Prof. Dr. Peter Thiemann

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Winter Term 2024/25

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 12

Exercise 1 (Hindley-Milner Type Inference)

In this exercise sheet, we are going to extend the lambda calculus interpreter from the previous exercise sheet with polymorphism and Hindley-Milner type inference.

This requires us to add a `let`-expression to our expression syntax, and extend the type syntax with type schemes:

```
data Exp = ...
         | ELet EVar Exp Exp

data TypeScheme = TForall [Var] Type

type Ctx = [(Var, TypeScheme)]
```

Here are some examples of how the interpreter should behave:

```
lambda> (let id = (\x. x) in ((id id) 42))

Type:
  Int
Reduction:
  (let id = (\x. x) in ((id id) 42))
  (((\x. x) (\x. x)) 42)
  ((\x. x) 42)
  42
Reduced to value.

lambda> (let x = (let y = 1 in (y + 2)) in (let z = (x + 3) in (z + 4)))

Type:
  Int
Reduction:
  (let x = (let y = 1 in (y + 2)) in (let z = (x + 3) in (z + 4)))
  (let z = ((let y = 1 in (y + 2)) + 3) in (z + 4))
  (((let y = 1 in (y + 2)) + 3) + 4)
  (((1 + 2) + 3) + 4)
  ((3 + 3) + 4)
  (6 + 4)
  10
Reduced to value.
```

Note, that the first example does not work without polymorphism: the `let` expression has a body `((id id) 42)` where the first `id` has type `(Int -> Int) -> (Int -> Int)` whereas the second `id` has type `Int -> Int`. This is only possible if the variable `id` has the polymorphic type $\forall\alpha.\ \alpha \to \alpha$ in the body of the `let`,

We recommend to proceed in the following steps:

1. Extend the parser to support `let`-expressions.

   For simplicity, we recommend to require explicit parentheses around `let`-expressions, as we did in the previous exercise sheet, e.g. `(\x. (let x = (1 + 2) in (x + 3)))`.

   Similar as with `true` and `false` you need to ensure that `let` and `in` cannot be parsed as variables.

2. Extend the `free` and substitution function with cases for `ELet`.

   Note, that the variable bound in a `ELet`-expression requires similar handling as the variable bound in a `ELam`-expression. Since `let x = e1 in e2` has the same meaning as `((\x. e2) e1)`, substitution should behave on `let x = e1 in e2` exactly as it does on a `((\x. e2) e1)` expression.

3. Extend the `step` function with a case for `ELet`. Since our interpreter does neither use a call-by-value nor a call-by-name evaluation strategy, but instead completely normalizes the expression, we only need a β-reduction rule for `ELet`, but no congruency rules. Recall that `let x = e1 in e2` should behave like `((\x. e2) e1)`, so it can always be reduced by substituting `e1` for `x` in `e2`, independent of whether `e1` or `e2` are already evaluated or not.

4. Extend the `pretty` function with a case for `ELet`.

5. Implement the type inference algorithm from the lecture.

   Computing the most general unifiers works the same as before and requires no change.

   Type substitution needs to be not only defined on types but additionally on type schemes: only those variables should be substituted which are free in the type scheme, i.e. which are not bound by the `TForall`.

   A new operation is required to compose two type substitutions:

   ```
   (.#) :: TypeSub -> TypeSub -> TypeSub
   ```

   If `s1 .# s2` is applied to a type, then it should behave like first applying `s2` to the type, and then applying `s1`, i.e. forall `s1` and `s2` and types `t` it should hold that

   ```
   substType (s1 .# s2) t == substType s1 (substType s2 t)
   ```

   This is analogous to how function composition behaves:

   ```
   applyFun (f1 . f2) x == applyFun f1 (applyFun f2 x)
   ```

   where `applyFun` is simply defined as

   ```
   applyFun f x = f x
   ```