CS611 Homework 6 — DUE: December 1, 2000

What to turn in

Turn in the assignment during class on the due date.

1. Subtyping and recursion (20 pts.)

   We saw in class that objects could be modeled roughly as recursive record types of the form $\mu X.\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$. This formulation gives some insight into sound subtyping rules for object types in object-oriented languages. Let us consider the typed lambda calculus extended with recursive types, record types, and subtyping. Any program that is well-formed in the language without recursive types (for which rules were given in class) should also be well-formed in this extended language.

   The rules for subtyping in this language can be obtained by modifying the rules given earlier for type equivalence in the typed lambda calculus with recursive types. As for type equivalence, we define a context E containing a set of assumed subtype relations $\tau_1 \leq \tau_2$, and use the following rule to terminate the unfolding of recursive types:

   $$\frac{}{E, \tau_1 \leq \tau_2 \vdash \tau_1 \leq \tau_2}$$

   (a) (2 pts.) Give the inference rule defining the subtype judgment $E \vdash \tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'$.

   (b) (2 pts.) Give the inference rules defining the subtype judgements $E \vdash \mu X.\tau \leq \tau'$ and $E \vdash \tau' \leq \mu X.\tau$, where τ' does not have the form $\mu Y.\tau''$.

   (c) (3 pts.) Give the inference rule defining the subtype judgement $E \vdash \mu X.\tau \leq \mu Y.\tau'$.

   (d) (5 pts.) Consider the following two types:

   $$\begin{aligned} \textsf{point} &= \mu P.\{x : \textsf{int}, y : \textsf{int}\} \\ \textsf{colored\_point} &= \mu Q.\{x : \textsf{int}, y : \textsf{int}, color : \textsf{int}\} \end{aligned}$$

   According to the rules you have given, we should have $\textsf{colored\_point} \leq \textsf{point}$. Say whether each of the following putative subtype relationships also holds. If the relationship holds, give a subtype derivation (proof tree). If it does not hold, give a counter-example: code demonstrating that the subtype relationship would be unsafe.

   i. $\mu P.\{x : P\} \leq \mu Q.\{x : Q\}$
   ii. $\mu P.\{mv : \textsf{int} \rightarrow P, y : \textsf{int}\} \leq \mu Q.\{mv : \textsf{int} \rightarrow Q\}$
   iii. $\mu P.\{eq : P \rightarrow \textsf{bool}, y : \textsf{int}\} \leq \mu Q.\{eq : Q \rightarrow \textsf{bool}\}$
   iv. $\mu P.\{a : \textsf{ref}\ P, y : \textsf{int}\} \leq \mu Q.\{a : \textsf{ref}\ Q\}$ (Note that $\textsf{ref}\ \tau \leq \textsf{ref}\ \tau'$ iff $\tau \cong \tau'$).
   v. $\mu P.\{l : P, r : P, v : \textsf{int}\} \leq \mu Q.\{l : Q, r : Q\}$

   (e) (8 pts.) Define a coercion function for the subtype judgment $E \vdash \mu X.\tau \leq \mu X'.\tau'$. Make sure that your function definition "bottoms out". Assume that you have available the tF fixed-point operator ($\textsf{rec}\ y : \tau \rightarrow \tau'.\textsf{fn}\ x\ e$).

2. Encoding weak existential types in the polymorphic lambda calculus (30 pts.)

   In class we have discussed how existential types can be used as a primitive encapsulation mechanism that hides some part of a type from code outside the definition of a value. Thus, existential types capture an important aspect of modular programming. In this problem we will show that weak existential types can be translated into the polymorphic lambda calculus, using some insights from the Curry-Howard isomorphism.

   Our source language will consist of the simply typed lambda calculus extended with predicative[1] existential types:

[1] The fact that they are predicative is not particularly important for this problem.

$$\begin{array}{rcl} \tau & ::= & B \mid X \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \tau \mid \sigma_1 \to \sigma_2 \mid \exists X.\sigma \\ e & ::= & b \mid x \mid \lambda x{:}\sigma.\, e \mid e_1\ e_2 \mid \mathsf{pack}\ [X = \tau, e] \mid \mathsf{unpack}\ e\ \mathsf{as}\ [X, x]\ \mathsf{in}\ e' \end{array}$$

We could easily add more types to the source language but they will not affect this problem in an interesting way. The typing rules for this language are standard:

$$\frac{}{\Delta; \Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{}{\Delta; \Gamma \vdash b : B}$$

$$\frac{\Delta; \Gamma, x{:}\sigma \vdash e : \sigma' \quad \Delta \vdash \sigma}{\Delta; \Gamma \vdash \lambda x{:}\sigma.\, e : \sigma \to \sigma'} \qquad \frac{\Delta; \Gamma \vdash e_1 : \sigma \to \sigma' \quad \Delta; \Gamma \vdash e_2 : \sigma}{\Delta; \Gamma \vdash e_1\ e_2 : \sigma'}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma\{\tau/X\} \quad \Delta \vdash \tau}{\Delta; \Gamma \vdash \mathsf{pack}\ [X = \tau, e] : \exists X.\sigma} \qquad \frac{\Delta; \Gamma \vdash e : \exists Y.\sigma_1 \quad \Delta, X; \Gamma, x{:}\sigma_1\{X/Y\} \vdash e_2 : \sigma_2 \quad X \notin \Delta \quad \Delta \vdash \sigma_2}{\Delta; \Gamma \vdash \mathsf{unpack}\ e_1\ \mathsf{as}\ [X, x]\ \mathsf{in}\ e_2 : \sigma_2}$$

In the unpack rule, the requirement that X is not in Δ ensures that it does not capture a type variable mentioned in the context Γ. In addition, the type σ_2 must be well-formed in Δ; this prevents values of the hidden representation type X from leaking out into a context in which they do not make sense.

The target language is the polymorphic lambda calculus, which supports impredicative polymorphism:

$$\begin{array}{rcl} \tau, \sigma & ::= & B \mid X \mid \tau_1 \to \tau_2 \mid \forall X.\sigma \\ e & ::= & b \mid x \mid \lambda x{:}\sigma.\, e \mid e_1\ e_2 \mid \Lambda X.e \mid e[\tau] \end{array}$$

The target language has the same typing rules as the source language (except for the rules for pack and unpack); it also has two rules for supporting polymorphism:

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

Note that the rule for type abstraction requires that the new type variable X be fresh, to prevent capture of type variables appearing in Γ.

The target language and the source language have the usual rules for well-formed type expressions:

$$\frac{}{\Delta, X \vdash X} \qquad \frac{}{\Delta \vdash B} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}$$

$$\frac{\Delta \vdash \sigma_1 \quad \Delta \vdash \sigma_2}{\Delta \vdash \sigma_1 \to \sigma_2} \qquad \frac{\Delta, X \vdash \sigma}{\Delta \vdash \forall X.\sigma} \qquad \frac{\Delta, X \vdash \sigma}{\Delta \vdash \exists X.\sigma}$$

Your goal in this problem is to provide a typed translation from the source language to the target language, and to show that this translation works. The typed translation will convert type judgments in the source language into type judgments in the target language, in such a way that a translated target program will automatically have a target-language type derivation if its source had a source-language type derivation.

You will define: (1) a translation $\mathcal{T}[\![\sigma]\!]$ that maps source-language types into target-language types; (2) a translation into the target language for the expression $\mathsf{pack}\ [X = \tau, e]$; (3) a translation for the expression $\mathsf{unpack}\ e\ \mathsf{as}\ [X, x]\ \mathsf{in}\ e'$. These are the only source-language constructs that are not allowed in the target language, so the remainder of the translation is largely boilerplate.

The Curry-Howard isomorphism will assist you in constructing this translation. We know that the type $\exists X.\sigma$ corresponds to the formula $\exists X.\phi$. We have seen that meaning-preserving program transformations, such as CPS conversion, result in programs whose types are logically equivalent to the original. From classical logic, we also know a formula that is equivalent to $\exists X.\phi$ but uses universals instead: $\neg\forall X.\neg\phi$.

(a) (2 pts.) Types whose formulas contain negation can be generated by using continuations, but our target language has no continuations. Give another formula that is equivalent to $\exists X.\phi$ but contains only logical operators for which there is a target-language equivalent.

(b) (2 pts.) We are translating source-language types σ into target-language types $\mathcal{T}[\![\sigma]\!]$. What logically equivalent target-language type should $\mathcal{T}[\![\cdot]\!]$ map $\exists X.\sigma$ to?

(c) (8 pts.)

You will define the important parts of a translation function $\mathcal{E}[\![\cdot]\!]$, which when applied to a source-language type judgement $\Delta; \Gamma \vdash e : \sigma$, produces a provable target-language type judgement $\Delta'; \Gamma' \vdash e' : \mathcal{T}[\![\sigma]\!]$. It will be useful to have a semantic function $\mathcal{G}[\![\cdot]\!]$ that simply maps all the types of variables in Γ into the target language: $\mathcal{G}[\![\emptyset]\!] = \emptyset$, $\mathcal{G}[\![\Gamma, x\!:\!\sigma]\!] = \mathcal{G}[\![\Gamma]\!], x\!:\!\mathcal{T}[\![\sigma]\!]$.

Complete the following translation rule for pack. You will need to introduce new variables; add any premises needed to control the selection of variable names.

$$\frac{\mathcal{E}[\![\Delta; \Gamma \vdash e : \sigma\{\tau/X\}]\!] = \ ?}{\mathcal{E}[\![\Delta; \Gamma \vdash \mathsf{pack}\ [X = \tau, e] : \exists X.\sigma]\!] = \ ?}$$

(d) (8 pts.)

Complete the following translation rule for unpack.

$$\frac{\mathcal{E}[\![\Delta; \Gamma \vdash e_1 : \exists Y.\sigma_1]\!] = \ ? \quad \mathcal{E}[\![\Delta, X; \Gamma, x\!:\!\sigma_1\{X/Y\} \vdash e_2 : \sigma_2]\!] = \ ?}{\mathcal{E}[\![\Delta; \Gamma \vdash \mathsf{unpack}\ e_1\ \mathsf{as}\ [X, x]\ \mathsf{in}\ e_2 : \sigma_2]\!] = \ ?}$$

(e) (2 pts.) Give a concise table explaining the purpose of each use of type abstraction ($\Lambda X.e$) and application ($e[\tau]$) in your translation.

(f) (8 pts.) Show that the expressions that are your translations of pack and unpack are well-formed. For each translated expression, give a type derivation in which the tops of the proof tree are either axioms or are judgments that you are assured of having because the source-language expression is well-formed.