Resource Analysis: Problem Set 3

Jan Hoffmann
Carnegie Mellon University

February 5, 2016

Due before 1:30pm on Monday, February 8

3.1 (12 Points) Unique Types

Consider the following subset of the type rules of the monomorphic type system and the corresponding subset of syntactic forms.

$$\frac{\Gamma(x) = T}{\Gamma \vdash^m x : T}\ (\text{T:VAR}) \qquad \frac{\Gamma, x{:}T_1 \vdash^m e : T_2}{\Gamma \vdash^m \text{lam}(x.e) : T_1 \to T_2}\ (\text{T:ABS})$$

$$\frac{\Gamma \vdash^m e_1 : T_2 \to T \qquad \Gamma \vdash^m e_2 : T_2}{\Gamma \vdash^m \text{app}(e_1, e_2) : T}\ (\text{T:APP}) \qquad \frac{}{\Gamma \vdash^m \text{nil} : L(T)}\ (\text{T:NIL})$$

$$\frac{\Gamma \vdash^m e_1 : T \qquad \Gamma \vdash^m e_2 : \text{L}(T)}{\Gamma \vdash^m \text{cons}(e_1, e_2) : \text{L}(T)}\ (\text{T:CONS})$$

$$\frac{\Gamma \vdash^m e : \text{L}(T') \qquad \Gamma \vdash^m e_1 : T \qquad \Gamma, x_1{:}T', x_2{:}\text{L}(T') \vdash^m e_2 : T}{\Gamma \vdash^m \text{matL}(e, e_1, (x_1, x_2).e_2) : T}\ (\text{T:MATL})$$

$$\frac{\Gamma, f{:}T_1 \to T_2, x{:}T_1 \vdash^m e_f : T_2 \qquad \Gamma, f{:}T_1 \to T_2 \vdash^m e : T}{\Gamma \vdash^m \text{rec}((f, x).e_f, f.e) : T}\ (\text{T:REC})$$

a) Show that there are expressions e such that $\Gamma \vdash^m e : T$ and $\Gamma \vdash^m e : T'$ for monomorphic types $T \neq T'$ and a type context Γ.

b) Add a minimal number of type annotations to the syntactic forms to make types unique.

c) Prove that your annotations make types unique: If $\Gamma \vdash^m e : T$ and $\Gamma \vdash^m e : T'$ in the modified type system then $T = T'$. For example, the rule T:PAIR with type annotations could look as follows.

$$\frac{\Gamma \vdash^m e_1 : T_1 \qquad \Gamma \vdash^m e_2 : T_2}{\Gamma \vdash^m \text{pair}(e_1 : T_1, e_2 : T_2) : T_1 * T_2}\ (\text{T:PAIR})$$

d) Prove that your annotations are minimal: Show that every type system that you obtain after removing one annotation has the property stated in part (a).

3.2 (18 Points) Unification in OCaml

A naive implementation of the declarative unification rules that we discussed in the lecture can be quite inefficient. To make type inference efficient, the developers of the OCaml compiler took the following approach. First, types are represented by the following mutually recursive type.

```
type type_exp =
  { mutable desc : type_desc; mutable mark : int }

and type_desc =
    Var
  | Arr of type_exp * type_exp
  | Pair of type_exp * type_exp
  | Link of type_exp
```

The record type `type_exp` contains to mutable fields. Mutable fields are a convenient short form and equivalent to fields that contain references. See `https://realworldocaml.org/v1/en/html/records.html` . (In fact, references are derived from mutable records.)

The mutual recursion in the type is a common pattern in OCaml code. It makes it more convenient to access (and update) data that is shared among all constructors of an inductive type. The type $a * a \to b$ can be represented as follows. Note that type-variable names are implicitly represented by "heap addresses".

```
let type_exp d : type_exp = {desc = d; mark = 0} in

let a = type_exp Var in
let b = type_exp Var in
let s = type_exp (Pair(a, a)) in
type_exp (Arr(s, b))
```

Values of type `type_exp` can by used as a union-find data structure.

```
(* union find operations *)
let rec find (t : type_exp) : type_exp =
    match t.desc with
    | Link l →
        let r = find l
        in t.desc ← Link r;
           r
    | _ → t

let union (t1 : type_exp) (t2 : type_exp) : unit =
  (find t1).desc ← Link (find t2)
```

Recall the rule U:ELIM2:

$$\frac{\sigma = \{X \mapsto T\} \qquad X \notin \mathrm{Var}(T)}{\{\langle X, T\rangle\} \cup C \implies \{\langle X, T\rangle\} \cup \widehat{\sigma}(C)} \text{ (U:ELIM2)}$$

As the first step of OCaml's unification algorithm, the rule E:ELIM2 is basically implemented by a call to the function `union` without checking the side condition $X \notin \mathrm{Var}(T)$. In this way, a graph structure is created on the heap. The record field `mark` is not used in this first step. In the second, step the graph structure is traversed to identify cycles. If a cycle is found then the types are not unifyable. Otherwise, the unifyied type can be derived by following the pointers on the heap (see function `string_of_type` in `unify.ml`).

a) Implement OCaml's efficient unification algorithm by completing the OCaml file `unify.ml` that is available on the web page.

b) Argue informally why this approach is equivalent to the declarative rules presented in the lecture.

c) In your solution, provide three (different) pairs of (different) values of type `type_exp` that are not unifyable. Similarly, provide three pairs of values of type `type_exp` that are unifyable.