Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Simon Wimmer

Lambda Calculus
Winter Term 2017/18
Solutions to Exercise Sheet 10

Exercise 1 (Recursive `let`)

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

$$t ::= x \mid (t_1\ t_2) \mid (\lambda x.\ t) \mid \texttt{letrec}\ x = t_1\ \texttt{in}\ t_2$$

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`?

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

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

Solution

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

$$\frac{\Gamma[x : \sigma_1] \vdash t_1 : \sigma_1 \qquad \Gamma[x : \sigma_1] \vdash t_2 : \sigma_2}{\Gamma \vdash (\texttt{letrec}\ x = t_1\ \texttt{in}\ t_2) : \sigma_2}\ \textsc{LetRec}$$

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

$$\frac{\begin{array}{c}\{\alpha_1 \dots \alpha_n\} = FV(\tau) \setminus FV(\Gamma) \\ \Gamma[x : \forall \alpha_1 \dots \alpha_n.\tau] \vdash t_1 : \tau \qquad \Gamma[x : \forall \alpha_1 \dots \alpha_n.\tau] \vdash t_2 : \tau_2\end{array}}{\Gamma \vdash \texttt{letrec}\ x = t_1\ \texttt{in}\ t_2 : \tau_2}\ \textsc{LetRec'}$$

b) The interesting property of this new typing rule is that we cannot know which $\alpha_1 \dots \alpha_n$ we need to generalize τ over before we have inferred τ (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: $\Gamma_1 = [x : \forall \alpha.\ \alpha \to \alpha]$ and $\Gamma_2 = [x : \forall \alpha.\ \alpha \to \alpha, y : \alpha]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash x : \alpha \to \alpha}\ \textsc{Var'} \quad \dfrac{\dfrac{}{\Gamma_2 \vdash x : \alpha \to \alpha}\ \textsc{Var'} \quad \dfrac{}{\Gamma_2 \vdash y : \alpha}\ \textsc{Var'}}{\Gamma_2 \vdash x\ y : \alpha}\ \textsc{App}}{\Gamma_2 \vdash x\ (x\ y) : \alpha}\ \textsc{App}}{\Gamma_1 \vdash \lambda y.\ x\ (x\ y) : \alpha \to \alpha}\ \textsc{Abs}$$

$$\dfrac{\dfrac{\text{see above}}{\Gamma_1 \vdash \lambda y.\ x\ (x\ y) : \alpha \to \alpha} \quad \dfrac{\dfrac{}{\Gamma_1 \vdash x : (\beta \to \beta) \to \beta \to \beta}\ \textsc{Var'} \quad \dfrac{}{\Gamma_1 \vdash x : \beta \to \beta}\ \textsc{Var'}}{\Gamma_1 \vdash x\ x : \beta \to \beta}\ \textsc{App}}{[] \vdash\ \texttt{letrec}\ x = \lambda y.\ x\ (x\ y)\ \texttt{in}\ x\ x : \beta \to \beta}\ \textsc{LetRec'}$$

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

$$\$ = \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$$

and

$$€ = \$.$$

Show that € is a fixed-point combinator.

Homework 4 (`let`-Polymorphism)

Give a derivation tree for the following statement, and so determine the type τ:

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

Homework 5 (Type Inference with Type Constructors)

We generalize $\rightarrow$ to type constructors. With type constructors, types are either elementary, a type variable or, constructed as $\Pi\ \tau_1 \ldots \tau_n$ where Π is a type constructor. Now $\rightarrow$ 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.