

Solutions for Unassessed Exercise Sheet 5+

Exercise 1: Type rules

letrec can be regarded as syntactic sugar in ML (which has a built-in **let** construct):

$$\mathbf{letrec} \ f = M \ \mathbf{in} \ N \quad \equiv \quad \mathbf{let} \ f = \mathbf{Y} (\lambda f. M) \ \mathbf{in} \ N$$

Show that it has a derived type rule:

The following typing derivation can be produced for the right hand side term form:

$$\frac{\frac{\frac{}{\Gamma \vdash \mathbf{Y} : \forall t. (t \rightarrow t) \rightarrow t} \text{Const} \quad \frac{\Gamma, f : T \vdash M : T}{\Gamma \vdash \lambda f. M : T \rightarrow T} \rightarrow \text{Intro}}{\Gamma \vdash \mathbf{Y} : (T \rightarrow T) \rightarrow T} \forall \text{Elim}}{\Gamma \vdash \mathbf{Y} (\lambda f. M) : T} \rightarrow \text{Elim} \quad \frac{\Gamma \vdash \mathbf{Y} (\lambda f. M) : T}{\Gamma \vdash \mathbf{Y} (\lambda f. M) : \forall \vec{t}. T} \forall \text{Intro} \quad \frac{}{\Gamma, f : \forall \vec{t}. T \vdash N : T'} \text{LET}}{\Gamma \vdash \mathbf{let} \ f = \mathbf{Y} (\lambda f. M) \ \mathbf{in} \ N : T'}$$

Hence the derived type rule for **letrec** is:

$$\frac{\Gamma, f : T \vdash M : T \quad \Gamma, f : \forall \vec{t}. T \vdash N : T'}{\Gamma \vdash \mathbf{letrec} \ f = M \ \mathbf{in} \ N : T'}$$

Exercise 2: Type inference

Given below is the definition of a function *take* which selects the first few elements of a list:

$$\mathbf{letrec} \ take = \lambda n. \lambda l. \ \mathbf{if} \ (n = 0) \ \mathbf{nil} \\ \quad \quad \quad (\mathbf{cons} \ (\mathbf{car} \ l) \ (\mathbf{take} \ (n - 1) \ (\mathbf{cdr} \ l)))$$

Calculate the principal type (most general type) of *take*, using type inference.

The general method is to assume unknown types t_1 and t_2 for the two parameters (n and l respectively) and type check the body of λ . This gives constraints on the type unknowns. Since we have a recursive definition, the function name *take* itself used in the body. So, we assume an unknown type t_0 for the type of *take*. In addition, each of the polymorphic constants used in the body should be given a *generic instance* of the polymorphic type, using fresh type variables for the quantified type variables. We shall use the following generic instances:

$$\begin{aligned} \mathbf{if} & : \ \mathbf{bool} \rightarrow t_3 \rightarrow t_3 \rightarrow t_3 \\ \mathbf{nil} & : \ \mathbf{list} \ t_4 \\ \mathbf{cons} & : \ t_5 \rightarrow \mathbf{list} \ t_5 \rightarrow \mathbf{list} \ t_5 \\ \mathbf{car} & : \ \mathbf{list} \ t_6 \rightarrow t_6 \\ \mathbf{cdr} & : \ \mathbf{list} \ t_7 \rightarrow \mathbf{list} \ t_7 \end{aligned}$$

Type checking the body gives the following constraints:

$$\begin{aligned} t_1 & = \ \mathbf{int} \\ t_3 & = \ \mathbf{list} \ t_4 \\ t_5 & = \ t_4 \\ t_6 & = \ t_4 \\ t_2 & = \ \mathbf{list} \ t_6 = \mathbf{list} \ t_4 \\ t_7 & = \ t_4 \\ t_0 & = \ \mathbf{int} \rightarrow \mathbf{list} \ t_4 \rightarrow \mathbf{list} \ t_4 \end{aligned}$$

The λ function is then of type:

$$\mathbf{int} \rightarrow \mathbf{list} \ t_4 \rightarrow \mathbf{list} \ t_4$$

which matches the assumed type t_0 for *take*. So the **letrec** definition type checks.

Renaming the type variables, we get the principal type of *take* to be

$$\mathbf{int} \rightarrow \mathbf{list} \ t \rightarrow \mathbf{list} \ t$$

Exercise 3: Explicit polymorphism

- a. Rewrite the definition of *take* in the explicitly-typed polymorphic lambda calculus.

$$\mathbf{letrec} \ take = \Lambda t. \lambda n : \mathbf{int}. \lambda l : \mathbf{list} \ t. \text{ if } [\mathbf{list} \ t] \ (n = 0) \ (\mathbf{nil} \ [t]) \\ (\mathbf{cons} \ [t] \ (\mathbf{car} \ [t] \ l) \ (take \ [t] \ (n - 1) \ (\mathbf{cdr} \ [t] \ l)))$$

- b. Rewrite it again in Java with generics. (Assume that list objects have methods `null`, `car` and `cdr`, as in Handout 6, paragraph 11.)

```
static <t> List<t> take(int n, List<t> l) {
    if (n == 0) {
        return <t>nil; }
    else {
        return <t>cons(l.car(), <t>take(n-1, l.cdr())); }
}
```

Exercise 4: Higher-order functions

- a. Write the recursive definition of a higher-order function *filter*, which takes as arguments a boolean function p and a list l , and returns a list of all the elements in l for which p is true.

$$\mathbf{letrec} \ filter = \lambda p. \lambda l. \text{ if } (\mathbf{null} \ l) \\ \mathbf{nil} \\ \text{if } (p \ (\mathbf{car} \ l)) \\ (\mathbf{cons} \ (\mathbf{car} \ l) \ (filter \ p \ (\mathbf{cdr} \ l))) \\ (filter \ p \ (\mathbf{cdr} \ l))$$

- b. Calculate the principal type of *filter*.

The principal type is:

$$(t \rightarrow \mathbf{bool}) \rightarrow \mathbf{list} \ t \rightarrow \mathbf{list} \ t$$

There are no constraints on the element type of the list except that the boolean function p should be applicable to them. So, making the argument type p to be the same as the element type of l is enough to satisfy the constraint.

- c. Rewrite *filter* in the explicitly-typed polymorphic lambda calculus.

$$\mathbf{letrec} \ filter = \Lambda t. \lambda p : t \rightarrow \mathbf{bool}. \lambda l : \mathbf{list} \ t. \text{ if } [\mathbf{list} \ t] \ (\mathbf{null} \ [t] \ l) \\ (\mathbf{nil} \ [t]) \\ \text{if } [\mathbf{list} \ t] \ (p \ (\mathbf{car} \ [t] \ l)) \\ (\mathbf{cons} \ [t] \ (\mathbf{car} \ [t] \ l) \ (filter \ [t] \ p \ (\mathbf{cdr} \ [t] \ l))) \\ (filter \ [t] \ p \ (\mathbf{cdr} \ [t] \ l))$$

Exercise 5: Higher-order functions

- a. Write the recursive definition of a higher-order function *map*, which takes as arguments a unary function f and a list l , and returns a list of values obtained by applying f to the elements of l .

$$\mathbf{letrec} \ map = \lambda f. \lambda l. \text{ if } (\mathbf{null} \ l) \\ \mathbf{nil} \\ (\mathbf{cons} \ (f(\mathbf{car} \ l)) \ (map \ f \ (\mathbf{cdr} \ l)))$$

- b. Calculate the principal type of *map*.

The principal type is:

$$(t \rightarrow u) \rightarrow \mathbf{list} \ t \rightarrow \mathbf{list} \ u$$