Implementing Inference Rules in **Standard ML**

Jon Sterling

October 7, 2017

Operationalizing inference rules in a computer program for proof checking or proof search is an important skill. In this tutorial, we will explain and demonstrate some basic techniques for implementing forward and backward inference in the **LCF** style [2, 4, 3].

When implementing a logical system in a programming language, it is important to understand and minimize the size of the portion of this system which must be "trusted", i.e. on which the correctness of the implementation depends. This is usually achieved by designing a *trusted kernel* with an abstract type together with some operations for constructing elements of that type; then, the only way to produce an object of this type is by calling the provided functions.

An **LCF** kernel consists in such an abstract type `proof`, together with functions which construct proofs according to the rules of inference of the logic. Then, you can use any programming techniques you want (even unsafe ones) to produce proofs, and any such proof which is actually produced is guaranteed to be correct relative to the correctness of the kernel.

1 Representing Syntax

The syntax of propositions and sequents in represented in **SML** using *datatypes*. For instance, we can represent the syntax of propositional logic as follows:

```
datatype prop =
   TRUE                 (* ⊤ *)
 | FALSE                (* ⊥ *)
 | ATOM of string       (* A *)
 | CONJ of prop * prop  (* A ∧ B *)
 | DISJ of prop * prop  (* A ∨ B *)
 | IMPL of prop * prop  (* A ⊃ B *)
```

It is often convenient to use infix notation in **SML** for the connectives; but note that you need to declare the fixity of these operators.

```
datatype prop =
   TRUE               (* ⊤ *)
 | FALSE              (* ⊥ *)
```

```
  | ` of string         (* A *)
  | /\ of prop * prop   (* A ∧ B *)
  | \/ of prop * prop   (* A ∨ B *)
  | ~> of prop * prop   (* A ⊃ B *)

infixr 3 ~>
infixr 4 /\ \/
```

Contexts are represented as lists of propositions, and we represent sequents as a context together with a proposition:

```
type context = prop list
datatype sequent = ===> of context * prop (* Γ ⟹ A *)
infixr 0 ===>
```

Example 1.1 (Structural recursion)**.** Using pattern matching in **SML**, we can write a function that processes the syntax of propositions. Here is such a function which counts how deep a proposition is:

```
val rec depth =
  fn TRUE => 0
   | FALSE => 0
   | `_ => 0
   | a /\ b => Int.max (depth a, depth b) + 1
   | a \/ b => Int.max (depth a, depth b) + 1
   | a ~> b => Int.max (depth a, depth b) + 1
```

Note that the above is only a more compact notation for the following equivalent **SML** program:

```
fun depth TRUE = 0
  | depth FALSE = 0
  | depth (`_) = 0
  | depth (a /\ b) = Int.max (depth a, depth b)
  (* ··· *)
```

Exercise 1.1 (Pretty printing)**.** Write a function to convert propositions into strings, adding parentheses in exactly the necessary and sufficient places according to the precedence of the grammar of propositions. Your solution need not account for associativity.

2 Forward Inference Kernel

Usually the trusted kernel of a **LCF**-based proof assistant consists in an implementation of *forward inference*, which is inference from premises to conclusion. We begin with the *signature* for a **LCF** kernel of the intuitionistic sequent calculus; in **SML**, signatures serve as type specifications for entire *structures* (modules).

```
signature KERNEL =
sig
  type proof

  (* What sequent is this a proof of? *)
  val infer : proof -> sequent
```

We represent hypotheses as indices into the context.

```
  type hyp = int
```

Next, we give signatures for the rules of intuitionistic sequent calculus, as functions on the type `proof`; these functions may take additional parameters which are necessary in order to ensure that an actual sequent calculus derivation is uniquely determined by a value of type `proof`.

```
  val init : context * hyp -> proof         (* init *)
  val trueR : context -> proof              (* ⊤R *)
  val falseL : hyp * sequent -> proof       (* ⊥L *)
  val conjR : proof * proof -> proof        (* ∧R *)
  val conjL1 : hyp * proof -> proof         (* ∧L₁ *)
  val conjR2 : hyp * proof -> proof         (* ∧L₂ *)
  val disjR1 : proof * prop -> proof        (* ∨R₁ *)
  val disjR2 : prop * proof -> proof        (* ∨R₂ *)
  val disjL : hyp * proof * proof -> proof  (* ∨L *)
  val implR : proof -> proof                (* ⊃R *)
  val implL : hyp * proof * proof           (* ⊃L *)
end
```

Next, we need to *implement* this signature as structure. To do this, we declare a structure called `Kernel` which **opaquely** ascribes the signature `KERNEL`; opaque ascription, written using `:>` below, ensures that the implementation of the type `proof` remains abstract, i.e. no client of the `Kernel` structure can see its actual concrete representation. This is what ensures that only the kernel needs to be verified and trusted!

```
structure Kernel :> KERNEL =
struct
```

At this point, we have to decide on an internal representation of proofs. We could choose a *transient* representation, in which the proof trace is forgotten:

```
  type proof = sequent
  fun infer s = s
```

Then, the implementations of the rules would simply check that their parameters and premises have the right form, and raise an exception if they do not. For instance:

```
(* ―――――――――― init *)
fun init (ctx, i) =
  ctx ===> List.nth (ctx, i)

(* ―――――――― TR *)
fun trueR ctx =
  ctx ===> TRUE

(* ―――――――――――― FL *)
fun falseL (i, ctx ===> a) =
  if List.nth (ctx, i) = FALSE then
    ctx ===> a
  else
    raise Fail "falseL not applicable"

(*   Γ ⟹ A   Γ ⟹ B
   ―――――――――――― ∧R *)
fun conjR (ctx1 ===> a1, ctx2 ===> a2) =
  if ctx1 = ctx2 then
    ctx1 ===> a1 /\ a2
  else
    raise Fail "conjR not applicable"

(*   Γ, A ∧ B, Δ, A ⟹ C
   ―――――――――――――― ∧L1 *)
fun conjL1 (i, ctx ===> c) =
  case ctx of
     a :: ctx' =>
     (case List.nth (ctx', i) of
         a' /\ b =>
           if a = a' then
             ctx' ===> c
           else
             raise Fail "conjL1 not applicable"
       | _ => raise Fail "conjL1 not applicable")
   | _ => raise Fail "conjL1 not applicable"

(* and so on *)
```

Now, a cleaner and more robust way to write the above rule is the following:

```
(*   Γ, A ∧ B, Δ, A ⟹ C
   ―――――――――――――― ∧L1 *)
fun conjL1 (i, ctx ===> c) =
  let
    val a :: ctx' = ctx
    val a' /\ b = List.nth (ctx', i)
```

```
      val true = a = a'
    in
      ctx' ===> c
    end
    handle _ => raise Fail "conjL1 not applicable"
```

This pattern is also applicable to the other rules of inference. We leave the implementation of the remaining rules as an exercise.

```
end
```

2.1 Evidence-Producing Kernels

The kernel described in the previous section is sufficient for developing and checking sequent calculus proofs. For instance, consider the following sequent calculus derivation:

$$\dfrac{\dfrac{}{A \wedge B, A \Longrightarrow A}\ \mathsf{init}}{A \wedge B \Longrightarrow A}\ \wedge\mathsf{L}_1$$

This is encoded in our kernel as follows:

```
structure K = Kernel
val d : K.proof = K.conjL1 (0, K.init ([`"A", `"A" /\ `"B"], 0))
```

However, the proof object `d` above does not actually contain any information about how the proof was derived; it may be more accurate to call it a "proof certificate" than to call it a "proof". If we wish to be able to inspect the proof derivation after it has been constructed, we may provide a different implementation of the `KERNEL` signature where the type `proof` is implemented by some kind of proof tree.

However, if we do this, then we lose abstraction: someone else could easily produce such a proof tree outside of our kernel. How can we cleanly achieve both abstraction and inspectability of proofs? One approach is to use a *view* together with an abstract type.

Let us begin by defining a type parametric in some type variable `'a`, which captures the shape of sequent calculus proof trees, but does not allows the subtrees to be implemented by `'a`.

```
type hyp = int

datatype 'a deriv =
   INIT of hyp
 | TRUE_R
 | FALSE_L of hyp
 | CONJ_R of 'a * 'a
 | CONJ_L1 of hyp * 'a
 | CONJ_L2 of hyp * 'a
 | DISJ_R1 of 'a
 | DISJ_R2 of 'a
```

```
  | DISJ_L of hyp * 'a * 'a
  | IMPL_R of 'a
  | IMPL_L of hyp * 'a * 'a
```

The idea is that the position of subtrees in each derivation rule are replaced with `'a`. Now, we can interleave the abstract proof type with the type of derivations by supplying it for `'a` in the following way:

```
structure EvidenceKernel :>
sig
  include KERNEL
  val unroll : proof -> proof deriv
end =
struct
  datatype proof = BY of sequent * proof deriv
  infix BY

  fun infer (s BY _) = s
  fun unroll (_ BY m) = m

  fun init (ctx, i) =
    ctx ===> List.nth (ctx, i) BY INIT i
    handle _ => raise Fail "init not applicable"

  fun trueR ctx =
    ctx ===> TRUE BY TRUE_R

  fun falseL (i, ctx ===> p) =
    let
      val FALSE = List.nth (ctx, i)
    in
      ctx ===> p BY (FALSE_L i)
    end
    handle _ => raise Fail "falseL not applicable"

  fun conjR (d1 as ctx1 ===> p1 BY _, d2 as ctx2 ===> p2 BY _) =
    let
      val true = ctx1 = ctx2
    in
      ctx1 ===> (p1 /\ p2) BY CONJ_R (d1, d2)
    end
    handle _ => raise Fail "conjR not applicable"

  fun conjL1 (i, d as ctx ===> r BY _) =
    let
      val p :: ctx' = ctx
```

```
      val p' /\ q = List.nth (ctx, i)
      val true = p = p'
    in
      ctx' ===> r BY CONJ_L1 (i, d)
    end
    handle _ => raise Fail "conjL1 not applicable"

  (* and so on *)
end
```

Exercise 2.1. Now construct a **SML** function to pretty print the derivation that corresponds to a value of type `EvidenceKernel.proof`, using `EvidenceKernel.unroll` and structural recursion.

```
fun pretty (d : proof) : string =
  raise Fail "TODO"
```

3 Refinement Proof and Backward Inference

It is often frustrating to construct proofs manually using the primitives that are exposed by a forward inference kernel `K:KERNEL`. Informally, sequent calculus is optimized for upward (backward) inference from premise to conclusion; the kernel seems to force us to perform proofs inside-out. When using the kernel, it is also necessary to pass annoying parameters, such as the context parameter in `K.trueR`.

Separately for any such kernel, we can develop what is called a *refiner*, which is a module that allows us to construct proofs from the bottom up, without needing to pass in any unnecessary parameters. Regarded as a component of a proof system, because the refiner ultimately is a mode of use of the kernel, it does not need to be trusted.

In the context of refinement proof, we will use the word "goal" to mean a sequent. A *refinement rule* is a partial function that assigns to some goal a list of subgoals, together with a *validation*. The input goal correspond to the *conclusion* of a sequent calculus rule, and the subgoals correspond to the premises. A *validation* is a function that takes a list of `proof` objects (proofs of the premises) and constructs a new `proof` object (a proof of the conclusion). Validations are always constructed using the forward inference rules exposed by the kernel.

In **SML**, these concepts are rendered as follows:

```
type goal = sequent
type subgoals = goal list
type validation = K.proof list -> K.proof
type rule = goal -> subgoals * validation
```

A completed refinement proof produces the empty list of subgoals; therefore, its validation can be instantiated with the empty list of proofs, yielding a proof of the main conclusion.

We will implement the refiner as a structure *fibered* over a kernel `K`:

```
signature REFINER =
sig
  structure K : KERNEL
  type goal = sequent
  type subgoals = goal list
  type validation = K.proof list -> K.proof
  type rule = goal -> subgoals * validation

  val init : hyp -> rule
  val trueR : rule
  val falseL : hyp -> rule
  val conjR : rule
  val conjL1 : hyp -> rule
  val conjL2 : hyp -> rule
  val disJR1 : rule
  val disjR2 : rule
  val disjL : rule
  val implR : rule
  val implL : rule
end
```

Such a signature is implemented via a *functor* from any kernel `K`:

```
functor Refiner (K : KERNEL) : REFINER =
struct
  structure K = K
  type goal = sequent
  type subgoals = goal list
  type validation = K.proof list -> K.proof
  type rule = goal -> subgoals * validation
```

Now observe how we implement the backward inference version of `init`. The input to our function is the *conclusion* of the rule, and we check that the side conditions are satisfied; then we return the empty list of subgoals (there were no premises), and for the validation, we call `K.init` from the kernel.

```
  fun init i (ctx ===> a) =
    let
      val true = List.nth (ctx, i) = a
    in
      ([], fn [] => K.init (ctx, i))
    end
    handle _ => raise Fail "init not applicable"
```

The next two rules follow a similar pattern and are not very interesting.

```
  fun trueR (ctx ===> a) =
    let
```

```
      val TRUE = a
    in
      ([], fn [] => K.trueR ctx)
    end
    handle _ => raise Fail "trueR not applicable"

  fun falseL i (ctx ===> a) =
    let
      val FALSE = List.nth (ctx, i)
    in
      ([], fn [] => K.falseL (i, ctx ===> a))
    end
    handle _ => raise Fail "falseL not applicable"
```

The rules for conjunction are a bit more illustrative:

```
  fun conjR (ctx ===> r)  =
    let
      val p /\ q = r
    in
      ([ctx ===> p, ctx ===> q],
       fn [d1, d2] => K.conjR (d1, d2))
    end
    handle _ => raise Fail "conjR not applicable"

  fun conjL1 i (ctx ===> r) =
    let
      val p /\ _ = List.nth (ctx, i)
    in
      ([p :: ctx ===> r],
       fn [d] => K.conjL1 (i, d))
    end
    handle _ => raise Fail "conjL1 not applicable"
```

We leave the remainder of the refinement rules as an exercise.

```
end

structure R = Refiner (EvidenceKernel)
```

4 Tactics: combinators for refinement rules

It is hard to see how to use refinement rules to construct proofs on their own. However, there are a number of well-known combinators for refinement rules which correspond to *derived rules*; in the **LCF** tradition, derived rules are called *tactics*, and the combinators from which thye are built are called *tacticals*.

```
signature TACTIC =
sig
  structure R : REFINER
  type tactic = R.rule

  val thenl : tactic * tactic list -> tactic
  val then_ : tactic * tactic -> tactic
  val orelse_ : tactic * tactic -> tactic
end
```

The most fundamental tactical is `thenl`; the tactic `thenl (t, ts)` uses the tactic `t` to decompose the current goal into n subgoals; then, the list of tactics `ts` (also of length n) is applied *pointwise* to decompose these subgoals. Then, the resulting lists of subgoals are all combined into a single list. The tactical `then_` is similar, except it uses its second argument to decompose *all* of the remaining subgoals.

Finally, the tactic `orelse_ (t1, t2)` tries to decompose the current goal with `t1`; if this fails, it continues with `t2`.

4.1 Implementing tacticals

The implementation of the standard tacticals above is provided below as a reference; it relies on some tricky list manipulation, but the good news is you only need to implement it once.

```
functor Tactic (R : REFINER) : TACTIC =
struct
  structure R = R
  open R

  type tactic = goal -> subgoals * validation

  fun split (_, []) = ([], NONE, [])
    | split (0, x :: xs) = ([], SOME x, xs)
    | split (n, x :: xs) =
      let
        val (hd, y, tl) = split (n - 1, xs)
      in
        (x :: hd, y, tl)
      end

  fun gobbleWith ([], []) args = []
    | gobbleWith (n :: ns, f :: fs) args =
      let
        val (xs, _, args') = split (n, args)
      in
        f xs :: gobbleWith (ns, fs) args'
```

```
      end

  fun stitchProof (validation, subgoalss, validations) =
    (List.concat subgoalss,
      validation o
        gobbleWith (map length subgoalss, validations))


  fun then_ (t1, t2) goal =
    let
      val (subgoals, validation) = t1 goal
      val (subgoalss, validations) =
        ListPair.unzip (List.map t2 subgoals)
    in
      stitchProof (validation, subgoalss, validations)
    end

  fun thenl (t, ts) goal =
    let
      val (subgoals, validation) = t goal
      val (subgoalss, validations) =
        ListPair.unzip
          (ListPair.mapEq
           (fn (t, g) => t g)
           (ts, subgoals))
    in
      stitchProof (validation, subgoalss, validations)
    end

  fun orelse_ (t1, t2) (goal : goal) =
    t1 goal handle _ => t2 goal
end

structure T = Tactic (R)
open T infix then_ thenl orelse_
```

Now, we can use tactics to capture the backward-inference version of the following proof

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B, A, B \Longrightarrow B}\ \mathsf{init} \quad \dfrac{}{A \wedge B, A, B \Longrightarrow A}\ \mathsf{init}}{A \wedge B, A, B \Longrightarrow B \wedge A}\ \wedge\mathsf{R}}{A \wedge B, A \Longrightarrow B \wedge A}\ \wedge\mathsf{L}_2}{A \wedge B \Longrightarrow B \wedge A}\ \wedge\mathsf{L}_1$$

```
val t : tactic =
  R.conjL1 0 then_
  R.conjL2 1 then_
```

```
  R.conjR thenl
    [R.init 0,
     R.init 1]

val result = t ([`"A" /\ `"B"] ===> `"B" /\ `"A")
val d : proof = #2 result []
```

4.2 Possible extensions

LCF-style tactics do not satisfy every need; many different extensions are possible.

Backtracking The `orelse_` tactical enables a form of proof search procedure, but it cannot be used to implement backtracking. There are at least two ways to extend **LCF** with support for backtracking; one way would be using continuations, but the most common way is to replace the type of tactics with something like the following:

```
type tactic = R.goal -> (R.subgoals * R.validation) stream
```

Then, a backtracking tactical `par : tactic * tactic -> tactic` will simply merge the results fof applying *both* tactics into a single stream:

```
fun par (t1, t2) goal =
  Stream.merge (t1 goal, t2 goal)
```

Existential variables and unification It is difficult to capture a usable refinement proof theory for logics with extistential quantifiers using pure **LCF**; rather than having a single introduction rule for the existential quantifier, it is necessary to have a countable family of such introduction rules, parameterized in the actual witness of the existential. This is highly disruptive to the refinement proof process, since it may be that one only determines how to instantiate the existential $\exists x.A(x)$ by attempting to prove the predicate A.

To resolve this contradiction, most modern proof assistants add a notion of existential variable, which allows one to decompose the goal $\exists x.A(x)$ into $A(?x)$; then, later on in this subproof, the variable $?x$ can be instantiated by *unification* with something concrete (like $?x := 42$).

Existential variables introduce many complexities into the design of **LCF**-style proof assistants, because of the difficulty (and in some cases, impossibility) of finding most-general unifiers. In **Coq** [9], a higher-order unification algorithm is used which produces unifiers which are not most general [10], but because **Coq** adheres to the **LCF** achitecture, this only affects the ergonomics of the tactic system as opposed to the core logic.

On the other hand, higher-order unification is built into the trusted kernel of **Isabelle**, which uses both *dynamic pattern unification* (which produces most general unifiers at the cost of being somewhat restrictive) and general higher-order unification, which may produce infinitely many unifiers (or none).

One benefit of building unification into the core is that it is possible to simplify the notion of proof refinement significantly, capturing both forward and backward

inference in a single kernel [5]; in turn, this obviates the notion of *validation*, which is the hardest part of **LCF**-style tactic systems to implement correctly.

Dependent refinement In the context of implementing dependent type theory, it is useful to consider a notion of refinement rule in which the *statement* of one premise may refer to the *proof* of a previous premise. This is best considered a separate issue from the matter of existential variables, but concrete implementations of this behavior may either use existential variables (as in [6, 1]) or not (as in [7, 8]).

References

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 64–69, 2011.

[2] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1979.

[3] Mike Gordon. From LCF to HOL: A short history. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 169–185. MIT Press, Cambridge, MA, USA, 2000.

[4] Lawrence C. Paulson. *Logic and computation : interactive proof with Cambridge LCF*. Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge, New York, Port Chester, 1987. Autre tirage : 1990 (br.).

[5] Lawrence C. Paulson. Strategic principles in the design of isabelle. In *In CADE-15 Workshop on Strategies in Automated Deduction*, pages 11–17, 1998.

[6] Arnaud Spiwack. *Verified Computing in Homological Algebra, A Journey Exploring the Power and Limits of Dependent Type Theory*. PhD thesis, École Polytechnique, 2011.

[7] Jonathan Sterling and Robert Harper. Algebraic foundations of proof refinement. `https://arxiv.org/abs/1703.05215`, 2017.

[8] Jonathan Sterling, Kuen-Bang Hou (Favonia), Daniel Gratzer, David Christiansen, Darin Morrison, Eugene Akentyev, and James Wilcox. RedPRL – the People's Refinement Logic. `http://www.redprl.org/`, 2016.

[9] The Coq Development Team. The Coq Proof Assistant Reference Manual, 2016.

[10] Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *20th ACM SIGPLAN International Conference on Functional Programming*, Vancouver, Canada, September 2015.