Cornell University
Computer Science 611

Problem Set 5 DUE: 3 December 2004

Instructions

Turn in the written part of the assignment by 5PM on the due date in Upson 4119. The original due date has been extended until December 3 by popular demand. However, be aware that Problem Set 6 will be released on Monday, November 29, and will also be due on the same day. You are advised not to wait until November 29 to start working on this problem set!

1. Logical relations (40 pts)

   Consider the CBV simply-typed lambda calculus extended by products, $\lambda^{\rightarrow *}$.

   The types are:

   $$\tau ::= \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \tau_2$$

   The terms are:

   $$e ::= x \mid e_1\ e_2 \mid n \mid \lambda x{:}\tau.\, e \mid (e_1, e_2) \mid \text{left } e \mid \text{right } e$$

   The values are:

   $$v ::= \lambda x{:}\tau.\, e \mid (v_0, v_1) \mid n$$

   The evaluation contexts are the usual:

   $$E[\cdot] ::= [\cdot]\ e \mid v\ [\cdot] \mid ([\cdot], e) \mid (v, [\cdot]) \mid \text{left } [\cdot] \mid \text{right } [\cdot]$$

   We have the usual rule:

   $$\frac{e_1 \longrightarrow e_2}{E[e_1] \longrightarrow E[e_2]}$$

   And the reductions:

   $$(\lambda x{:}\tau.\, e)v \longrightarrow e\{v/x\} \quad \text{left } (v_0, v_1) \longrightarrow v_0 \quad \text{right } (v_0, v_1) \longrightarrow v_1$$

   The type system is standard, with the rules:

   $$\frac{}{\Gamma \vdash n : \text{int}} \quad \frac{}{\Gamma, x{:}\tau \vdash x : \tau}$$

   $$\frac{\Gamma \vdash e_0 : \sigma \rightarrow \tau \quad \Gamma \vdash e_1 : \sigma}{\Gamma \vdash e_0\ e_1 : \tau} \quad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x{:}\sigma.\, e : \sigma \rightarrow \tau} \quad \frac{\Gamma \vdash e_0 : \tau_1 \quad \Gamma \vdash e_1 : \tau_2}{\Gamma \vdash (e_0, e_1) : \tau_1 * \tau_2}$$

   $$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \text{left } e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \text{right } e : \tau_2}$$

   (a) Try to prove the strong normalization theorem (for each expression e such that $\vdash e : \tau$ for some type τ, there is no infinite sequence of reduction steps starting from e) for $\lambda^{\rightarrow}$ (*not* for $\lambda^{\rightarrow *}$) by structural induction on e. Prove all the cases which are amenable to this proof and point clearly where the proof fails.

(b) The proof in the previous section failed, which perhaps isn't that surprising, given that the not-so-obvious technique of logical relations has been introduced at the lecture to prove this fact. You will use the same technique to do the working proof of strong normalization for[1] $\lambda^{\rightarrow *}$.

Give the appropriate definition of the relation R_τ for $\lambda^{\rightarrow *}$.

(c) Give the appropriate extension of the definition of the substitution operator $\hat{\gamma}$ for $\lambda^{\rightarrow *}$. You may refer to the rules given in the lecture (from 2001) for $\hat{\gamma}$ for $\lambda^{\rightarrow}$, including them in your writeup is not necessary.

(d) Prove by the induction on the height of the derivation (in other words, by induction on the typing rules) the following claim:

$$\forall \Gamma \vdash e : \tau.\ (\gamma \models \Gamma \Rightarrow \hat{\gamma}(e) \in R_\tau)$$

This claim implies trivially the strong normalization for $\lambda^{\rightarrow *}$. State clearly every part of the proof, in particular the inductive hypothesis and where it's used. You may find the following fact useful:

$$(e \in R_\tau \wedge e \rightarrow e') \Leftrightarrow (e' \in R_\tau \wedge e \rightarrow e')$$

You may use it as a lemma without proving it.

2. Maybe types (30 pts)

In many languages (e.g., C, Java) it is convenient to have a special "null" value that acts like a member of any reference type that is desired. However, the possibility that every reference may turn out to be null also creates difficulties for both the programmer and the language implementer. A neat way to have the expressive power of null without the undesirable side effects is to introduce a special type constructor maybe that effectively augments any type τ with a special null value $\langle\rangle$. Because the null value can be represented by a distinguished pointer value, a τ maybe is easily implemented just as compactly as a C pointer or a Java reference. In this problem you will develop the semantics of maybes.

We start with the typed lambda calculus $\lambda^{\rightarrow}$ and extend it as follows:

$$\begin{array}{lll} \tau ::= \ldots & | & \tau \text{ maybe} \\ e ::= \ldots & | & \langle e \rangle \mid \langle\rangle \mid \text{if } \langle x \rangle = e_0 \text{ then } e_1 \text{ else } e_2 \\ v ::= \ldots & | & \langle v \rangle \mid \langle\rangle \end{array}$$

Informally, the extensions works as follows. The new introduction form $\langle e \rangle$ injects the value of e into the corresponding maybe type. The introduction form $\langle\rangle$ is the special null value. The special if form checks whether an expression e_0 evaluates to a non-empty maybe; if so, the expression e_1 is evaluated with x bound to the injected value. If not, the expression e_2 is evaluated instead.

(a) Give rules defining the new reductions needed for the extended language.

(b) Assuming left-to-right evaluation and the values given above, define how to extend the legal evaluation contexts $E[\cdot]$ in which reductions can occur.

(c) Give any new typing rules that are required for the extended language.

(d) Define the weakest sound subtyping relationship on types τ maybe and τ' maybe and justify it by defining the appropriate coercion function.

[1] The proof easily extends to $\lambda^{\rightarrow *+}$. What is more remarkable is that the same technique can be extended to prove the strong normalization for $\lambda 2$ (System F), the lambda-calculus with impredicative polymorphism, and many other very powerful type systems.

(e) Do the same for τ maybe and τ. Why would such a subtype relationship be helpful?

(f) Give a typed translation from this language ($\lambda^{\rightarrow}$ extended with maybe) to the language $\lambda^{\rightarrow+}$. It should translate type derivations in the source language to terms with type derivations in the target language, inductively demonstrating that any well-typed source term produces a well-typed target term.

3. Type inference (30 pts)

The source file inference.sml contains a partial implementation of type inference for a simple SML-like language (but without let-polymorphism). Finish this implementation. You can change anything you want but you should only have to write code in the places that say "IMPLEMENT ME". Function declarations may be recursive in SML, and in this language too. Submit the completed implementation via CMS. You may work with another student on this implementation.