```
(* Natural Deduction *)
(* Version 0: Direct embedding in ML *)
(* Proof checking is ML type checking *)
(* Note that ML permits too many functions because of recursion
   and effects, so in addition to type-checking we have to
   verify by hand the the functional arguments to "Lam" and "Case"
   are pure
*)
(* Author: Frank Pfenning *)

signature ND0 =
sig

  type ('a, 'b) And
  type True
  type ('a, 'b) Implies
  type ('a, 'b) Or
  type False

  val Pair : 'a * 'b -> ('a, 'b) And
  val Fst : ('a, 'b) And -> 'a
  val Snd : ('a, 'b) And -> 'b
  val Unit : True
  val Lam : ('a -> 'b) -> ('a, 'b) Implies
  val App : ('a, 'b) Implies * 'a -> 'b
  val Inl : 'a -> ('a, 'b) Or
  val Inr : 'b -> ('a, 'b) Or
  val Case : ('a, 'b) Or * ('a -> 'c) * ('b -> 'c) -> 'c
  val Abort : False -> 'a

end;

structure ND0a :> ND0 =
struct
```

```
  type ('a, 'b) And = 'a * 'b
  type True = unit
  type ('a, 'b) Implies = 'a -> 'b
  datatype ('a, 'b) Or = inl of 'a | inr of 'b
  type False = unit  (* correct because of opaque ascription *)
  exception Impossible

  fun Pair (M1, M2) = (M1, M2)
  fun Fst (M1, M2) = M1
  fun Snd (M1, M2) = M2
  val Unit = ()
  fun Lam M = M
  fun App (M1, M2) = M1 M2
  fun Inl (M) = inl M
  fun Inr (M) = inr M
  fun Case (inl (M), N1, N2) = N1 M
    | Case (inr (M), N1, N2) = N2 M
  fun Abort () = raise Impossible

end;  (* structure ND0a *)

structure ND0 :> ND0 =
struct

  datatype ('a, 'b) And = Pair of 'a * 'b
  fun Fst (Pair (M1, M2)) = M1
  fun Snd (Pair (M1, M2)) = M2

  datatype True = Unit

  datatype ('a, 'b) Implies = Lam of ('a -> 'b)
  fun App (Lam (M1), M2) = M1 M2

  datatype ('a, 'b) Or =
      Inl of 'a
```

```
    | Inr of 'b

  fun Case (Inl (M), N1, N2) = N1 M
    | Case (Inr (M), N1, N2) = N2 M

  (* Even internally now type is empty *)
  structure Void :> sig type t end =
  struct
    type t = unit
  end
  exception Impossible

  type False = Void.t
  fun Abort (M:False) = raise Impossible

end;  (* structure ND0 *)
(* Propositions *)
(* Author: Frank Pfenning *)
(* Negation is treated as an abbreviation *)

signature PROP =
sig

  (* Propositions *)
  datatype Prop =                         (* A ::=          *)
    Atom of string                        (*      P         *)
  | And of Prop * Prop                    (*    | A1 & A2   *)
  | True                                  (*    | t         *)
  | Implies of Prop * Prop                (*    | A1 -> A2 *)
  | Or of Prop * Prop                     (*    | A1 v A2   *)
  | False                                 (*    | f         *)

  val Not : Prop -> Prop                  (* ~ A = A => f   *)

  (* eq (A, B) = true  iff  A = B as propositions *)
```

```
  val eq : Prop * Prop -> bool

end;  (* signature PROP *)

structure Prop :> PROP =
struct

  (* Propositions *)
  datatype Prop =                                     (* A ::=          *)
    Atom of string                                    (*      P         *)
  | And of Prop * Prop                                (*    | A1 & A2   *)
  | True                                              (*    | t         *)
  | Implies of Prop * Prop                            (*    | A1 -> A2 *)
  | Or of Prop * Prop                                 (*    | A1 v A2   *)
  | False                                             (*    | f         *)

  (* ~ A = A => f *)
  fun Not (A) = Implies (A, False)

  (* eq (A, B) = true  iff  A = B as propositions *)
  (* Could use built-in equality *)
  fun eq (Atom(p), Atom(q)) = (p = q)
    | eq (And(A1, A2), And(B1, B2)) = eq (A1, B1) andalso eq (A2, B2)
    | eq (True, True) = true
    | eq (Implies(A1, A2), Implies(B1, B2)) =
        eq (A1, B1) andalso eq (A2, B2)
    | eq (Or(A1, A2), Or(B1, B2)) =
        eq (A1, B1) andalso eq (A2, B2)
    | eq (False, False) = true
    | eq _ = false

end;  (* structure Prop *)

(* Global abbreviation: P = Prop *)
structure P = Prop;
```

```
(* Contexts *)
(* Author: Frank Pfenning *)
(*
   Variables (represented by name) are mapped to
   arbitrary values (normally propositions)
*)

signature CTX =
sig

  (* Contexts *)
  (* Later declarations shadow earlier ones *)
  datatype 'a Ctx =                         (* G ::=       *)
    Null                                    (*       .     *)
  | Decl of 'a Ctx * (string * 'a)          (*    | G, x:A *)

  val lookup : 'a Ctx * string -> 'a option

end;  (* signature CTX *)

structure Ctx :> CTX =
struct

  (* Contexts *)
  (* Later declarations shadow earlier ones *)
  datatype 'a Ctx =                         (* G ::=       *)
    Null                                    (*       .     *)
  | Decl of 'a Ctx * (string * 'a)          (*    | G, x:A *)

  (* lookup (G, x) = NONE if x not declared in G
                   = SOME(A) if x:A is last declaration of x in G
  *)
  fun lookup (Null, x) = NONE
    | lookup (Decl (G, (y, A)), x) =
      if (x = y) then SOME(A)
```

```
      else lookup (G, x)

end;  (* structure Ctx *)

(* Global abbreviation *)
structure C = Ctx;
(* Natural Deductions *)
(* Version 1: Valid proof terms have unique type *)
(* Proof checking proceeds bottom-up *)
(* Author: Frank Pfenning *)

signature ND1 =
sig

  (* Proof terms *)
  datatype Term =                         (* M ::=          *)
    Var of string                         (*      u         *)
  | Pair of Term * Term                   (*    | <M1,M2>   *)
  | Fst of Term                           (*    | fst M     *)
  | Snd of Term                           (*    | snd M     *)
  | Unit                                  (*    | <>        *)
  | Lam of (string * P.Prop) * Term       (*    | \u:A. M   *)
  | App of Term * Term                    (*    | M1 M2     *)
  | Inl of P.Prop * Term                  (*    | inl^A M   *)
  | Inr of P.Prop * Term                  (*    | inr^A M   *)
  | Case of Term * (string * Term) * (string * Term)
                                          (*    | case M of inl u1 => M1
                                                          | inr u2 => M2 *)
  | Abort of P.Prop * Term                (*    | abort^A M *)

  exception Invalid of string

  (* check (G, M, A) = () if G |- M : A, raises Invalid otherwise *)
  (* syn (G, M) = A       if G |- M : A, raises Invalid if no such A exists *)
  val check : P.Prop C.Ctx * Term * P.Prop -> unit
```

```
  val syn : P.Prop C.Ctx * Term -> P.Prop

end;  (* signature ND1 *)

structure ND1 :> ND1 =
struct

  (* Proof terms *)
  datatype Term =                             (* M ::=          *)
    Var of string                             (*       u        *)
  | Pair of Term * Term                       (*     | <M1,M2>  *)
  | Fst of Term                               (*     | fst M    *)
  | Snd of Term                               (*     | snd M    *)
  | Unit                                      (*     | <>       *)
  | Lam of (string * P.Prop) * Term           (*     | \u:A. M  *)
  | App of Term * Term                        (*     | M1 M2    *)
  | Inl of P.Prop * Term                      (*     | inl^A M  *)
  | Inr of P.Prop * Term                      (*     | inr^A M  *)
  | Case of Term * (string * Term) * (string * Term)
                                              (*     | case M of inl u1 => M1
                                                                | inr u2 => M2 *)
  | Abort of P.Prop * Term                    (*     | abort^A M *)


  exception Invalid of string

  (* check (G, M, A) = () if G |- M : A, raises Invalid otherwise *)
  (* syn (G, M) = A       if G |- M : A, raises Invalid if no such A exists *)
  fun check (G, M, A) =
      if P.eq (syn (G, M), A) then ()
      else raise Invalid("Mismatch between expected and synthesized type")

  and syn (G, Var(u)) =
      (case C.lookup (G, u)
         of SOME(A) => A
```

```
          | NONE => raise Invalid("Undeclared variable " ^ u))
    | syn (G, Pair(M1, M2)) = P.And(syn (G, M1), syn (G, M2))
    | syn (G, Fst(M)) =
      (case syn (G, M)
         of P.And(A, B) => A
          | _ => raise Invalid("Argument to Fst not product"))
    | syn (G, Snd(M)) =
      (case syn (G, M)
         of P.And(A, B) => B
          | _ => raise Invalid("Argument to Snd not product"))
    | syn (G, Unit) = P.True
    | syn (G, Lam((u, A), M)) = P.Implies(A, syn (C.Decl (G, (u, A)), M))
    | syn (G, App(M1, M2)) =
      (case syn (G, M1)
         of P.Implies(A, B) => (check (G, M2, A); B)
          | _ => raise Invalid("First argument to App not function"))
    | syn (G, Inl(B, M)) = P.Or(syn (G, M), B)
    | syn (G, Inr(A, M)) = P.Or(A, syn (G, M))
    | syn (G, Case(M, (u1, N1), (u2, N2))) =
      (case syn (G, M)
         of P.Or(A, B) =>
            let
              val C1 = syn (C.Decl(G, (u1, A)), N1)
              val C2 = syn (C.Decl(G, (u2, B)), N2)
            in
              if P.eq (C1, C2) then C1
              else raise Invalid("Branches of Case have different type")
            end
          | _ => raise Invalid("First argument to Case not a sum"))
    | syn (G, Abort(C, M)) =
      (case syn (G, M)
         of P.False => C
          | _ => raise Invalid("Argument to Abort not void"))

end;  (* structure ND1 *)
```

```
(* Natural Deductions *)
(* Version 1: Compact representation *)
(* Proof checking alternates bottom-up and top-down *)
(* Author: Frank Pfenning *)

signature ND2 =
sig

  (* Annotated proof terms *)
  (* Divided into introduction and elimination forms *)
  (* Note the special status of Case and Abort *)

  datatype ITerm =                        (* I ::=        *)
    Pair of ITerm * ITerm                 (*      <I1,I2> *)
  | Unit                                  (*    | <>      *)
  | Lam of string * ITerm                 (*    | \u. I   *)
  | Inl of ITerm                          (*    | inl I   *)
  | Inr of ITerm                          (*    | inr I   *)
  | Case of ETerm * (string * ITerm) * (string * ITerm)
                                          (*    | case E of inl u1 => I1
                                                          | inr u2 => I2 *)
  | Abort of ETerm                        (*    | abort E *)
  | Elim of ETerm                         (*    | E       *)
  and ETerm =                             (* E ::=        *)
    Var of string                         (*      u       *)
  | Fst of ETerm                          (*    | fst E   *)
  | Snd of ETerm                          (*    | snd E   *)
  | App of ETerm * ITerm                  (*    | E I     *)
  | Intro of ITerm * P.Prop               (*    | I : A   *)

  exception Invalid of string

  (* check (G, I, A) = () if G |- I : A, raises Invalid otherwise *)
  (* syn (G, E) = A       if G |- E : A, raises Invalid if no such A exists *)
  val check : P.Prop C.Ctx * ITerm * P.Prop -> unit
```

```
  val syn : P.Prop C.Ctx * ETerm -> P.Prop

end;  (* signature ND2 *)

structure ND2 :> ND2 =
struct

  (* Annotated proof terms *)
  (* Divided into introduction and elimination forms *)
  (* Note the special status of Case and Abort *)

  datatype ITerm =                        (* I ::=         *)
    Pair of ITerm * ITerm                 (*       <I1,I2> *)
  | Unit                                  (*     | <>      *)
  | Lam of string * ITerm                 (*     | \u. I   *)
  | Inl of ITerm                          (*     | inl I   *)
  | Inr of ITerm                          (*     | inr I   *)
  | Case of ETerm * (string * ITerm) * (string * ITerm)
                                          (*     | case E of inl u1 => I1
                                                           | inr u2 => I2 *)
  | Abort of ETerm                        (*     | abort E *)
  | Elim of ETerm                         (*     | E       *)
  and ETerm =                             (* E ::=         *)
    Var of string                         (*       u       *)
  | Fst of ETerm                          (*     | fst E   *)
  | Snd of ETerm                          (*     | snd E   *)
  | App of ETerm * ITerm                  (*     | E I     *)
  | Intro of ITerm * P.Prop               (*     | I : A   *)

  exception Invalid of string
  fun check (G, Pair(I1, I2), P.And(A1, A2)) =
        (check (G, I1, A1); check (G, I2, A2))
    | check (G, Unit, True) = ()
    | check (G, Lam(x, I), P.Implies (A, B)) =
        check (C.Decl (G, (x, A)), I, B)
```

```
    | check (G, Inl(I), P.Or(A, B)) = check (G, I, A)
    | check (G, Inr(I), P.Or(A, B)) = check (G, I, B)
    | check (G, Case(E, (x1, I1), (x2, I2)), C) =
        (case syn (G, E)
           of P.Or (A, B) => (check (C.Decl (G, (x1, A)), I1, C);
                              check (C.Decl (G, (x2, B)), I2, C))
            | _ => raise Invalid ("Case subject does not have sum type"))
    | check (G, Abort(E), C) =
        (case syn (G, E)
           of P.False => ()
            | _ => raise Invalid ("Abort subject does not have void type"))
    | check (G, Elim(E), C) =
      if P.eq (syn (G, E), C) then ()
      else raise Invalid("Mismatch between expected and synthesized type")
    | check _ =
        raise Invalid ("Introduction term does not match expected type")

  and syn (G, Var(u)) =
      (case C.lookup (G, u)
         of SOME(A) => A
          | NONE => raise Invalid("Undeclared variable " ^ u))
    | syn (G, Fst(E)) =
      (case syn (G, E)
         of P.And (A, B) => A
          | _ => raise Invalid("Argument to Fst not product"))
    | syn (G, Snd(E)) =
      (case syn (G, E)
         of P.And (A, B) => B
          | _ => raise Invalid("Argument to Snd not product"))
    | syn (G, App(E, I)) =
      (case syn (G, E)
         of P.Implies (A, B) => (check (G, I, A); B)
          | _ => raise Invalid("First argument to App not function"))
    | syn (G, Intro(I, A)) =
      (check (G, I, A); A)
```

```
end;  (* structure ND2 *)
(* Direct embeddings *)
(* Proof checking is ML type checking *)
local
  open ND0a
in
  (* (A => B => C) => (A & B => C) *)
  fun t1 () : (('a, ('b, 'c) Implies) Implies,
               (('a, 'b) And, 'c) Implies) Implies
    = Lam (fn u => Lam (fn w => App (App (u, Fst w), Snd w)))
end;

local
  open ND0
in
  (* (A => B => C) => (A & B => C) *)
  fun t1 () = Lam (fn u => Lam (fn w => App (App (u, Fst w), Snd w)))
end;

(* More flexible implementations *)

val A = P.Atom("A");
val B = P.Atom("B");
val C = P.Atom("C");

(* Bottom-up type-checker *)

local
  open P
  open ND1
in

  fun check1 (M, A) =
      (check (C.Null, M, A); print ("OK\n"))
```

```
      handle Invalid (msg) => print ("ERROR:" ^ msg ^ "\n")

  val u = ND1.Var("u")
  val w = ND1.Var("w")

  (* (A => B => C) => (A & B => C) *)
  val _ = check1 (Lam (("u", Implies(A, Implies (B, C))),
                       Lam (("w", And (A, B)),
                            (App (App (u, Fst w), Snd w)))),
                  Implies (Implies (A, Implies (B, C)),
                           Implies (And (A, B), C)))

end;

(* Bi-directional type-checker *)

local
  open P
  open ND2
in

  fun check2 (M, A) =
      (check (C.Null, M, A); print ("OK\n"))
      handle Invalid (msg) => print ("ERROR:" ^ msg ^ "\n")

  val u = Var("u")
  val w = Var("w")

  (* (A => B => C) => (A & B => C) *)
  val _ = check2 (Lam ("u", Lam ("w", Elim (App (App (u, Elim (Fst w)),
                                                    Elim (Snd w))))),
                  Implies (Implies (A, Implies (B, C)),
                           Implies (And (A, B), C)))
end;
```