Cornell University
Computer Science 611

Problem Set 6 [Last updated: Thursday 12/2 6PM] DUE: 3 December 2004

Turn in the written part of the assignment by 5PM on the due date in Upson 4119.

1. Recursive types [7 pts]

   Consider mutually recursive type definitions like the following:

   type Node = Edge list
   type Edge = Node * Node

   Eliminate the mutual recursion by giving μ-recursive types for Node and Edge, and show that the unfoldings of your Node and Edge types satisfy their respective equations. You may assume that list and * are built-in type constructors.

2. Axiomatic semantics [8 pts]

   Consider this partial correctness assertion:

   $$\{x < 1\}\textbf{while}\ x < 10\ \textbf{do}\ x := x + 1\{x = 10\}$$

   Give a Hoare logic proof of this assertion. What is the loop invariant?

3. Objects vs. Datatypes [35 pts]

   Object-oriented languages and functional languages take different approaches to extensibility. Objects make it easy to extend existing data types with new fields, but adding new operations is less easy because they need to be added to every class. With algebraic datatypes as in SML, new operations can be added easily but adding new data representations requires modifying every case expression to handle the new cases. In this problem we will convert between these two ways to represent data abstractions, by exploiting the Curry-Howard correspondence.

   Negation is needed to order to establish a correspondence. In System F, the polymorphic lambda calculus, there is no type 0 corresponding to logical false, but a good approximation is the type $\forall X.X$. In fact, quantification over a type variable can be used to achieve negation more generally. For example, by the Curry-Howard correspondence, the type $A \to \forall X.X$ corresponds to the boolean formula $\neg A$.

   Recall that System F has the same typing rules as the typed lambda calculus, extended with two expression forms that support parametric polymorphism. The typing rule for type abstraction requires that X is fresh to prevent type-variable capture.

   $$\frac{\Delta, X; \Gamma \vdash e : \tau \quad X \notin \Delta}{\Delta; \Gamma \vdash \Lambda X.e : \forall X.\tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall X.\tau \quad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e[\tau'] : \tau\{\tau'/X\}}$$

   (a) Show that an equivalent representation of $\neg A$ is the type $\forall X.A \to X$ by giving terms that coerce from $A \to \forall X.X$ to $\forall X.A \to X$ and in reverse.

   Ignoring recursion, a record type $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ is a reasonable approximation of an object type. It is also structurally similar to a tuple type $\tau_1 * \ldots * \tau_n$, so it corresponds similarly to a proposition $\phi_1 \wedge \ldots \wedge \phi_n$. Ignoring recursion, an SML datatype is a *variant type* $[l_1 : \tau_1, \ldots, l_n : \tau_n]$: essentially a labeled disjoint sum in the same way that a record type is a labeled product. So it corresponds to a disjunction $\phi_1 \vee \ldots \vee \phi_n$. We can add records and variants to both the typed lambda calculus and System F straightforwardly, using rules like those in Pierce, chapter 11 (evaluation contexts are standard):

$$
\begin{array}{rcl}
e & ::= & \ldots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \mid l(e) \mid \mathsf{case}\ e\ \mathsf{of}\ l_1(x_1 : \tau_1).e_1 | \ldots | l_n(x_n : \tau_n).e_n \\
\tau & ::= & \ldots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \mid [l_1 : \tau_1, \ldots, l_n : \tau_n]
\end{array}
$$

$$
\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \longrightarrow v_i \qquad \mathsf{case}\ l_i(v)\ \mathsf{of}\ l_1(x_1 : \tau_1).e_1 | \ldots | l_n(x_n : \tau_n).e_n \longrightarrow e_i\{v/x_i\}
$$

$$
\frac{\Gamma \vdash e_i : \tau_i \quad {}^{\forall i \in 1..n}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}
$$

$$
\frac{\Gamma \vdash e : [l_1 : \tau_1, \ldots, l_n : \tau_n] \quad \Delta; \Gamma, x_i : \tau_i \vdash e_i : \tau \quad {}^{\forall i \in 1..n}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ l_1(x_1 : \tau_1).e_1 | \ldots | l_n(x_n : \tau_n).e_n : \tau} \qquad \frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash l_i(e) : [l_1 : \tau_1, \ldots, l_n : \tau_n]}
$$

(b) Use DeMorgan's rule to produce a type that is equivalent to $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ but uses a variant type constructor instead of a record type constructor. You'll need to perform negation using a trick similar to that in part (a). Dually, give a type that is equivalent to $[l_1 : \tau_1, \ldots, l_n : \tau_n]$ but uses a record type constructor instead of a variant.

Now, we will use the solution to the previous part to give a typed translation from the typed lambda calculus with records and variants to System F with variants and records. That is, records in the original program should be converted into variants in the target, and vice-versa. You will need to use universals in the target language translation. The translation should operate on a typing derivation to produce a new target-language expression whose typing derivation follows inductively from the source typing derivation. Types must be translated correspondingly. That is, we require that

$$
[\![\Gamma \vdash e : \tau]\!] = \Delta; [\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau]\!]
$$

where the right-hand side is derivable if the left-hand side is, and $\Delta \vdash [\![\tau]\!]$.

To get you started, here is the translation for Γ, λ and $\tau_1 \to \tau_2$, with premises included to help make the inductive argument that the above condition is satisfied:

$$
\begin{array}{c}
[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] = x_1 : [\![\tau_1]\!], \ldots, x_n : [\![\tau_n]\!] \\
[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!]
\end{array}
$$

$$
\left[\!\!\left[\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\, e : \tau_1 \to \tau_2} \right]\!\!\right] = \frac{\Delta; [\![\Gamma]\!], x : [\![\tau_1]\!] \vdash [\![e]\!] : [\![\tau_2]\!]}{\Delta; [\![\Gamma]\!] \vdash \lambda x : [\![\tau_1]\!].\, [\![e]\!] : [\![\tau_1 \to \tau_2]\!]}
$$

Some people have been a bit confused by this rule. It is really a pun: the use of $[\![e]\!]$, isn't strictly correct because translation operates on derivations, not terms. What it's shorthand for is the following. We can define a translation relation

$$\Gamma \vdash e : \tau \quad \longmapsto \quad \Delta; [\![\Gamma]\!] \vdash e' : [\![\tau]\!]$$

meaning that the derivation on the left can be translated to the derivation on the right. Thus translation isn't really a function because there may be more than one translation corresponding to the derivation on the left, with different Δ's. Then if

$$\Gamma, x : \tau_1 \vdash e : \tau_2 \quad \longmapsto \quad \Delta; [\![\Gamma]\!], x : [\![\tau_1]\!] \vdash e' : [\![\tau_2]\!]$$

then the following translation holds:

$$\Gamma \vdash \lambda x : \tau_1 . \, e : \tau_1 \to \tau_2 \quad \longmapsto \quad \Delta; [\![\Gamma]\!] \vdash \lambda x : [\![\tau_1]\!] . \, e' : [\![\tau_1]\!] \to [\![\tau_2]\!]$$

The reason is that the derivation of the translation of e can be used to derive the translation of $\lambda x : \tau_1 . \, e$:

$$\frac{\Delta; [\![\Gamma]\!], x : [\![\tau_1]\!] \vdash e' : [\![\tau_2]\!]}{\Delta; [\![\Gamma]\!] \vdash \lambda x : [\![\tau_1]\!] . \, e' : [\![\tau_1]\!] \to [\![\tau_2]\!]}$$

The notation used above is a way of writing all this at once.

(c) Give the translation rules for a variable, for application, and for the four new expression forms. Your answer to the previous part should already explain how to translate record types and variant types. Remember that record and variant types may mention other record/variant types. To check your work you can convince yourself that the translation is sound by induction on the typing derivation, but you don't need to include this argument in your solution. If you get stuck, you may need to revisit your solution to part (b).

(d) Give the width subtyping coercion term $\Theta(\{l_1 : \tau_1, \ldots, l_{n+1} : \tau_{n+1}\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\})$ and then construct its translation.