Programmation 1

TD n°13

15 décembre 2020

1 Unification et typage

Exercise 1 :

The relation $\leq$ on the substitutions is not antisymmetric.

1. Show that $\sigma \leq \tau \wedge \tau \leq \sigma$ if and only if σ and τ differ only by a renaming.

2. Show that if there is a solution to a unification problem, there is only one most general (except renaming).

> **Solution:**
>
> 1. (Just some algebraic manipulation)
> 2. We need to show the following two conditions (as with any rewriting system)
>   — The system terminates.
>   — The system preserves the set of solutions.
>
>   (Ref. to class notes) It is interesting to check if the set of substitutions with $\leq$ is a DCPO. Is the set of solutions directed? This would give a proof of existence which does not use an effective algorithm.

Exercise 2 :

Apply the "naive" (exponential) unification algorithm seen below (see Figure 1) to the follo-

$$(E \cup \{f(s_1, \ldots, s_m) \doteq f(t_1, \ldots, t_m)\}, \theta) \to (E \cup \{s_1 \doteq t_1, \ldots, s_m \doteq t_m\}, \theta) \qquad \text{(Dec)}$$

$$(E \cup \{f(s_1, \ldots, s_m) \doteq g(t_1, \ldots, t_n)\}, \theta) \to \mathsf{Fail} \qquad \text{si } f \neq g \qquad \text{(DecFail)}$$

$$(E \cup \{x \doteq x\}, \theta) \to (E, \theta) \qquad \text{(Triv)}$$

$$(E \cup \{x \doteq t\}, \theta) \to (E[x := t], \theta[x := t]) \qquad \text{si } x \notin \mathrm{fv}(t) \qquad \text{(Bind)}$$

$$(E \cup \{t \doteq x\}, \theta) \to (E[x := t], \theta[x := t]) \qquad \text{si } x \notin \mathrm{fv}(t) \qquad \text{(Bind')}$$

$$(E \cup \{x \doteq t\}, \theta) \to \mathsf{Fail} \qquad \text{si } t \neq x \in \mathrm{fv}(t) \qquad \text{(Check)}$$

$$(E \cup \{t \doteq x\}, \theta) \to \mathsf{Fail} \qquad \text{si } t \neq x \in \mathrm{fv}(t) \qquad \text{(Check')}$$

FIGURE 1 – Algorithme d'unification de ROBINSON.

wing systems of equations. Can you find unifiers other than the mgu? **Note :** The symbols $\dot{3}, \dot{5}$, etc. are to be seen as nullary functions.

1. $\{y \doteq f(x, z), y \doteq f(\dot{3}, \dot{5})\}$
2. $\{f(g(x)) \doteq f(z), g(z) \doteq g(g(\dot{3}))\}$
3. $\{\mathsf{a}(x, x) \doteq \mathsf{a}(\mathbf{int}, \mathsf{a}(\mathbf{int}, \mathbf{int}))\}$
4. $\{f(x) \doteq f(f(f(x)))\}$
5. $\{\alpha \doteq \beta \to \beta, \beta \doteq \gamma \to \gamma, \gamma \doteq \delta \to \delta\}$

> **Solution:**
> (Easy application.)

Exercise 3 :

1. Show that the algorithm seen before (c.f. Figure 1) is necessarily exponential.
2. Propose a data structure for the mgu which circumvents the problem mentioned in the previous question.
3. Propose a modification of the rules of the naive algorithm adapted to this new structure.
4. What is the complexity of the algorithm obtained?

> **Solution:**
>
> 1. It suffices to construct a problem E for which the output is of exponential size. For example, $x_i = f(x_{i+1}, x_{i+1})$ will give a complete binary tree for x_0 of depth n.

> 2. The idea is to use a directed acyclic graph, this gives a linear representation of the previously exponential term.
>
> 3. The rules are essentially the same, but you have to simply move pointers around to preserve equalities.

Exercise 4 :

1. Give an example of a closed term from pureML that does not type into monomorphic pureML.

2. Give an example of a closed term which does not type in pureML but which does not reduce to Wrong.

> **Solution:**
> 1. We must use polymorphism, for example, by constructing (`f 3, f "trois"`).
> 2. The same works because in fact the semantics does not care about types.

Exercise 5 :

Imagining the natural generalization of the pureML typing rules, type the given program :

```
let r = ref (fun x -> x)
in
  r := (fun n -> n+1);
    !r "abc" ;;
```

Is it well-typed ?

> **Solution:**
> It is indeed well-typed, but still wrong !

Exercise 6 :

Write a function `length` in OCaml for the following type :

```
type 'a mycroft =
  | Nil
  | Cont of 'a * ('a list) mycroft
```

Explain.

> **Solution:**
> The problem is that the type changes in the meantime, and therefore we cannot build the function ! It is the same thing as with the classic type :
>
> `type 'a bush = Nil | Cont of 'a * ('a bush bush)`
>
> But, in OCaml we can write this function if we do as follows
>
> ```
> let rec f : type a. a mycroft -> int = function
>       Nil -> 0
>     | Cont (_,m) -> 1 + f m
> ```