Exercise 1 (Recursive let)

Recursive let expressions are one way (besides Y-combinators) to add recursion to $\lambda\to$.

\[ t := x \mid (t_1 \ t_2) \mid (\lambda x. \ t) \mid \text{letrec } x = t_1 \ \text{in} \ t_2 \]

a) Modify the standard typing rule for let to create a suitable rule for letrec.

b) Considering type inference, what is the problematic property of this rule compared to the rule for let?

Solution

a) The rule for letrec is like the rule for let, but we also add $x$ to $\Gamma$ when checking $t_1$.

\[
\frac{\Gamma \vdash x : \sigma_1 \quad \Gamma \vdash t_1 : \sigma_1 \quad \Gamma \vdash t_2 : \sigma_2}{\Gamma \vdash \text{letrec } x = t_1 \ \text{in} \ t_2 : \sigma_2} \quad \text{LETREC}
\]

Alternatively, we can combine this rule with the $\forall$-intro typing rule:

\[
\frac{\{\alpha_1 \ldots \alpha_n\} = FV(\tau) \setminus FV(\Gamma) \quad \Gamma \vdash x : \forall \alpha_1 \ldots \alpha_n. \tau \vdash t_1 : \tau \quad \Gamma \vdash x : \forall \alpha_1 \ldots \alpha_n. \tau \vdash t_2 : \tau_2}{\Gamma \vdash \text{letrec } x = t_1 \ \text{in} \ t_2 : \tau_2} \quad \text{LETREC’}
\]

b) The interesting property of this new typing rule is that we cannot know which $\alpha_1 \ldots \alpha_n$ we need to generalize $\tau$ over before we have inferred $\tau$ (the type of $t_1$). Thus, typical compilers will only allow $x$ to be used monomorphically in $t_1$. Alternatively, the user can explicitly specify a type schema for $x$, so that it can be used polymorphically.

Exercise 2 (Type Inference in Haskell (2))

Extend the implementation of the type inference algorithm from the last exercise with let and letrec constructs.

Solution

See type_inference_let.hs.
Exercise 3 (Peirce’s Law in Intuitionistic Logic)

Prove the following variant of Peirce’s Law in intuitionistic logic:

\[ (((P \rightarrow Q) \rightarrow P) \rightarrow P) \rightarrow Q \]

Solution

Let \( A_3 = (P \rightarrow Q) \rightarrow P \), \( A_2 = A_3 \rightarrow P \), and \( A_1 = A_2 \rightarrow Q \).

\[
\begin{array}{c}
A_1, A_3, P \vdash A_1 \\
\hline
A_1, A_3, P \vdash A_2 \\
\hline
A_1, A_3, P \vdash P \rightarrow Q \\
\hline
A_1, A_3 \vdash P \\
\hline
A_1, A_3 \vdash A_3 \\
\hline
A_1 \vdash A_2 \\
\hline
A_1 \vdash Q \\
\hline
\vdash (((P \rightarrow Q) \rightarrow P) \rightarrow P) \rightarrow Q \\
\end{array}
\]

\[ \rightarrow I \]

\[ \rightarrow E \]

\[ \rightarrow E \]

\[ \rightarrow I \]

\[ \rightarrow E \]

\[ \rightarrow I \]
Homework 4 (Fixed-point combinator)
Let
$$ S = \lambda abcdefghijklmnopqrstuvwxyz. r(thisisafixedpointcombinator) $$
and
$$ E = \text{thisisafixedpointcombinator} $$
Show that $E$ is a fixed-point combinator.

Homework 5 (let-Polymorphism)
Give a derivation tree for the following statement, and so determine the type $\tau$:
$$ [z : \tau_0] \vdash \text{let } x = \lambda y z. z y y \text{ in } x (x z) : \tau $$

Homework 6 (Constructive Logic)

a) Prove the following statement using the calculus for intuitionistic propositional logic:
$$ ((c \to b) \to b) \to (c \to a) \to ((a \to b) \to b) $$

Hint: To make your proof tree more compact, you may remove unneeded assumptions to the left of the $\vdash$ during the proof as you see fit. For example, the following step is valid:
$$ \frac{p \vdash p}{p, q \vdash p} $$

b) Give a well-typed expression in $\lambda \rightarrow$ with the type
$$ ((\gamma \to \beta) \to \beta) \to (\gamma \to \alpha) \to ((\alpha \to \beta) \to \beta) $$
(You don’t need to give the derivation tree.)