Assignment #6, Update 2: Polymorphism

15-312: Principles of Programming Languages

Out: Thursday, April 24, 2013
Due: Thursday, May 1, 2013 1:29PM

Introduction

In this assignment, we will explore Church encodings in System F and work with abstract types.

Submission

As usual, to submit your solutions place your `assn6.pdf` file in your handin directory.

If you wish to write your code as **well-formatted** `verbatim` text instead of fighting the TEX macros, you may do so, writing $\Lambda t.e$ as `FN(t)e`, $\forall x.e$ as `ALL x.e`, and $\exists x.e$ as `EXISTS x.e`. Where the assigment suggests writing $\lambda x_1 x_2 x_3.e$ as an abbreviation of $\lambda\,(x_1 : \tau_1)\,\lambda\,(x_2 : \tau_2)\,\lambda\,(x_3 : \tau_3)\,e$, you can write `fn(x1,x2,x3)e`.

1 Church Encodings

In this section we will explore the Church encoding of regular expressions. That is, we will encode regular expressions in System F. We will work with regular expressions governed by the following grammar:

$$R \quad ::= \quad a \mid b \mid \mathbf{0} \mid \mathbf{1} \mid R_1 + R_2 \mid R_1 \cdot R_2 \mid R^*$$

where, for simplicity, a and b are the only two characters. The semantics of regular expressions are specified by the $\mathcal{L}(\cdot)$ function, which maps a regular expression to a set of strings (called its *language*):

$$\begin{array}{lcl}
\mathcal{L}(a) & = & \{a\} \\
\mathcal{L}(b) & = & \{b\} \\
\mathcal{L}(\mathbf{0}) & = & \emptyset \\
\mathcal{L}(\mathbf{1}) & = & \{\epsilon\} \text{ (where } \epsilon \text{ is the empty string)} \\
\mathcal{L}(R_1 + R_2) & = & \mathcal{L}(R_1) \cup \mathcal{L}(R_2) \\
\mathcal{L}(R_1 \cdot R_2) & = & \{s_1 s_2 \mid s_1 \in \mathcal{L}(R_1) \text{ and } s_2 \in \mathcal{L}(R_2)\} \\
\mathcal{L}(R^*) & = & \{s_1 s_2 \ldots s_n \mid n \geq 0 \text{ and } s_i \in \mathcal{L}(R) \text{ for all } 1 \leq i \leq n\}
\end{array}$$

In addition to these introductory forms, we will also need to define an elimination form in order to examine and recursively compute on a regular expression:

$$\mathtt{iter}_{\mathtt{re}}(R; M_a; M_b; M_{\mathbf{0}}; x.y.M_+; M_{\mathbf{1}}; x.y.M.; x.M_*)$$

If $\tau_{\texttt{re}}$ is the type of your implementation of regular expressions in System F, the intro and elim forms should together obey the following typing rules:

$$\frac{}{\Delta\,\Gamma \vdash a : \tau_{\texttt{re}}}(\text{RE}_1) \qquad \frac{}{\Delta\,\Gamma \vdash b : \tau_{\texttt{re}}}(\text{RE}_2) \qquad \frac{}{\Delta\,\Gamma \vdash \mathbf{0} : \tau_{\texttt{re}}}(\text{RE}_3) \qquad \frac{}{\Delta\,\Gamma \vdash \mathbf{1} : \tau_{\texttt{re}}}(\text{RE}_4)$$

$$\frac{\Delta\,\Gamma \vdash R_1 : \tau_{\texttt{re}} \qquad \Delta\,\Gamma \vdash R_2 : \tau_{\texttt{re}}}{\Delta\,\Gamma \vdash R_1 + R_2 : \tau_{\texttt{re}}}(\text{RE}_5) \qquad \frac{\Delta\,\Gamma \vdash R_1 : \tau_{\texttt{re}} \qquad \Delta\,\Gamma \vdash R_2 : \tau_{\texttt{re}}}{\Delta\,\Gamma \vdash R_1 \cdot R_2 : \tau_{\texttt{re}}}(\text{RE}_6) \qquad \frac{\Delta\,\Gamma \vdash R : \tau_{\texttt{re}}}{\Delta\,\Gamma \vdash R^* : \tau_{\texttt{re}}}(\text{RE}_7)$$

$$\frac{\begin{array}{c}\Delta\,\Gamma \vdash R : \tau_{\texttt{re}} \qquad \Delta\,\Gamma \vdash M_a : \tau \qquad \Delta\,\Gamma \vdash M_b : \tau \qquad \Delta\,\Gamma \vdash M_{\mathbf{0}} : \tau \\ \Delta\,\Gamma \vdash M_{\mathbf{1}} : \tau \qquad \Delta\,\Gamma, x:\tau, y:\tau \vdash M_+ : \tau \qquad \Delta\,\Gamma, x:\tau, y:\tau \vdash M. : \tau \qquad \Delta\,\Gamma, x:\tau \vdash M_* : \tau\end{array}}{\Delta\,\Gamma \vdash \texttt{iter}_{\texttt{re}}(R; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*) : \tau}(\text{RE}_8)$$

The iterator should also satisfy the following specification for the dynamics, given equationally:

$$\begin{array}{lcl}
\texttt{iter}_{\texttt{re}}(a; M_a; _; _; _; _; _; _) & \equiv & M_a \\
\texttt{iter}_{\texttt{re}}(b; _; M_b; _; _; _; _; _) & \equiv & M_b \\
\texttt{iter}_{\texttt{re}}(\mathbf{0}; _; _; M_{\mathbf{0}}; _; _; _; _) & \equiv & M_{\mathbf{0}} \\
\texttt{iter}_{\texttt{re}}(\mathbf{1}; _; _; _; M_{\mathbf{1}}; _; _; _) & \equiv & M_{\mathbf{1}} \\
\texttt{iter}_{\texttt{re}}(R_1 + R_2; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*) & \equiv & [\texttt{iter}_{\texttt{re}}(R_1; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*), \\
 & & \texttt{iter}_{\texttt{re}}(R_2; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*)/x, y]M_+ \\
\texttt{iter}_{\texttt{re}}(R_1 \cdot R_2; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*) & \equiv & [\texttt{iter}_{\texttt{re}}(R_1; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*), \\
 & & \texttt{iter}_{\texttt{re}}(R_2; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*)/x, y]M. \\
\texttt{iter}_{\texttt{re}}(R^*; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*) & \equiv & [\texttt{iter}_{\texttt{re}}(R; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*)/x]M_*
\end{array}$$

Task 1.1 (10%). Implement the type $\tau_{\texttt{re}}$, constructors (a, b, $\mathbf{0}$, ...) and the eliminator ($\texttt{iter}_{\texttt{re}}$) in System F. For brevity, you may write $\lambda x_1 x_2 x_3.x$ for $\lambda x_1.\lambda x_2.\lambda x_3.x$, omitting type annotations if they are obvious from your definition of $\tau_{\texttt{re}}$.

$$\begin{array}{ll}
\tau_{\texttt{re}} & \triangleq \\
a & \triangleq \\
b & \triangleq \\
\mathbf{0} & \triangleq \\
\mathbf{1} & \triangleq \\
R_1 + R_2 & \triangleq \\
R_1 \cdot R_2 & \triangleq \\
R^* & \triangleq
\end{array}$$

$$\texttt{iter}_{\texttt{re}}(R; M_a; M_b; M_{\mathbf{0}}; M_{\mathbf{1}}; x.y.M_+; x.y.M.; x.M_*) \quad \triangleq$$

Task 1.2 (10%). Write a function to compute a new expression R' from the expression R so that $s \in \mathcal{L}(R)$ if and only if the reverse of s is in $\mathcal{L}(R')$. For example, $abbb \in \mathcal{L}(R)$ if and only if $bbba \in \mathcal{L}(R')$. You can use $\texttt{iter}_{\texttt{re}}$.

Task 1.3 (15%). Write a function to determine whether an expression R is nullible. (That is, whether the empty string $\epsilon \in \mathcal{L}(R)$?) You may assume that `true`, `false`, $\texttt{if}(M; M_{\texttt{true}}; M_{\texttt{false}})$, `and` and `or` are already defined. You can use $\texttt{iter}_{\texttt{re}}$.

2 Abstract Types and Sorted Sets

In this section we will practice programming with existential types. Below we show a version of System F extended with primitive existentials and a few other types to facilitate our programming. These types include binary products, natural numbers (`nat`), booleans (`bool`), orderings representing the results of comparison (`ord`), lists of natural numbers (`list`) and binary trees (`tree`).

$$
\begin{array}{llll}
\mathsf{Typ} & \tau & ::= & t \mid \mathtt{nat} \mid \mathtt{bool} \mid \mathtt{ord} \mid \mathtt{list} \mid \mathtt{tree} \mid \tau \times \tau \mid \tau \to \tau \mid \forall t.\tau \mid \exists t.\tau \\
\mathsf{Exp} & e & ::= & x \mid \mathtt{z} \mid \mathtt{succ}(e) \mid \mathtt{iter}_{\mathtt{nat}}(e; e_0; x.e_1) \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}(e; e_t; e_f) \mid \\
 & & & \mathtt{LT} \mid \mathtt{EQ} \mid \mathtt{GT} \mid \mathtt{case}_{\mathtt{ord}}(e; e_l; e_e; e_g) \mid \mathtt{nil} \mid \mathtt{cons}(e_h; e_t) \mid \mathtt{iter}_{\mathtt{list}}(e; e_n; x.y.e_c) \mid \\
 & & & \mathtt{empty} \mid \mathtt{branch}(e; e_l; e_r) \mid \mathtt{iter}_{\mathtt{tree}}(e; e_e; x.y.z.e_b) \mid \\
 & & & \langle e_1, e_2 \rangle \mid e \cdot \mathtt{l} \mid e \cdot \mathtt{r} \mid \lambda x{:}\tau.e \mid e_1(e_2) \mid \Lambda t.e \mid e[\tau] \mid \mathtt{let}\ x = e\ \mathtt{in}\ e' \mid \\
 & & & \mathtt{pack}\ \rho\ \mathtt{with}\ e\ \mathtt{as}\ \exists t.\tau \mid \mathtt{open}\ e_1\ \mathtt{as}\ t\ \mathtt{with}\ x : \tau\ \mathtt{in}\ e_2
\end{array}
$$

Orderings, of type `ord`, can be one of three values: `LT` ("less-than"), `EQ` ("equal") and `GT` ("greater-than"). An example list is $\mathtt{cons}(\mathtt{z}; \mathtt{cons}(\mathtt{z}; \mathtt{nil}))$ and $\mathtt{branch}(\mathtt{z}; \mathtt{empty}; \mathtt{branch}(\mathtt{s}(\mathtt{z}); \mathtt{empty}; \mathtt{empty}))$ is an example tree. All of the constructs in our language are eagerly evaluated.

Below we include some critical dynamics rules that explain the semantics of our new constructs:

$$\frac{}{\mathtt{iter}_{\mathtt{nat}}(\mathtt{z}; e_0; x.e_1) \mapsto e_0} \qquad \frac{e\ \mathsf{val}}{\mathtt{iter}_{\mathtt{nat}}(\mathtt{succ}(e); e_0; x.e_1) \mapsto [\mathtt{iter}_{\mathtt{nat}}(e; e_0; x.e_1)/x]e_1}$$

$$\frac{}{\mathtt{if}(\mathtt{true}; e_1; e_2) \mapsto e_1} \qquad \frac{}{\mathtt{if}(\mathtt{false}; e_1; e_2) \mapsto e_2}$$

$$\frac{}{\mathtt{case}_{\mathtt{ord}}(\mathtt{LT}; e_1; e_2; e_3) \mapsto e_1} \qquad \frac{}{\mathtt{case}_{\mathtt{ord}}(\mathtt{EQ}; e_1; e_2; e_3) \mapsto e_2} \qquad \frac{}{\mathtt{case}_{\mathtt{ord}}(\mathtt{GT}; e_1; e_2; e_3) \mapsto e_3}$$

$$\frac{}{\mathtt{iter}_{\mathtt{list}}(\mathtt{nil}; e_0; x.y.e_2) \mapsto e_0} \qquad \frac{e_h\ \mathsf{val} \quad e_t\ \mathsf{val}}{\mathtt{iter}_{\mathtt{list}}(\mathtt{cons}(e_h; e_t); e_0; x.y.e_2) \mapsto [e_h, \mathtt{iter}_{\mathtt{list}}(e_t; e_0; x.y.e_2)/x, y]e_2}$$

$$\frac{}{\mathtt{iter}_{\mathtt{tree}}(\mathtt{empty}; e_0; x.y.z.e_3) \mapsto e_0}$$

$$\frac{e_n\ \mathsf{val} \quad e_l\ \mathsf{val} \quad e_r\ \mathsf{val}}{\mathtt{iter}_{\mathtt{tree}}(\mathtt{branch}(e_n; e_l; e_r); e_0; x.y.z.e_3) \mapsto [e_n, \mathtt{iter}_{\mathtt{tree}}(e_l; e_0; x.y.z.e_3), \mathtt{iter}_{\mathtt{tree}}(e_r; e_0; x.y.z.e_3)/x, y, z]e_3}$$

$$\frac{}{\mathtt{z}\ \mathsf{val}} \qquad \frac{e\ \mathsf{val}}{\mathtt{succ}(e)\ \mathsf{val}} \qquad \frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\mathtt{cons}(e_1; e_2)\ \mathsf{val}} \qquad \frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val} \quad e_3\ \mathsf{val}}{\mathtt{branch}(e_1; e_2; e_3)\ \mathsf{val}}$$

2.1 Recursors from Iterators

The primitive iterators (`iter`) are sometimes difficult to use. We will make them more user-friendly in this subsection before returning to the main problem. Take `nat` for example: the e_1 in $\texttt{iter}_{\texttt{nat}}(e_n; e_0; x.e_1)$ has no direct access to the predecessor of e_n, only to the result of the recursion on the predecessor. In many cases, however, the code would be cleaner if e_1 could also access the predecessor directly. For example, a straightforward definition of the factorial function multiplies the result of the recursion with the predecessor.

We can remedy this by defining the recursor $\texttt{rec}_{\texttt{nat}}(e_n; e_0; x.y.e_1)$ where x is the predecessor of e_n and y is the result of recursion on x. In other words the specification for $\texttt{rec}_{\texttt{nat}}$ is,

$$\begin{array}{lcl} \texttt{rec}_{\texttt{nat}}(\texttt{z}; e_0; x.y.e_1) & \equiv & e_0 \\ \texttt{rec}_{\texttt{nat}}(\texttt{succ}(e); e_0; x.y.e_1) & \equiv & [e, \texttt{rec}_{\texttt{nat}}(e; e_0; x.y.e_1)/x, y]e_1 \end{array}$$

The recursor can be implemented by maintaining a pair containing both the result along with the reconstructed predecessor internally to the iteration:

$$\texttt{rec}_{\texttt{nat}}(e; e_0; x.y.e_1) \triangleq \texttt{iter}_{\texttt{nat}}(e; \langle \texttt{z}, e_0 \rangle; x.\langle \texttt{s}(x \cdot \texttt{l}), [x \cdot \texttt{l}, x \cdot \texttt{r}/x, y]e_1 \rangle) \cdot \texttt{r}$$

Task 2.1 (10%). Write down a similar specification for $\texttt{rec}_{\texttt{tree}}(e; e_0; x.y.y'.z.z'.e_3)$ and define it using $\texttt{iter}_{\texttt{tree}}$ so that y and z are the left and right subtrees of e, and y' and z' are the result of recursion on these subtrees, respectively.

2.2 Interface and Code

The following is the interface for sorted sets, which we will implement in two ways: using lists and binary search trees:

$$\tau_i = \exists t.(t \times ((\texttt{nat} \to t \to t) \times (\texttt{nat} \to t \to \texttt{bool}))$$

Here, t is an abstract type representing the implementation of the set. The first element of the tuple is an empty set. The second inserts a new number into the set; if the number is already present, the set remains the same. The third element in the tuple queries whether a number is in the set.

Task 2.2 (10%). First, let us write a client which uses the package of type τ_i. Write an expression e that takes a list of numbers as input and produces the unique numbers in that list. Your term should have type $impl : \tau_i \vdash e : \texttt{list} \to \texttt{list}$, where $impl$ is an implementation of sorted set available in the context. You should use the sorted set implementation to keep track of the unique elements seen in the list. The existential type allows you to answer this task without any knowledge about the implementation of the interface.

$$\begin{array}{l} \texttt{open } impl \texttt{ as } t \texttt{ with } x : t \times ((\texttt{nat} \to t \to t) \times (\texttt{nat} \to t \to \texttt{bool}) \texttt{ in} \\ \texttt{let } \mathit{emp} = x \cdot \texttt{l} \texttt{ in} \\ \texttt{let } \mathit{ins} = x \cdot \texttt{r} \cdot \texttt{l} \texttt{ in} \\ \texttt{let } \mathit{member} = x \cdot \texttt{r} \cdot \texttt{r} \texttt{ in} \\ \ldots \end{array}$$

Task 2.3 (25%). Write two implementations e and e' of this interface, that is. The first implementation, e, should use `list` as the basis of the data structure, and the second implementation e' should use `tree` as the basis of the data structure. You should have

$$\mathit{cmp} : \mathtt{nat} \to \mathtt{nat} \to \mathtt{ord} \vdash e : \mathtt{list} \times ((\mathtt{nat} \to \mathtt{list} \to \mathtt{list}) \times (\mathtt{nat} \to \mathtt{list} \to \mathtt{bool}))$$

and

$$\mathit{cmp} : \mathtt{nat} \to \mathtt{nat} \to \mathtt{ord} \vdash e' : \mathtt{tree} \times ((\mathtt{nat} \to \mathtt{tree} \to \mathtt{tree}) \times (\mathtt{nat} \to \mathtt{tree} \to \mathtt{bool}))$$

You may assume that you have a definition of $\mathtt{rec}_{\mathtt{list}}$ similar to $\mathtt{rec}_{\mathtt{tree}}$ and use it.

- $\mathtt{cmp}\,\overline{n}\,\overline{m} \cong \mathtt{LT} : \mathtt{ord}$ iff. $n < m$.
- $\mathtt{cmp}\,\overline{n}\,\overline{m} \cong \mathtt{EQ} : \mathtt{ord}$ iff. $n = m$.
- $\mathtt{cmp}\,\overline{n}\,\overline{m} \cong \mathtt{GT} : \mathtt{ord}$ iff. $n > m$.

In both implementations, you are required to maintain the invariant that underlying `list` or `tree` is sorted in ascending order.

Task 2.4 (10%). Given e and e' as defined in the previous task, package them up into expressions of type τ_i.

Task 2.5 (10%). Along the lines of the discussion in Chapter 21.4 of *PFPL*, complete the following statement about the requirements on the implementations for them to be bisimilar. These requirements are guided by the type of the interface.

"The two implementations e and e' are bisimilar if there exists a relation R between expressions of type `list` and `tree` such that ..."