Concepts of Programming Language Design

Polymorphism Exercises

Gabriele Keller

November 6, 2024

1. Find the MGU of the following pairs of type terms, if it exists:

   (a)
   - $(\texttt{a} \to \texttt{a}) * \texttt{Int}$ and $\texttt{b} * \texttt{c}$
   - $(\texttt{a} \to \texttt{a}) * \texttt{Int}$ and $\texttt{b} \to \texttt{c}$
   - $(\texttt{a} \to \texttt{a}) * \texttt{a}$ and $\texttt{b} * \texttt{b}$
   - $\texttt{Rec t.}(\texttt{t} + \texttt{Int})$ and $\texttt{Rec t.}(\texttt{t} + \texttt{a})$
   - $\texttt{Rec t.}(\texttt{t} + \texttt{Int})$ and $\texttt{Rec t.}(\texttt{a} + \texttt{Int})$

   Solution:

   - $MGU = [b := a \to a, c := Int]$
   - No unifier at all, since topmost type constructors differ
   - No unifier, since a has to be the same as b, and $a \to a$ can't be unified with a
   - $MGU = [a := Int]$
   - No unifier. We would get the same types if we replace the a of the second type with t. Note that this cannot be achieved via substitution, because applying substitution $[a := t]$ to term $\texttt{Rec t.}(\texttt{t} + \texttt{a})$ is undefined, as t occurs freely in the right hand side of the substitution.

2. **Parametric polymorphism** In the lecture, we introduced two sets of typing rules for an implicitly typed variant of MinHs. While the first set of rules specified when a certain polymorphic type can be inferred, it did not describe an algorithm to infer the principal type of an expression, in contrast to the second set, which is a formalisation of the Hindley-Milner type inference algorithm.

   (a) For the non-algorithmic typing rules, the rule for the `Pair` constructor is exactly the same as for explicitly typed MinHs. For each set of rules, show that

   $$\{\} \vdash \texttt{Apply(Recfun } (f.x.\texttt{Pair } x\ x)) \ (\texttt{Num 5}) :: (\texttt{Int} * \texttt{Int})$$

   is derivable.

   (b) Consider this (explicitly typed) program (we're using Haskell notation here):

   ```
   g :: forall b. ((forall a. (a -> a)) -> b -> Int)
   g f x = if f True
             then (f 1)
             else 2
   ```

   The type of `g` is not a legal type in polymorphic MinHs as discussed in the lecture. Which restriction did we put on MinHs which excludes this type?

   Solution: Polymorphic types are not first class: the $\forall$- quantifier can only appear at the outermost position. The type annotation requires `f` to be polymorphic (as opposed to any function which has the same argument and result type)

   (c) In a language which permits this type, what can you tell about the behaviour of the function `g`?

Solution: I `f` is truly polymorphic, then it is either the identity function or a non-terminating function (or one which triggers a runtime error). If it returns a result, it will therefore always be `1`.

(d) The higher-order abstract syntax term below represents this program:

```
Recfun (g.f. (Recfun (h.x. (If (Apply f (Const True)) (Num 1) (Num 2)))))
```

Is the type $\forall b.((\forall a.(a \to a)) \to b \to Int)$ for this expression inferable with the non-algorithmic or the algorithmic typing rules? Explain why.

Solution: With the non-algorithmic rules, the `Recfun` rule requires us to make a "guess" as to what the (mono) type of the function and the argument are. If we allow the guessed types to be a polymorphic type, then we can derive the above type.

The algorithmic rules only ever allow us to add quantifiers at the outermost position of a type, and in contrast to the non-alogrithmic rules, there is no way to change without making them non-algorithmic anymore.

(e) What is the type of the expression as derived by Hindley-Milner?

Solution: $\forall a.(\texttt{Bool} \to \texttt{Bool}) \to a \to Int$

3. **Coercions and Subtyping**

You are given the type Rectangle, parameterised by its height and width, and the type Square parameterised by the length of one of its sides. Neither type is mutable.

(a) Which type is the subtype, which type is the supertype?

(b) Give a subtype/supertype ordering of the following set of function types:

```
Rectangle -> Rectangle
Rectangle -> Square
Square -> Rectangle
Square -> Square.
```

(c) Define a data type Square and a data type Rectangle in Haskell. Then define a coercion function from elements of the subclass to elements of the superclass.

(d) Show that the ordering you have given in the previous question is correct by defining coercion functions for each pair of types in a subtyping relationship in part (b).

Solution: If we stick to a subset interpretation, square is a subtype of rectangle. For a coercion interpretation, we can go either way: we can convert a rectangle into a square by dropping either width or height (or the shortest of both, or the longest of both), although automatic coercion which changes an object significantly can be problematic. In particular, if there are other subtyping relationships, it is easy to end up with a non-coherent system (see the float, int string example in the lecture).

Depending on the choice the answers to the other questions vary. For the function subtyping, it is important to keep in mind that the function type constructor is contravariant in the first, covariant in the second argument.

4. **Constructor variance**

List some examples of a covariant, contravariant and invariant type constructor.

Solution: Co-variant: sum or pairs, contravariant the function type constructor in its first argument, invariant for example (updatable) array and reference types. Other examples in Haskell (all consequences of the previous):

```
data Covariant     a = Co     (Int -> a)
data Contravariant a = Contra (a   -> Int)
data Invariant     a = In     (a   -> a)
```