\qquad \text{T-TRY}\ \frac{\Gamma \vdash e_1 : D \qquad \Gamma, x : C \vdash e_2 : D}{\Gamma \vdash \texttt{try}\, e_1\ \texttt{catch}(C\ x)\ \{e_2\} : D}$$

(a) Extend the definition of evaluation contexts E and the small-step operational semantics to propagate exceptions, using expressions of the form `throw new` $\mathrm{C}(\overline{v})$ to represent exceptions that have been thrown. You only need to give the new evaluation contexts and rules.

(b) Extend the operational semantics so that down casts step throw a `ClassCastException` instead of getting stuck. (You may assume that the program P already contains suitable definitions of classes `Exception` and `ClassCastException`.)

(c) State the progress theorem. (**Karma:** prove it!)

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 send email to `jnfoster@cs.cornell.edu`.