```
(* Terms *)
(* Simplest Conceivable Version *)
(* Author: Frank Pfenning *)

signature TERM =
sig

  (* Terms *)
  datatype Term =                              (* t ::=             *)
    Var of int                                 (*        x          *)
  | Con of string * Term list                  (*      | f(t1,...,tn) *)

end;

structure Term :> TERM =
struct

  (* Terms *)
  datatype Term =                              (* t ::=             *)
    Var of int                                 (*        x          *)
  | Con of string * Term list                  (*      | f(t1,...,tn) *)

end;
```

```
(* Supporting data types for indexing *)
(* Simple version *)
(* Author: Frank Pfenning *)

(* Ordered Keys *)
signature ORDER =
sig
  type key
  val compare : key * key -> order
end;

(* Ordered Sets *)
signature SET =
sig
  structure O : ORDER

  type set

  val empty : set
  val insert : O.key * set -> set

  val union : set * set -> set
  val intersect : set * set -> set

  val size : set -> int
end;
```

```
(* Index Signature *)
(* Persistent *)
(* No deletion for simplicity *)
signature INDEX =
sig

  structure S : SET

  type index

  val empty : index
  val insert : index * Term.Term * S.set -> index

  val getEq : index -> Term.Term -> S.set
  val getGen : index -> Term.Term -> S.set
  val getInst : index -> Term.Term -> S.set
  val getUnif : index -> Term.Term -> S.set

end;  (* signature INDEX *)
```

```
(* Path Indexing *)
(* Author: Frank Pfenning *)

(*
   Based on tries to store paths in term.
   Data structures are persistent.

   Indexes only linear terms (all variables are considered
   distinct) and is therefore not a perfect filter.

   Children are just ordered lists (not tries or hashtables).

   Currently does not implement a delete operation.

   Index is a mapping from terms to sets.

   {} --- empty index
   {} (t) = {} for all terms t

   index + {t' |=> set} --- index extension
   (index + {t' |=> set})(t) = index (t) if t <> t'
   (index + {t' |=> set})(t') = index (t') Union set
*)
```

```
functor PathIndex
  (structure Set : SET)
  : INDEX where type S.O.key = Set.O.key
           where type S.set = Set.set =
struct

  structure S = Set

  (* key k ::= "string", ordered lexicographically *)
  (* integer values would be more efficient *)
  type key = string                        (* var has key "" *)

  (* Tries *)
  datatype Trie =
      Node of (key * Child) list           (* node n, ordered by key *)
  and Child =                              (* child ::=              *)
      Var of S.set                         (*           *            *)
    | Con of S.set                         (*         | c            *)
    | Fun of Trie list                     (*         | f(n1,...,nk) *)

  type index = Trie

  (* key (t) ==> k, the key of t *)
  fun key (Term.Var _) = ""
    | key (Term.Con (f, _)) = f
```

```
(* empty ==> {} *)
val empty = Node []

(* insert (index, term, set) ==> index + {term |=> set} *)
fun insert (trie, term, set) =
let
  fun createNode (t) = Node [(key(t), createChild(t))]
  and createChild (Term.Con (c, nil)) = Con (set)
    | createChild (Term.Con (f, ts)) = Fun (createNodes ts)
    | createChild (Term.Var _) = Var (set)
  and createNodes (nil) = nil
    | createNodes (t::ts) = createNode(t)::createNodes(ts)
```

```
    fun insert (Node (chn), t) = Node (insertChn (chn, (key(t), t)))
    and insertChn (nil, (k, t)) = (k, createChild t)::nil
      | insertChn (chn as (k’, child)::chn’, (k, t)) =
        (case String.compare (k, k’)
           of EQUAL => (k’, insertChild (child, t))::chn’
            | LESS => ((k, createChild (t))::chn)
            | GREATER => (k’, child)::insertChn (chn’, (k, t)))
    and insertChild (Var (set’), _) = Var (S.union(set, set’))
      | insertChild (Con (set’), _) = Con (S.union(set, set’))
      | insertChild (Fun (ns), Term.Con (f, ts)) =
          Fun (inserts (ns, ts))
    and inserts (nil, nil) = nil
      | inserts (n::ns, t::ts) = insert (n, t)::inserts (ns, ts)
      (* other cases impossible if terms are arity-correct *)
  in
    insert (trie, term)
  end
```

```
(* getEq index term ==> set if index(term) = set *)
fun getEq (Node (chn)) t = getEqChn ((key(t), t), chn)
and getEqChn (_, nil) = S.empty
  | getEqChn ((k, t), (k', child)::chn') =
    (case String.compare (k, k')
       of EQUAL => getEqChild (child, t)
        | LESS => S.empty
        | GREATER => getEqChn ((k, t), chn'))
and getEqChild (Var (set'), _) = set'
  | getEqChild (Con (set'), _) = set'
  | getEqChild (Fun (ns), Term.Con (_, ts)) =
      getEqs (ns, ts)
(* getEqs (ns, ts), |ns| = |ts| >= 1 *)
and getEqs (n::nil, t::nil) = getEq n t
  | getEqs (n::ns, t::ts) =
      S.intersect (getEq n t, getEqs (ns, ts))
```

```
(* getGen index term ==>
   Union {set | index(t) = set and t[sigma] = term}
*)
fun getGen (Node (chn)) t = getGenChn ((key(t), t), chn)
and getGenChn (_, nil) = S.empty
  | getGenChn ((k, t), (k', Var (set'))::chn') =
      S.union (set', getGenChn ((k, t), chn'))
  | getGenChn ((k, t), (k', child)::chn') =
    (case String.compare (k, k')
       of EQUAL => getGenChild (child, t)
        | LESS => S.empty
        | GREATER => getGenChn ((k, t), chn'))
and getGenChild (Con (set'), _) = set'
  | getGenChild (Fun (ns), Term.Con (_, ts)) =
      getGens (ns, ts)
  (* getGenChild (child, t), child <> Var _ *)
(* getGens (ns, ts), |ns| = |ts| >= 1 *)
and getGens (n::nil, t::nil) = getGen n t
  | getGens (n::ns, t::ts) =
      S.intersect (getGen n t, getGens (ns, ts))
```

```
(* getAll index ==>
   Union {set | index(t) = set}
*)
fun getAll (Node (chn)) = getAllChn (chn)
and getAllChn (nil) = S.empty
  | getAllChn ((k, child)::chn’) =
      S.union (getAllChild (child), getAllChn (chn’))
and getAllChild (Var (set’)) = set’
  | getAllChild (Con (set’)) = set’
  | getAllChild (Fun (ns)) = getAlls (ns)
and getAlls (n::nil) = getAll (n)
  | getAlls (n::ns) =
      S.union (getAll (n), getAlls (ns))
```

```
(* getInst index term ==>
   Union {set | index(t) = set and t = term[sigma]}
*)
fun getInst n (Term.Var _) = getAll (n)
  | getInst (Node (chn)) t = getInstChn ((key(t), t), chn)
and getInstChn (_, nil) = S.empty
  | getInstChn ((k, t), (k', child)::chn') =
    (case String.compare (k, k')
       of EQUAL => getInstChild (child, t)
        | LESS => S.empty
        | GREATER => getInstChn ((k, t), chn'))
and getInstChild (Con (set'), _) = set'
  | getInstChild (Fun (ns), Term.Con (_, ts)) =
      getInsts (ns, ts)
  (* getInstChild (child, t), child <> Var _ *)
(* getInsts (ns, ts), |ns| = |ts| >= 1 *)
and getInsts (n::nil, t::nil) = getInst n t
  | getInsts (n::ns, t::ts) =
      S.intersect (getInst n t, getInsts (ns, ts))
```

```
  (* getUnif index term ==>
     Union {set | index(t) = set and t[tau] = term[sigma]}
  *)
  (* direct combination of Gen and Inst *)
  fun getUnif n (Term.Var _) = getAll (n)
    | getUnif (Node (chn)) t = getUnifChn ((key(t), t), chn)
  and getUnifChn (_, nil) = S.empty
    | getUnifChn ((k, t), (k', Var(set'))::chn') =
        S.union (set', getUnifChn ((k, t), chn'))
    | getUnifChn ((k, t), (k', child)::chn') =
      (case String.compare (k, k')
         of EQUAL => getUnifChild (child, t)
          | LESS => S.empty
          | GREATER => getUnifChn ((k, t), chn'))
  and getUnifChild (Con (set'), _) = set'
    | getUnifChild (Fun (ns), Term.Con (_, ts)) =
        getUnifs (ns, ts)
    (* getUnifChild (child, t), child <> Var _ *)
  (* getUnifs (ns, ts), |ns| = |ts| >= 1 *)
  and getUnifs (n::nil, t::nil) = getUnif n t
    | getUnifs (n::ns, t::ts) =
        S.intersect (getUnif n t, getUnifs (ns, ts))

end;
```

```
(* Path Index Structure over Integer Sets *)
structure PathIndex
  : INDEX where type S.O.key = int
          where type S.set = IntSet.set =
  PathIndex (structure Set = IntSet);
```

```
(* Discrimination Trees *)
(* Author: Frank Pfenning *)

functor DiscIndex
  (structure Set : SET)
  : INDEX where type S.O.key = Set.O.key
           where type S.set = Set.set =
struct

  structure S = Set

  (* key k ::= "string", ordered lexicographically *)
  (* integer values would be more efficient *)
  type key = string                    (* var has key "" *)

  (* Tries *)
  (* node n ::= (set_, [(k1, a1, n1),...])
     list of children ordered by key ki
     ai is arity of function symbol with key ki
     ni is subtrie below ki
  *)
  datatype Trie =
    Node of S.set option * (key * int * Trie) list
```

```
type index = Trie

(* key (t) = k *)
(* More efficient would be to assign integer to each constant *)
fun key (Term.Var _) = ""
  | key (Term.Con (f, _)) = f

(* arity (t) = a, the number of immediate subterms in t *)
fun arity (Term.Var _) = 0
  | arity (Term.Con (f, ts)) = List.length ts

(* args (t) = ts, the list of immediate subterms of t *)
fun args (Term.Var _) = nil
  | args (Term.Con (f, ts)) = ts
```

```
(* empty ==> {} *)
val empty = Node (NONE, [])

(* insert (index, term, set) ==> index + {term |=> set} *)
fun insert (trie, term, set) =
let
  fun createNode (t, stack) =
        Node (NONE, [(key(t), arity(t),
                     createNodes (args(t), stack))])
  and createNodes (nil, nil) = Node (SOME(set), nil)
    | createNodes (nil, ts::stack) =
        createNodes (ts, stack)
    | createNodes (t::ts, stack) =
        createNode (t, ts::stack)
```

```
    fun insert (t, stack, Node (set’, chn)) =
          Node (set’,
                insertChn (key(t), arity(t), args(t), stack, chn))
    and insertChn (k, a, ts, stack, nil) =
          (k, a, createNodes (ts, stack))::nil
      | insertChn (k, a, ts, stack, chn as (k’, a’, subtrie)::chn’) =
        (case String.compare (k, k’)
           of EQUAL => (* a = a’ *)
                      (k, a, inserts (ts, stack, subtrie))::chn’
            | LESS => (k, a, createNodes (ts, stack))::chn
            | GREATER => (k’, a’, subtrie)
                         ::insertChn (k, a, ts, stack, chn’))
    and inserts (nil, nil, Node (NONE, subtries)) =
          Node (SOME(set), subtries)
      | inserts (nil, nil, Node (SOME(set’), subtries)) =
          Node (SOME(Set.union (set, set’)), subtries)
      | inserts (nil, ts::stack, n) = inserts (ts, stack, n)
      | inserts (t::ts, stack, n) = insert (t, ts::stack, n)
  in
    insert (term, nil, trie)
  end
```

```
(* getEq index term ==> set if index(term) = set *)
fun getEq n t =
let
  fun getEq (t, stack, Node (set', chn)) =
        getEqChn (key(t), arity(t), args(t), stack, chn)
  and getEqChn (k, a, ts, stack, nil) = S.empty
    | getEqChn (k, a, ts, stack, chn as (k', a', subtrie)::chn') =
      (case String.compare (k, k')
         of EQUAL => getEqs (ts, stack, subtrie)
          | LESS => S.empty
          | GREATER => getEqChn (k, a, ts, stack, chn'))
  and getEqs (nil, nil, Node (NONE, subtries)) = S.empty
    | getEqs (nil, nil, Node (SOME(set'), subtries)) = set'
    | getEqs (nil, ts::stack, n) = getEqs (ts, stack, n)
    | getEqs (t::ts, stack, n) = getEq (t, ts::stack, n)
in
  getEq (t, nil, n)
end
```

```
(* getGen index term ==>
   Union {set | index(t) = set and t[sigma] = term}
*)
fun getGen n t =
let
  fun getGen (t, stack, Node (set’, chn)) =
        getGenChn (key(t), arity(t), args(t), stack, chn)
  and getGenChn (k, a, ts, stack, nil) = S.empty
    | getGenChn (k, a, ts, stack, chn as ("", 0, subtrie)::chn’) =
      (* variable *)
      S.union (getGens (nil, stack, subtrie),
               getGenChn (k, a, ts, stack, chn’))
    | getGenChn (k, a, ts, stack, chn as (k’, a’, subtrie)::chn’) =
      (case String.compare (k, k’)
         of EQUAL => getGens (ts, stack, subtrie)
          | LESS => S.empty
          | GREATER => getGenChn (k, a, ts, stack, chn’))
  and getGens (nil, nil, Node (NONE, subtries)) = S.empty
    | getGens (nil, nil, Node (SOME(set’), subtries)) = set’
    | getGens (nil, ts::stack, n) = getGens (ts, stack, n)
    | getGens (t::ts, stack, n) = getGen (t, ts::stack, n)
in
  getGen (t, nil, n)
end
```

```
(* getInst index term ==>
   Union {set | index(t) = set and t = term[sigma]}
*)
fun getInst n t =
let
  fun getInst (Term.Var _, stack, n) =
      (* skip all subterms rooted at n,
         match stack against remainder
      *)
        skip (stack, n, 0)
    | getInst (t, stack, Node (set’, chn)) =
        getInstChn (key(t), arity(t), args(t), stack, chn)
  and getInstChn (k, a, ts, stack, nil) = S.empty
    | getInstChn (k, a, ts, stack,
                  chn as (k’, a’, subtrie)::chn’) =
      (case String.compare (k, k’)
         of EQUAL => getInsts (ts, stack, subtrie)
          | LESS => S.empty
          | GREATER => getInstChn (k, a, ts, stack, chn’))
  and getInsts (nil, nil, Node (NONE, subtries)) = S.empty
    | getInsts (nil, nil, Node (SOME(set’), subtries)) = set’
    | getInsts (nil, ts::stack, n) = getInsts (ts, stack, n)
    | getInsts (t::ts, stack, n) = getInst (t, ts::stack, n)
```

```
    and skip (stack, Node (set', chn), j) =
          skipChn (stack, chn, j)
    and skipChn (stack, nil, j) = S.empty
      | skipChn (stack, (k', 0, subtrie)::chn, 0) =
          S.union (getInsts (nil, stack, subtrie),
                   skipChn (stack, chn, 0))
      | skipChn (stack, (k', a', subtrie)::chn, j) =
          S.union (skip (stack, subtrie, j+a'-1),
                   skipChn (stack, chn, j))
  in
    getInst (t, nil, n)
  end

...
end;  (* functor DiscIndex *)
```