Exercise 1 (Recursive \texttt{let})

Recursive \texttt{let} expressions are one way (besides \texttt{Y}-combinators) to add recursion to \(\lambda\rightarrow\).

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

a) Modify the standard typing rule for \texttt{let} to create a suitable rule for \texttt{letrec}.

b) Considering \textit{type inference}, what is the problematic property of this rule compared to the rule for \texttt{let}?

c) Give a derivation tree for the following statement, and so determine the type \(\tau\):

\[
[] \vdash \texttt{letrec} \ x = \lambda y. \ x \ (x \ y) \ \text{in} \ x \ x : \tau
\]

Solution

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

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

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

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

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.

c) Abbreviations: \(\Gamma_1 = [x : \forall \alpha. \alpha \rightarrow \alpha]\) and \(\Gamma_2 = [x : \forall \alpha. \alpha \rightarrow \alpha, y : \alpha]\).

\[
\frac{\Gamma_2 \vdash x : \alpha \rightarrow \alpha}{\text{VAR'}} \quad \frac{\Gamma_2 \vdash x : \alpha \rightarrow \alpha}{\text{VAR'}} \quad \frac{\Gamma_2 \vdash y : \alpha}{\text{VAR'}} \quad \frac{\Gamma_2 \vdash x y : \alpha}{\text{APP}} \quad \frac{\Gamma_2 \vdash x (x y) : \alpha}{\text{APP}} \quad \frac{\Gamma_2 \vdash \lambda y. \ x (x y) : \alpha \rightarrow \alpha}{\text{ABS}}
\]

see above

\[
\frac{\Gamma_1 \vdash \lambda y. \ x (x y) : \alpha \rightarrow \alpha}{\text{VAR'}} \quad \frac{\Gamma_1 \vdash (\lambda x. \ x) : (\beta \rightarrow \beta) \rightarrow \beta}{\text{VAR'}} \quad \frac{\Gamma_1 \vdash x : (\beta \rightarrow \beta)}{\text{VAR'}} \quad \frac{\Gamma_1 \vdash x : \beta \rightarrow \beta}{\text{VAR'}} \quad \frac{\Gamma_1 \vdash \texttt{letrec} \ x = \lambda y. \ x (x y) \ \text{in} \ x \ x : \beta \rightarrow \beta}{\text{LETREC'}}
\]
Exercise 2 (Type Inference in Haskell (2))

In this exercise, we will extend the implementation of the type inference algorithm from last exercise for the `let` and `letrec` constructs.

a) Have a look at the template on the website, which adds `let` and `letrec` to the term language.

b) Extend the type inference algorithm for `let` and `letrec`.

Solution

See `type_inference_let.hs`.

Homework 3 (Fixed-point combinator)

Let

$$S = \lambda abcd efghi jklmnopqstuvwxyvr. r \ (this\ is\ a\ fixed-point\ combinator)$$

and

$$E = \$\$$

Show that $E$ is a fixed-point combinator.

Homework 4 (`let`-Polymorphism)

Give a derivation tree for the following statement, and so determine the type $\tau$:

$$[z : \tau_0] \vdash \text{let } x = \lambda yz. z \ y \ \text{in } (x \ z) : \tau$$

Homework 5 (Type Inference with Type Constructors)

We generalize $\to$ to type constructors. With type constructors, types are either elementary, a type variable or, constructed as $\Pi \ \tau_1 \ldots \tau_n$ where $\Pi$ is a type constructor. Now $\to$ is just a type constructor that takes two arguments. Your task is to extend the Haskell inference algorithm towards type constructors.

a) Extend the type language with type constructors. Type constructors should take lists of type arguments.

b) Extend the type inference algorithm for type constructors. To specify the set of valid type constructors, we will just start with a non-empty environment that will pre-specify the type of some free variables that then act as data constructors.