A5 : SIGNATURES, TERMS, SUBSTITUTIONS, UNIFIERS

file : assignment5.ml

```
type symbol = Constsymbol of int | Funcsymbol of char ;;
type term = Variable of char | Node of symbol * (term list) ;;

(* example of terms : x , y , (+, [x;y]) , etc.
      where   x, y  are  Variable
*)
let one = Constsymbol 1 ;;
let zero = Constsymbol 0 ;;
let oR = Funcsymbol '|' ;;
let aND = Funcsymbol '&' ;;
let nOT = Funcsymbol '~' ;;

(* example : let trm = Node (aND, [Node (oR, [x;z]) ; y]) ;;
      where  let x = Variable 'x' ;;  let y = Variable 'y';
             let z = Variable 'z' ;;
*)

let rec validSymbol sym = match sym with          (* checks if the input symbol is valid *)
      | Constsymbol x -> true
      | Funcsymbol x -> true
;;
let rec symbolExists elem lst = match lst with    (* checks if symbol 'elem' exists in a list 'lst' of [symbol, arity] *)
      | [] -> false
      | (hd_sym , hd_ar)::tl -> (elem = hd_elem) ||
                                (symbolExists elem tl)
;;
let rec dupSymbolExists lst = match lst with      (* does lst contain duplicate symbols *)
      | [] -> false
      | (hd_sym, hd_ar)::tl -> (symbolExists hd_sym tl) ||
                               (dupSymbolExists tl)
;;
** let rec check_sig sig_set = match sig_set with   (* checks if sig_set is a valid signature *)
      | [] -> true
      | (sym, ar) :: tl -> (validSymbol sym) &&
                           (ar >= 0) &&
                           (((symbolExists sym tl) || (dupSymbolExists tl)) = false)
;;
```

file : assignment5.ml

Page-2

```
(* sig_set is the representation of a signature
   which is basically a list of (symbol, int) pair
   e.g. let test_sig = [(zero, 0) ; (one, 1) ; (nOT, 1) ; (aND, 2)]
*)

let rec arity sym signature =                      (* gives the arity
        match signature with                          of 'sym' w.r.t a
        | [] -> -1                                    'signature'. Returns
        | (sym1, ar) :: tl -> if (sym = sym1)         -1 if 'sym' doesn't
                              then ar                 exist in 'signature'
                              else (arity sym tl) ;;  *)

let test_sig = [(zero, 0) ; (one, 1) ; (nOT, 1) ; (aND, 2) ; (OR, 2)] ;;
        (* global signature *)

** let rec wfterm trm =                            (* checks if the term
        match trm with                                'trm' if well-formed w.r.t
        | Variable v -> true                          the signature test_sig (global)
                                                      *)
        | Node (sym, trm_lst) -> (validSymbol sym) &&
                                 (List.length trm_lst = (arity sym test_sig)) &&
                                 (let rec wflist l =
                                     match l with
                                     | [] -> true
                                     | hd :: tl -> (wfterm hd) && (wflist tl)
                                  in
                                  wflist trm_lst )
;;
```

file : assignment5.ml

```
** let rec ht t = ~~mte~~ match t with        (* returns the ~~heigt~~ height
     | Variable v -> 0                            of the tree representing
     | Node (Constsymbol x, l) -> 0               the term 't' *)
     | Node (AND, [t1; t2]) -> 1 + (max (ht t1) (ht t2))
     | Node (OR, [t1; t2]) -> 1 + (max (ht t1) (ht t2))
     | Node (NOT, [t1]) -> 1 + (ht t1)
     ;;

** let rec size t = match t with              (* returns the size of
      | Variable v -> 1                            term 't' *)
      | Node (Constsymbol x, l) -> 1
      | Node (sym, [t1; t2]) -> 1 + (size t1) + (size t2)
      | Node (sym, [t1]) -> 1 + (size t1)
      ;;

let rec filter_uniq l = match l with          (* returns the
     | [ ] -> l                                    list 'l' with all
     | hd::tl -> if (List.memq hd tl)              its duplicate
                 then (filter_uniq tl)             elements removed *)
                 else hd :: (filter_uniq tl)
     ;;

** let rec vars t =                            (* returns the list
     let res = match t with                       of unique variables
        | Variable v -> [v]                       in term 't' *)
        | Node (sym, [t1; t2]) -> List.append (vars t1) (vars t2)
        | Node (sym, [t1]) -> vars t1
     in
     filter_uniq res ;;
```

(* Representation of substitutions :

{ x ↦ t } is represented as (x, t)

and a substitution is a list of such (x, t) pairs

where x is Variable (term)

t is term

∴ ~~type substitution~~ = ~~Substitution of~~

*)

file : assignment5.ml

```
let rec subst_var v s =                    (* substitutes the Variable 'v'
    match s with                              w.r.t. the substitution 's' *)
    | [] -> Variable v
    | (Variable v1, Variable t1)::tl -> if v=v1 then Variable t1
                                        else (subst_var v tl)
    | (Variable v1, Node (s1, l))::tl -> if v=v1
                                         then Node (s1, l)
                                         else (subst_var v tl)
;;

** let rec subst t s =                     (* substitutes the term 't'
      match t with                            w.r.t. the substitution
                                              's' *)
    | Variable v -> subst_var v s
    | Node (sym, [t1; t2]) -> Node (sym, [(subst t1 s); (subst t2 s)])
    | Node (sym, [t1]) -> Node (sym, [(subst t1 s)])
    | _ -> t
;;

let rec append_tuple s (x1, t1) =          (* appends the tuple
    match s with                              (x1, t1) to tuple-list
    | [] -> [(x1, t1)]                        's' if (x1, t1) doesn't
    | (x2, t2) :: tl -> if x1 = x2 then []    already exist in 's'
                        else (append_tuple tl (x1, t1))   *)
;;

let rec append_uniq s1 s2 =                (* appends the substitutions 's1'
    math s2 with                              and 's2' such that the
    | [] -> s1                                resulting substitution is valid *)
    | hd :: tl -> List.append (append_tuple s1 hd) (append_uniq s1 tl)
;;

** let rec compose s1 s2 =                 (* returns the composition of
    let temp =                                the substitutions 's1' and 's2'
    (match s1 with                            *)
        | [] -> []
        | (v, t) :: tl -> (v, (subst t s2)) :: (compose tl s2)
    )
    in append_uniq temp s2
;;
```

file : assignment5.ml

```
exception NOT_UNIFIABLE ;;

let rec occurs v l =                    (* 'occurs-check' for mgu :
    match l with                           checks returns true if the
    | [] -> false                          variable 'v' occurs in the
                                           term list 'l' *)
    | (Node (sym, l1)) :: tl -> (occurs v l1) || (occurs v tl)
    | (Variable v1) :: tl -> if v = v1 then true
                             else (occurs v tl)

;;

let rec mgu t1 t2 =                     (* returns the mgu of terms 't1'
                                           & 't2' if they are unifiable
                                           else raises exception *)
    match (t1, t2) with
    | (Variable v1, Variable v2) -> if v1 = v2 then []
                                    else [(Variable v1, Variable v2)]

    | (Variable v1, Node (sym, l)) -> if (occurs v1 l)
                                      then raise NOT_UNIFIABLE
                                      else [(Variable v1, Node (sym, l))]

    | (Node (sym, l), Variable v2) -> if (occurs v2 l)
                                      then raise NOT_UNIFIABLE
                                      else [(Variable v1, Node (sym, l))]

    | (Node (sym1, l1), Node (sym2, l2)) -> if (sym1 = sym2)
                then
                match (l1, l2) with
                | ([tr1], [tr2]) -> mgu tr1 tr2
                | ([tr1; tr2], [tr3; tr4]) -> List.append
                                              (mgu tr1 tr3)
                                              (mgu tr2 tr4)

                else raise NOT_UNIFIABLE

;;
```

------- END -------

Sumit Singh
2016PH10575