Homework 3: Products, Sums, and Church Encodings

15-814: Types and Programming Languages
Fall 2017
Instructor: Karl Crary
TA: Yong Kiam Tan

Out: Oct 09, 2017 (05 pm)
Due: Oct 16, 2017 (11 pm)

Notes:

- Welcome to 15-814's third homework assignment!
- Please email your work as a PDF file to `yongkiat@cs.cmu.edu` titled "15-814 Homework 3". Your PDF should be named "<your-name>-hw3-sol.pdf".
- This is a short homework and it is due in **one week**.

1 Fibonacci Numbers

The Fibonacci sequence, $0, 1, 1, 2, 3, 5, \cdots$ may be generated as follows:

$$f(n) = \begin{cases} n & \text{if } n \leq 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

In class, we saw a general strategy for implementing well-founded recursion using n-ply unfolded recursive definitions. For many problems, one can also get by without that technique. For example, it is well known that the Fibonacci numbers can be computed by maintaining only the two preceding values in the sequence.

Task 1 *Define the function f in the language* **LNS** *from Homework 2, without using well-founded recursion. You will want to make use of pairs.*

Task 2 *Define the function f in System* **T***, using well-founded recursion.*

2 Isomorphisms

In class, we saw that `void` is the unit for sums, and `unit` is the unit for products. This was made more precise by the notion of an isomorphism between types. In this section, we shall similarly define functions between two types[1] For the base language, we will use the simply-typed lambda calculus extended with products and sums. Its syntax is given as follows, and we shall use eager semantics for products and sums following PFPL 10-11.

$$\begin{array}{rcl} \tau & ::= & \texttt{unit} \mid \texttt{void} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \\ e & ::= & \star \mid \texttt{abort}[\tau](e) \mid \lambda x{:}\tau.e \mid e\ e \mid \langle e, e \rangle \mid \pi_1 e \mid \pi_2 e \mid \texttt{inl}(e) \mid \texttt{inr}(e) \mid \texttt{case}\ e\ \{x.e; x.e\} \end{array}$$

[1] Some of these will not actually be isomorphisms.

Task 3 *For each of the following pairs of types τ_1, τ_2, define expressions f, g with types $f : \tau_1 \to \tau_2$ and $g : \tau_2 \to \tau_1$ respectively.*

1. *(Currying)* $\tau_1 \to (\tau_2 \to \tau_3)$, $(\tau_1 \times \tau_2) \to \tau_3$
2. *(Permutation)* $(\tau_1 + \tau_2) + \tau_3$, $(\tau_2 + \tau_3) + \tau_1$
3. *(Distributivity)* $\tau \times (\tau_1 + \tau_2)$, $(\tau \times \tau_1) + (\tau \times \tau_2)$
4. *(De Morgan)* $(\tau_1 + \tau_2) \to \texttt{void}$, $(\tau_1 \to \texttt{void}) \times (\tau_2 \to \texttt{void})$
5. *(Soundness)* $\tau \times (\tau \to \texttt{void})$, $\texttt{void}$
6. *(Triple Negation)* $((\tau \to \texttt{void}) \to \texttt{void}) \to \texttt{void}$, $\tau \to \texttt{void}$

3 Church Encodings

Recall the syntax of System **F**.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \\ e & ::= & \lambda x{:}\tau.e \mid ee \mid \Lambda\alpha.e \mid e[\tau] \end{array}$$

As we saw in class, using the few type constructors available in System **F**, it is possible to define types with the same behavior as almost any of the types we have studied. These are called *Church encodings* and were originally defined by Alonzo Church in the untyped λ-calculus.

Task 4 *Lists can either be* `Nil` *or* `Cons` *of an element and a list. In ML-style, we write*

$$\tau\ \texttt{list} \triangleq \texttt{Nil} \mid \texttt{Cons of}\ \tau \times \tau\ \texttt{list}$$

Define the following types and functions in System **F**. *As usual, briefly explain (in 1-2 lines) the intuition behind your answer.*

1. *Define* τ `list` *in System* **F**.
2. *Define* $\texttt{nil}_\tau : \tau$ `list`, *the empty list.*
3. *Give the representation of the list* $[a_1, a_2, \ldots, a_n]$ *where* $a_i : \tau$.
4. *Define* $\texttt{cons}_\tau : \tau \to \tau$ `list` $\to \tau$ `list`.
5. *Define* $\texttt{listrec}(l, e_0, x.y.e_1) : \rho$ *(analogous to* `natrec` *in System* **T**). *Its static and dynamic semantics are given below. Here, x is bound to the head of the list, and y is bound to the result of the computation on the tail of the list.*

$$\frac{\Gamma \vdash l : \tau\ \texttt{list} \quad \Gamma \vdash e_0 : \rho \quad \Gamma, x : \tau, y : \rho \vdash e_1 : \rho}{\Gamma \vdash \texttt{listrec}(l, e_0, x.y.e_1) : \rho}\ (\textsc{ListRec})$$

$$\frac{}{\texttt{listrec}(\texttt{nil}, e_0, x.y.e_1) \mapsto e_0}\ (\textsc{ListRec-Nil})$$

$$\frac{}{\texttt{listrec}(\texttt{cons}(h, t), e_0, x.y.e_1) \mapsto [h, \texttt{listrec}(t, e_0, x.y.e_1)/x, y]e_1}\ (\textsc{ListRec-S})$$

6. *Define the function* $\texttt{append} : \tau$ `list` $\to \tau$ `list` $\to \tau$ `list`, *which takes two lists and appends the second list at the end of the first. (You can define this by working it out with* `listrec`, *but there is a cleaner, more elegant solution.)*