```
(* Simple Generic Prover Interface *)
(* Author: Frank Pfenning *)

signature PROVER =
sig

  datatype Reason =
    NoProof                                     (* no proof exists *)
  | Bound                                       (* bound exceeded *)

  datatype Result =
    Success of int                              (* level at which found *)
  | Failure of Reason                           (* no proof exists *)

  (* prove (bound, G, A) = result *)
  val prove : int * P.Prop C.Ctx * P.Prop -> Result

end;  (* signature PROVER *)
```

```
(* Simple Prover Directly Based on Focusing *)
(* Author: Frank Pfenning *)

structure Focus :> PROVER =
struct

  datatype Reason =
    NoProof                                     (* no proof exists *)
  | Bound                                       (* bound exceeded *)

  datatype Result =
    Success of int                              (* level at which found *)
  | Failure of Reason                           (* no proof exists *)

  (* assume (G, A) = G,A *)
  (* since we do not yet build prove terms, hyp is no named *)
  fun assume (G, A) = C.Decl (G, ("", A))

  (* orResult (r1, r2) = r3 *)
  (* There is definitely no proof only if both choices
     definitely fail.  r1 and r2 must be failures.
  *)
  fun orResult (Failure (NoProof), Failure (NoProof)) =
        Failure (NoProof)
    | orResult (_, _) = Failure (Bound)
```

```
(* Search Combinators (Tactics) *)
(* Implementation via success continuations *)
type cont = unit -> Result
type tactic = cont -> Result

(* Disjunctive choice *)
infixr ORELSE
fun (F ORELSE G) : tactic =
    fn sc => orResult (F sc, G sc)

(* Failure *)
fun FAIL (reason) : tactic =
    fn sc => Failure (reason)

(* Conjunctive choice *)
infixr ANDALSO
fun (F ANDALSO G) : tactic =
    fn sc => F (fn () => G (fn () => sc ()))

(* Success *)
val SUCCEED : tactic =
    fn sc => sc ()
```

```
(* Focused Search *)
(* In all functions, n >= 0 is distance remaining before
   search is cut off

   A ::= P | A1 & A2 | T | A1 => A2 | A1 v A2 | F
   L ::= P | A1 => A2
   R ::= P | A1 v A2 | F

   G ::= . | G, A
   D ::= . | D, L
*)

(* searchR (n, D, G, A) searches D; G => A; . *)
fun searchR (0, D, G, A) = FAIL (Bound)
  | searchR (n, D, G, A) = (* n > 0 *)
      activeRight (n, D, G, A)

(* searchL (n, D, G, A) searches D; G => .; R *)
and searchL (0, D, G, R) = FAIL (Bound)
  | searchL (n, D, G, R) = (* n > 0 *)
      activeLeft (n, D, G, R)
```

```
(* activeRight (n, D, G, A)
   D; G => A; .  applies active right rules to A *)

and activeRight (n, D, G, P.And (A, B)) =
      activeRight (n, D, G, A)
      ANDALSO activeRight (n, D, G, B)
  | activeRight (n, D, G, P.True) =
      SUCCEED
  | activeRight (n, D, G, P.Implies (A, B)) =
      activeRight (n, D, assume (G, A), B)
  | activeRight (n, D, G, A) = (* A = P | A v B | F *)
      activeLeft (n, D, G, A)
```

```
(* activeLeft (n, D, G, R)
   D; G => .; R  applies active left rules to G *)

and activeLeft (n, D, C.Decl (G, (u, P.And (A, B))), R) =
      activeLeft (n, D, assume (assume (G, A), B), R)
  | activeLeft (n, D, C.Decl (G, (u, P.True)), R) =
      activeLeft (n, D, G, R)
  | activeLeft (n, D, C.Decl (G, (u, P.Or (A, B))), R) =
      activeLeft (n, D, assume (G, A), R)
      ANDALSO activeLeft (n, D, assume (G, B), R)
  | activeLeft (n, D, C.Decl (G, (u, P.False)), R) =
      SUCCEED
  | activeLeft (n, D, C.Decl (G, (u, L)), R) = (* L = P | A => B *)
      activeLeft (n, assume (D, L), G, R)

  (* next two cases transition to focusing phase *)
  | activeLeft (n, D, C.Null, R as P.Atom(P)) =
      pickLeft (n, D, D, R)
  | activeLeft (n, D, C.Null, R) = (* R = A v B | F *)
      focusRight (n, D, R)
      ORELSE pickLeft (n, D, D, R)
```

```
(* focusRight (n, D, A)
   D; . >> A; .  applies focus right rules to A *)

and focusRight (n, D, P.Or (A, B)) =
      focusRight (n, D, A)
      ORELSE focusRight (n, D, B)
  | focusRight (n, D, P.False) =
      FAIL (NoProof)
  | focusRight (n, D, A) = (* A = P | A1 & A2 | T | A1 => A2 *)
      searchR (n-1, D, C.Null, A)
```

```
(* pickLeft (n, D, D', R) where D' is prefix of D
   D; . => .; R  picks left focus formula from D' *)

and pickLeft (n, D, C.Decl (D', (u, L)), R) =
      focusLeft (n, D, L, R)
      ORELSE pickLeft (n, D, D', R)
  | pickLeft (n, D, C.Null, R) =
      FAIL (NoProof)

(* focusLeft (n, D, A, R)
   D; A >> .; R  applies focus left rules to A *)

and focusLeft (n, D, L as P.Atom(Q), R) =
    if P.eq (L, R) then SUCCEED
    else FAIL (NoProof)
  | focusLeft (n, D, P.Implies(A, B), R) =
      focusLeft (n, D, B, R)
      ANDALSO searchR (n-1, D, C.Null, A)
  | focusLeft (n, D, P.And (A, B), R) =
      focusLeft (n, D, A, R)
      ORELSE focusLeft (n, D, B, R)
  | focusLeft (n, D, P.True, R) =
      FAIL (NoProof)
  | focusLeft (n, D, A, R) = (* A = A1 v A2 | F *)
      searchL (n-1, D, assume (C.Null, A), R)
```

```
  (* prove (n, G, A) = r *)
  (* Uses iterative deepening to search for
     G => A, represented by .; G => A; .
     bounded by n.  Bound is decreased along every
     branch whenever search on a new sequent
     D; . => A; . or D; . => .; R is initiated.
  *)
  fun prove (bound, G, A) =
      let
        exception Done of int
        fun iterate (n) =
            if n > bound then Failure (Bound)
            else next (n, searchR (n+1, C.Null, G, A)
                          (fn () => raise Done(n)))
        and next (n, Failure (NoProof)) = Failure (NoProof)
          | next (n, Failure (Bound)) = iterate (n+1)
      in
        iterate (0)
        handle Done (n) => Success (n)
      end

end;  (* structure Focus *)
```