```
(* 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;
```

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

(* Integers *)
structure IntOrder : ORDER where type key = int =
struct
  type key = int
  val compare = Int.compare
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;

(*
   Ordered sets as ordered lists
   Some alternative implementations: dictionaries, bit vectors
*)
functor Set
  (structure Order : ORDER)
  : SET where type O.key = Order.key =
struct
 ...
end;
```

```
(* Sets of integers *)
structure IntSet : SET where type O.key = int =
  Set (structure Order = IntOrder);
```

```
(* Index Signature *)
(* Persistent *)
(* No deletion or unfication 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

end;  (* signature INDEX *)
```

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

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

(* empty : index *)
val empty = Node []

(* insert index (term, set) : index *)
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 of equal term labels *)
fun getEq (Node (chn)) t = getEqChn (chn, (key(t), t))
and getEqChn (nil, _) = S.empty
  | getEqChn ((k', child)::chn', (k, t)) =
    (case String.compare (k, k')
       of EQUAL => getEqChild (child, t)
        | LESS => S.empty
        | GREATER => getEqChn (chn', (k, t)))
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 ==> set of more general term labels *)
fun getGen (Node (chn)) t = getGenChn (chn, (key(t), t))
and getGenChn (nil, _) = S.empty
  | getGenChn ((k', Var (set'))::chn', (k, t)) =
      S.union (set', getGenChn (chn', (k, t)))
  | getGenChn ((k', child)::chn', (k, t)) =
    (case String.compare (k, k')
       of EQUAL => getGenChild (child, t)
        | LESS => S.empty
        | GREATER => getGenChn (chn', (k, t)))
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 ==> set of all terms labels in index *)
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 ==> set of less general term labels *)
  fun getInst n (Term.Var _) = getAll (n)
    | getInst (Node (chn)) t = getInstChn (chn, (key(t), t))
  and getInstChn (nil, _) = S.empty
    | getInstChn ((k, child)::chn', (k', t)) =
      (case String.compare (k, k')
         of EQUAL => getInstChild (child, t)
          | GREATER => S.empty
          | LESS => getInstChn (chn', (k', t)))
  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))

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);

structure T = Term;
...
val t1 = f [a, b];
val t2 = f [f [a, b], c];
val t3 = f [x, y];
val t4 = f [g [a], g[b]];
val t5 = f [a, f[a, b]];

structure I = PathIndex;
structure S = IntSet;

fun sgn (k) = S.insert (k, S.empty);

val i1 = I.insert (I.empty, t1, sgn(1));
val i2 = I.insert (i1, t2, sgn(2));
val i3 = I.insert (i2, t3, sgn(3));
val i4 = I.insert (i3, t4, sgn(4));
val i5 = I.insert (i4, t5, sgn(5));
```

```
val getEq = I.getEq i5;
val getGen = I.getGen i5;
val getInst = I.getInst i5;

val y1 = getEq (t1);
val y2 = getEq (t2);
val y3 = getEq (t3);
val y4 = getEq (t4);
val y5 = getEq (t5);

val n1 = getEq (f [a, a]);
val n2 = getEq (f [g [a], g[c]]);
val n3 = getEq (f [b, x]);

val g1 = getGen (f [a, f[a, b]]);

val l1 = getInst (f [a, x]);
```