CS611 Homework 6 DUE: December 7, 2001

What to turn in

Turn in the assignment during class on the due date.

1. Strong normalization (25 pts.)

   (a) (20 pts.) Show that all expressions in the language $\lambda^{\rightarrow\times+}$ are strongly normalizing by extending the proof of strong normalization for $\lambda^{\rightarrow}$.

   (b) (5 pts.) Where does the proof fail if we add recursive types to the language?

2. Object encodings (35 pts.)

   In this problem we will examine two ways to encode object types. These formulations give some insight into sound subtyping rules for object types in object-oriented languages. Let us consider the typed lambda calculus extended with pairs, records, recursive types, existential types, and subtyping:

$$\begin{aligned} e ::= {} & b \mid x \mid \lambda x:\tau.e \mid e_0\ e_1 \mid \langle e_0, e_1\rangle \mid \mathsf{left}\ e \mid \mathsf{right}\ e \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \mid \\ & \mathsf{pack}\ [X = \tau, e] \mid \mathsf{unpack}\ e\ \mathsf{as}\ [X, x]\ \mathsf{in}\ e' \mid \mathsf{fold}_{\mu X.\tau}\ e \mid \mathsf{unfold}\ e \\ \tau ::= {} & B \mid X \mid \tau \rightarrow \tau' \mid \tau * \tau' \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \mid \mu X.\tau \mid \exists X.\tau \end{aligned}$$

   We define a context Δ to contain the type variables in scope:

$$\Delta ::= \emptyset \mid \Delta, X$$

   and augment this definition with the usual rules about the irrelevance of ordering type variables in Δ. A judgment of the form $\Delta \vdash \tau$ asserts that τ is well-formed in Δ. Thus, we have:

$$\frac{}{\Delta \vdash B} \qquad \frac{}{\Delta, X \vdash X} \qquad \frac{\Delta \vdash \tau \quad \Delta \vdash \tau'}{\Delta \vdash \tau \rightarrow \tau'} \qquad \frac{\Delta \vdash \tau \quad \Delta \vdash \tau'}{\Delta \vdash \tau * \tau'}$$

$$\frac{\Delta \vdash \tau_i^{\forall i \in \{1,\ldots,n\}}}{\Delta \vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \qquad \frac{\Delta, X \vdash \tau}{\Delta \vdash \mu X.\tau} \qquad \frac{\Delta, X \vdash \tau}{\Delta \vdash \exists X.\tau}$$

   Encoding as recursive records

   One way to model objects is as recursive record types of the form $\mu X.\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$. For instance:

$$\begin{aligned} \mathsf{point} &= \mu P.\{x : \mathsf{real}, y : \mathsf{real}\} \\ \mathsf{mvpoint} &= \mu Q.\{x : \mathsf{real}, y : \mathsf{real}, \mathit{move} : \mathsf{real} \rightarrow \mathsf{real} \rightarrow Q\} \end{aligned}$$

   To construct a point object we might write the function:

$$\mathsf{make\_point} = \lambda a : \mathsf{real}.\ \lambda b : \mathsf{real}.\ \mathsf{fold}_{\mathsf{point}}\ \{x = a, y = b\}$$

   The rules for subtyping in this language can be obtained by modifying the rules given in class for type equivalence in the typed lambda calculus extended with recursive types. As with 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:

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

Recall the subtyping rules for function types, pairs, and records:

$$\frac{\Delta; E \vdash \tau_1' \leq \tau_1 \quad \Delta; E \vdash \tau_2 \leq \tau_2'}{\Delta; E \vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'} \qquad \frac{\Delta; E \vdash \tau_1 \leq \tau_1' \quad \Delta; E \vdash \tau_2 \leq \tau_2'}{\Delta; E \vdash \tau_1 * \tau_2 \leq \tau_1' * \tau_2'}$$

$$\frac{\Delta; E \vdash \tau_i \leq \tau_i' \ ^{\forall i \in \{1,\ldots,n\}}}{\Delta; E \vdash \{l_1 : \tau_1, \ldots, l_m : \tau_m\} \leq \{l_1 : \tau_1', \ldots, l_n : \tau_n'\}} \text{ where } m \geq n$$

(a) (5 pts.) Give the inference rules defining the subtype judgments $\Delta; E \vdash \mu X.\tau \leq \tau'$ and $\Delta; E \vdash \tau' \leq \mu X.\tau$, where τ' does not have the form $\mu Y.\tau''$.

(b) (5 pts.) Give the inference rule defining the subtype judgment $\Delta; E \vdash \mu X.\tau \leq \mu Y.\tau'$.

(c) (5 pts.) According to the rules you have given, we should have $\mathsf{mvpoint} \leq \mathsf{point}$. 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. $\mu P.\{mv : \mathsf{int} \to P, y : \mathsf{int}\} \leq \mu Q.\{mv : \mathsf{int} \to Q\}$

iii. $\mu P.\{eq : P \to \mathsf{bool}, y : \mathsf{int}\} \leq \mu Q.\{eq : Q \to \mathsf{bool}\}$

iv. $\mu P.\{a : \mathsf{ref}\ P, y : \mathsf{int}\} \leq \mu Q.\{a : \mathsf{ref}\ Q\}$ (Note that $\mathsf{ref}\ \tau \leq \mathsf{ref}\ \tau'$ iff $\tau \cong \tau'$).

v. $\mu P.\{l : P, r : P, v : \mathsf{int}\} \leq \mu Q.\{l : Q, r : Q\}$

Encoding as recursive records with existentials

One problem with the recursive record encoding is that an object's implementation is visible to its clients. In languages such as Smalltalk, Java, or C++, the implementation may be hidden by declaring some fields of the object private; in fact, in Smalltalk, all class members are private and clients can access the object only through its method interface.

To model this property, we hide the implementation of an object using existential types. We treat objects as a pair of a state, represented by the type variable X, and a record of methods, each of which takes an argument of type X. This argument can be considered analogous to this in Java or C++ or self in Smalltalk. As an example, we might encode the point type above as

$$\mathsf{point} = \mu P.\ \exists X. X * \{x : X \to \mathsf{real}, y : X \to \mathsf{real}\}$$

Note that we are no longer constrained to implement a point as a record of two reals with named x and y. For instance, we might implement a point as a pair using polar coordinates and define the following function to create a new point from its Cartesian coordinates:

```
make_point = $\lambda a$ : real. $\lambda b$ : real.
                 letrec $m : \{x : (\mathsf{real} * \mathsf{real}) \to \mathsf{real}, y : (\mathsf{real} * \mathsf{real}) \to \mathsf{real}\}$ = {
                     $x = \lambda s : (\mathsf{real} * \mathsf{real}).\ (\mathsf{left}\ s) * \mathsf{cos}(\mathsf{right}\ s)$,
                     $y = \lambda s : (\mathsf{real} * \mathsf{real}).\ (\mathsf{left}\ s) * \mathsf{sin}(\mathsf{right}\ s)$} in
                 let $r = \mathsf{sqrt}(a * a + b * b)$ in
                 let $\theta = \mathsf{arcsin}(b/r)$ in
                 $\mathsf{fold}_{\mathsf{point}}$ (pack $[X = \mathsf{real} * \mathsf{real}, \langle\langle r, \theta\rangle, m\rangle]$)
```

Now, to call the method x on a point p, we do:

```
unpack (unfold $p$) as $[X, o]$ in
    let $s : X$ = (left $o$) in
    let $m : \{x : X \to \mathsf{real}, y : X \to \mathsf{real}\}$ = (right $o$) in
    $(m.x\ s)$
```

Observe that the caller does not know the implementation of the state X.

(d) (3 pts.) Using this encoding, what is the type encoding of mvpoint?

(e) (5 pts.) Define a function make_mvpoint which, given two reals, returns a new mvpoint representing that point. The *move* method should take two real numbers dx and dy and return a new mvpoint where the x and y coordinates of the original point are offset by dx and dy, respectively. You are free to implement an mvpoint's state in any way you choose.

(f) (10 pts.) Give the inference rule defining the subtype judgment $\Delta; E \vdash \exists X.\tau \leq \exists X'.\tau'$. The rule should not require that the types be equivalent; however, if the subtype relation holds in both directions, the rule should enforce equivalence of the two types.

(g) (2 pts.) Show that $\emptyset; \emptyset \vdash \mathsf{mvpoint} \leq \mathsf{point}$.

3. Encoding sum and product types in the polymorphic lambda calculus (40 pts.)

Sum types and product types, as seen in $\lambda^{\rightarrow\times+}$, are important features in programming languages. In this problem, we will show that by using some insights from the Curry-Howard isomorphism, sum types in $\lambda^{\rightarrow\times+}$ can be encoded using product types and universal types in the polymorphic lambda calculus. Similarly, we can encode product types as sum types.

Our source language will consist of the simply typed language extended with sum and product types.

$$\begin{aligned}
\tau &::= B \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2 \\
e &::= b \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid \mathsf{inl}_{\tau_1+\tau_2} e \mid \mathsf{inr}_{\tau_1+\tau_2} e \mid \mathsf{case}\ e_0\ \mathsf{of}\ e_1 | e_2 \mid \langle e_1, e_2 \rangle \mid \mathsf{left}\ e \mid \mathsf{right}\ e
\end{aligned}$$

The typing rules for this language are the same as defined in lecture 30.

The target-sum language is the polymorphic lambda calculus extended with only sum types.

$$\begin{aligned}
\tau &::= B \mid X \mid \tau_1 \rightarrow \tau_2 \mid \forall X.\tau \mid \tau_1 + \tau_2 \\
e &::= b \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid \Lambda X.e \mid e[\tau] \mid \mathsf{inl}_{\tau_1+\tau_2} e \mid \mathsf{inr}_{\tau_1+\tau_2} e \mid \mathsf{case}\ e_0\ \mathsf{of}\ e_1 | e_2
\end{aligned}$$

The target-product language is the polymorphic lambda calculus extended with only product types.

$$\begin{aligned}
\tau &::= B \mid X \mid \tau_1 \rightarrow \tau_2 \mid \forall X.\tau \mid \tau_1 * \tau_2 \\
e &::= b \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid \Lambda X.e \mid e[\tau] \mid \langle e_1, e_2 \rangle \mid \mathsf{left}\ e \mid \mathsf{right}\ e
\end{aligned}$$

The target languages have the same typing rules as the source language, extended with two rules for supporting polymorphism:

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

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.

Both target languages have the usual rules for well-formed type expressions:

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

$$\frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 * \tau_2} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 + \tau_2} \qquad \frac{\Delta, X \vdash \tau}{\Delta \vdash \forall X.\tau}$$

Your goal in this problem is to provide typed translations from the source language to each of the two target languages, and to show that these translations work. Each typed translation will convert type judgments in the source language into type judgments in the target language, in such a way that a translated program will automatically have a target language type derivation if its source had a type derivation in the source language.

The Curry-Howard isomorphism will assist you in constructing these translations. We know that the type $\tau_1 + \tau_2$ corresponds to the formula $T_1 \vee T_2$ and the type $\tau_1 * \tau_2$ corresponds to the formula $T_1 \wedge T_2$. From De Morgan's rules in classical logic, we know that $T_1 \vee T_2 \equiv \neg(\neg T_1 \wedge \neg T_2)$, and vice versa.

To help you in your efforts, we have provided on the course web page an interpreter for the polymorphic lambda calculus. See the file `test.poly` for example syntax.

(a) (5 pts.) As mentioned in class, types whose formulae contain negation can be generated by using continuations, but our target language has no continuations. Give a formula that is equivalent to $A \wedge B$ but contains only logical operators for which there is a target-sum language equivalent. Give a formula that is equivalent to $A \vee B$ but contains only logical operators for which there is a target-product language equivalent.

(b) (5 pts.) We translate a source language type τ into a target-sum language type $\mathcal{T}^+[\![\tau]\!]$ and into a target-product language type $\mathcal{T}^*[\![\tau]\!]$. What logically equivalent target-sum language type should $\mathcal{T}^+[\![\cdot]\!]$ map $\tau_1 * \tau_2$ to? What logically equivalent target-product language type should $\mathcal{T}^*[\![\cdot]\!]$ map $\tau_1 + \tau_2$ to?

(c) (10 pts.) In this part, you will define the important parts of a translation function $\mathcal{E}^+[\![\cdot]\!]$, which when applied to a source language type judgment $\Delta; \Gamma \vdash e : \sigma$, produces a provable target-sum language type judgment $\Delta'; \Gamma' \vdash e' : \mathcal{T}^+[\![\sigma]\!]$. 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 translation of type checking rules for $\langle e_1, e_2 \rangle$, $\mathsf{left}\ e$, and $\mathsf{right}\ e$ into type checking rules for target-sum languages. You will need to introduce new variables; add any premises needed to control the selection of variable names.

(d) (10 pts.) Similarly, you can define $\mathcal{E}^*[\![\cdot]\!]$ (and $\mathcal{G}^*[\![\cdot]\!]$) to translate a source language type judgment into a target-product language type judgment.

Complete the translation of type checking rules for $\mathsf{inl}_{\tau_1+\tau_2} e$, $\mathsf{inr}_{\tau_1+\tau_2} e$, and $\mathsf{case}\ e_0\ \mathsf{of}\ e_1 | e_2$ into type checking rules for target-product languages.

(e) (10 pts.) Show that the expressions that are your translations in 3(c) and 3(d) 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.