Isar/HOL Tutorial

Steve Nuchia

July 26, 2001

Contents

1 Introduction

This is a retracement in Isar of the examples, with solutions to the exercises, in the Isabelle/HOL tutorial.

It is a journal of my experience trying to learn the system, beginning from when I gave up on prving my solution to exercise 3.4.4 and went back to the beginning to learn Isar. I hope to make a pedagogically useful tutorial introdution to both the tutorial and to Isar out of all this when I am done.

I will update the online copy of this stuff periodically. see http://euler.mcs.utulsa.edu/ nuchiast/isartu

My email address is swn1@alumni.rice.edu and I welcome feedback on this project!

2 ToyList

theory *ToyList* = *PreList*:

We begin with the definitions for the toy list theory verbatim from the tutorial.

datatype *'a list* = *Nil* ([])
| *Cons 'a 'a list* (**infixr** # *65*)

consts *app* :: *'a list* => *'a list* => *'a list* (**infixr** @ *65*)
rev :: *'a list* => *'a list*

primrec
[] @ *ys* = *ys*
app-rdef:
(*x*#*xs*) @ *ys* = *x* # (*xs* @*ys*)

primrec
rev [] = []
rev-rdef:
rev (*x*#*xs*) = (*rev xs*) @ (*x* # [])

Now we need to do an inductive proof. Nowhere that I have found is the basic idiom for induction in Isar explained, yet we need it right off the bat. Here is a boiled-down version. It is a little more verbose than the Isabelle version but the "is" construct really helps.

lemma *app-assoc* [*simp*]: (*xs* @ *ys*) @ *zs* = *xs* @ (*ys* @ *zs*) (**is** *?P xs*)
proof (*induct xs*)
show *?P* [] **by** *simp*
next
fix *x xs* **assume** *?P xs*
show *?P* (*x*#*xs*) **by** *simp*
qed

Here's the deal. The "is" clause tells the system to treat the entire theorem statement as a predicate (truth-valued function) of "xs". This is nothing more than shorthand, but it really helps.

The induct rule splits the current goal, using an induction theorem determined by the type of the variable you name. You clear a goal by showing it is true. Having named the predicate saves me having to type the whole thing in again.

Isar is philosophically committed to the idea that the proof script should be human readable. In Isabelle you can just say "induct_tac xs, simp" and go on to work on the inductive case, but a reader would have to know that the

simplifier had eliminated the base case in order to follow your proof script.

lemma *app-Nil2* [*simp*]: *xs* @ [] = *xs* (**is** *?P xs*)
proof (*induct xs*)
  show *?P* [] **by** *simp*
next
  fix *x xs* **assume** *?P xs*
  show *?P* (*x*#*xs*) **by** *simp*
qed

This next lemma requires something new. Even though "auto" works in the Isabelle script, here we have to guide the proof step by step. This will turn out to be a good thing later, when blind simplification can seriously break a proof. So, we dive immediately into the calculation ("also have") style.

lemma *rev-app* [*simp*]: *rev*(*xs*@*ys*) = (*rev ys*)@(*rev xs*) (**is** *?P xs*)
proof (*induct xs*)
  show *?P* [] **by** *simp*
next
  fix *x xs* **assume** *?P xs*
  show *?P* (*x*#*xs*) (**is** *?LHS* = *?RHS*)
  proof −
    have *?RHS* = (*rev ys*)@(*rev xs*)@(*x*#[]) **by** *simp*
    also from *prems* **have** ... = *?LHS* **by** *simp*
    finally show *?LHS* = *?RHS* **by** *simp*
  qed
qed

A lot of experimentation went into the writing of that little proof. Except for the sugar word "next" and the "is" clauses, nothing can be left out. Play around with it.

Note that the "fix" clause must name x as well as xs. I'm not sure why, but if you drop the x it won't go through. On the other hand, it has no such qualms about ys. Go figure.

Now for the main theorem of this section.

theorem *rev-rev* [*simp*]: *rev*(*rev xs*) = *xs* (**is** *?P xs*)
proof (*induct xs*)
  show *?P* [] **by** *simp*
next
  fix *x xs* **assume** *?P xs*
  show *?P* (*x*#*xs*) **by** *simp*
qed

end

3 Mirrors

theory *Mirrors* = *Main*:

Solution to exercise 2.4.1. I'm not at all sure why the first few proofs in the ToyList theory went through without "from prems" but these both seem to need that clause. Go figure.

datatype *'a tree* = *Tip* | *Node 'a tree 'a 'a tree*

consts *mirror* :: *'a tree* => *'a tree*
primrec
mirror Tip = *Tip*
mirror (*Node l a r*) = *Node* (*mirror r*) *a* (*mirror l*)

lemma *mirror-mirror*: *mirror*(*mirror t*) = *t* (**is** *?mm t*)
proof (*induct t*)
  show *?mm Tip* **by** *simp*
next
  fix *l a r* **assume** *?mm l ?mm r*
  from *prems* **show** *?mm* (*Node l a r*) **by** *simp*
qed

consts *flatten*:: *'a tree* => *'a list*
primrec
flatten Tip = []
flatten (*Node l a r*) = (*flatten l*)@[*a*]@(*flatten r*)

lemma *flatten*(*mirror t*) = *rev*(*flatten t*) (**is** *?P t*)
proof (*induct t*)
  show *?P Tip* **by** *simp*
next
  fix *l a r* **assume** *?P l ?P r*
  from *prems* **show** *?P* (*Node l a r*) **by** *simp*
qed

end

4 BoolExp

theory *BoolExp* = *Main*:

Tracing the development beginning in section 2.4.5.

Proving the trivial lemma about cases over lists is actually very tricky in Isar. Let's see why.

Note that I've rewritten the final term of the case expression slightly, just to make it slightly less trivial (at least formally).

lemma (*case xs of* [] => [] | *y#ys* => *y#ys*) = *xs* (**is** *?P xs*)
proof (*cases xs*)
— So far, so good, but if you now ask it to show ?P [] it gives a warning (in the minibuffer, the last line of the window) and the proof won't go through. Here's what you have to say:
  assume *xs*=[] **from** *prems* **show** *?thesis* **by** *simp*
next
  fix *a list* **assume** *xs*=*a#list* **from** *prems* **show** *?thesis* **by** *simp*
qed

This looks like a massive step backwards from the Isabelle style proofs presented in the tutorial, but the tables will be turned when the theorems get deeper. I hope.

Here comes the boolexp stuff.

datatype *boolex* = *Const bool* | *Var nat* | *Neg boolex*
  | *And boolex boolex*

consts *value*:: *boolex* => (*nat* => *bool*) => *bool*
primrec
value (*Const b*) *env* = *b*
value (*Var x*) *env* = *env x*
value (*Neg b*) *env* = (~*value b env*)
value (*And b c*) *env* = (*value b env* & *value c env*)

datatype *ifex* = *CIF bool* | *VIF nat* | *IF ifex ifex ifex*
consts *valif*:: *ifex* => (*nat* => *bool*) => *bool*
primrec
valif (*CIF b*) *env* = *b*
valif (*VIF x*) *env* = *env x*
valif (*IF b t e*) *env* = (*if valif b env then valif t env else valif e env*)

consts *bool2if*:: *boolex* => *ifex*
primrec
bool2if (*Const b*) = *CIF b*
bool2if (*Var x*) = *VIF x*
bool2if (*Neg b*) = *IF* (*bool2if b*) (*CIF False*) (*CIF True*)
bool2if (*And b c*) = *IF* (*bool2if b*) (*bool2if c*) (*CIF False*)

lemma *valif* (*bool2if b*) *env* = *value b env* (**is** *?P b*)
proof (*induct b*)
  fix *v* **show** *?P* (*Const v*) **by** *simp*
next
  fix *x* **show** *?P* (*Var x*) **by** *simp*
next

```
  fix b assume ?P b from prems show ?P (Neg b) by simp
next
  fix b c assume ?P b ?P c
  from prems show ?P (And b c) by simp
qed

consts normif:: ifex => ifex => ifex => ifex
       norm::   ifex => ifex
       normal:: ifex => bool

primrec
normif (CIF b)   t e  = IF (CIF b) t e
normif (VIF x)   t e  = IF (VIF x) t e
normif-rdef:
normif (IF b t e) u f = normif b (normif t u f) (normif e u f)

primrec
norm (CIF b) = CIF b
norm (VIF x) = VIF x
norm (IF b t e) = normif b (norm t) (norm e)

lemma [simp]: !t e. valif (normif b t e) env =
                        valif (IF b t e) env (is ?P b)
proof (induct b)
   fix v show ?P (CIF v) by simp
next
   fix x show ?P (VIF x) by simp
next
   fix x y z assume ?P x ?P y ?P z
   from prems show ?P (IF x y z) by simp
qed
```

That looks simple, but it took a real struggle to get it right. If you mess up anything in the last stanza it fails, leading you into a tangled and very messy subproof. In particular, using a symbol for IF x y z like we did for a#list in the trivial list cases lemma fails. Why?????

```
theorem valif (norm b) env = valif b env (is ?P b)
proof (induct b)
  fix v show ?P (CIF v) by simp
next
  fix x show ?P (VIF x) by simp
next
  fix b t e assume ?P b ?P t ?P e
  from prems show ?P (IF b t e) by simp
```

qed

primrec
normal (*CIF b*) = *True*
normal (*VIF x*) = *True*
normal (*IF b t e*) = (*normal t* & *normal e* &
  (*case b of CIF b* => *True* | *VIF x* => *True* | *IF x y z* => *False*))

lemma [*simp*]: !*t e*. *normal*(*normif b t e*) = (*normal t* & *normal e*) (**is** *?P b*)
proof (*induct b*)
  fix *v* **show** *?P* (*CIF v*) **by** *simp*
next
  fix *x* **show** *?P* (*VIF x*) **by** *simp*
next
  fix *x y z* **assume** *?P x ?P y ?P z*
  from *prems* **show** *?P* (*IF x y z*) **by** *simp*
qed

theorem *normal* (*norm b*) (**is** *?P b*)
proof (*induct b*)
  fix *v* **show** *?P* (*CIF v*) **by** *simp*
next
  fix *x* **show** *?P* (*VIF x*) **by** *simp*
next
  fix *x y z* **assume** *?P x ?P y ?P z*
  from *prems* **show** *?P* (*IF x y z*) **by** *simp*
qed

Now that I'm getting the hang of it this isn't so bad. The time I spent with Isabelle proof scripts is getting in the way: you have to think very formally about quantification in Isar, and with the ProofGeneral front end you can't always see why an obvious statement isn't working out. Watch the minibuffer for the warning that what you are trying to prove does not satisfy a goal. The warning happens when you say "show", but you don't get an error until later.

Something similar happens in the development of a calculation sequence: the "also" command applies transitivity rules to what you "have" so far. If those rules can't deduce any new facts, the "also" fails! When that happens, say "moreover" instead; that will just append your new fact without trying to reason about it.

On to the exercise, strengthening the definition of normal. The modified predicate test function simply changes the result for CIF in the case.

consts *normyl*:: *ifex* => *bool*

primrec
normyl (*CIF b*) = *True*
normyl (*VIF x*) = *True*
normyl (*IF b t e*) = (*normyl t* & *normyl e* &
(*case b of CIF b => False* | *VIF x => True* | *IF x y z => False*))

As a preliminary step, let's prove that normyl is stronger than normal. For that, we need a lemma to help pierce the case structure embedded in the definition.

lemma *normx*: !*y z*. *normyl* (*IF x y z*) −−> (*normyl x* & *normyl y* & *normyl z*)
(**is** *?P x*)
proof (*cases x*)
fix *v* **assume** *x* = *CIF v* **from** *prems* **show** *?P x* **by** *simp*
next
fix *n* **assume** *x* = *VIF n* **from** *prems* **show** *?P x* **by** *simp*
next
fix *b t e* **assume** *x* = *IF b t e*
from *prems* **show** *?P x* **by** *simp*
qed

I've figured something out. The induct and cases methods build very different goals! Isabelle has a lot of metatheorems that will restructure the goals willy-nilly so you don't notice. In Isar, it matters. Set up a simple goal (doesn't have to be true) and look at the first subgoal generated by each method. Cases leaves your predicate intact but inserts a symbol binding into the premises, while induct rewrites the predicate and leaves the premesis alone. Wild.

This is important since proving an implication should be as easy as "assume premis, show conclusion" but depending on how the predicate has been decorated it may not like it. I am not happy with the higher level unification that is (not) happening in Isar.

A direct consequence of the cases/induct issue is that if you begin a proof using cases and decide to change to induction you probably will have to modify your proof of the base case, even though it is logically identical to the first cases case. Harumph.

I am not going to try to prove strictly stronger, since the management of an existence proof is beyond what I want to tackle right now. Onward, then:

theorem *normyl x* −−> *normal x* (**is** *?P x*)
proof (*induct x*)
fix *v* **show** *?P* (*CIF v*) **by** *simp*
next
fix *n* **show** *?P* (*VIF n*) **by** *simp*
next

```
  fix p q r assume ?P p ?P q ?P r
  show ?P (IF p q r) (is ?Q p)
  proof (cases p)
    fix v assume p = CIF v from prems show ?Q p by simp
  next
    fix n assume p = VIF n from prems show ?Q p by simp
  next
    fix a b c assume p = IF a b c from prems show ?Q p by simp
  qed
qed
```

Well, we didn't need the lemma after all. This last proof nicely illustrates the assymetry between cases and induct.

As an aside, everything from "proof (cases p)" down can be replaced with "by (cases p, simp_all add: prems) qed". That is starting to look a lot more like an Isabelle proof.

Now to normylize an expression.

```
consts normyf:: ifex => ifex => ifex => ifex
       normy:: ifex => ifex
```

The key to the exercise, of course, is to simulate the effect of IF on constants:

```
primrec
normyf (CIF b)   t e  = (if b then t else e)
normyf (VIF x)   t e  = IF (VIF x) t e
normyf-rdef:
normyf (IF b t e) u f = normyf b (normyf t u f) (normyf e u f)


primrec
normy (CIF v) = CIF v
normy (VIF x) = VIF x
normy (IF b t e) = normyf b (normy t) (normy e)
```

A note about incremental proof development: the proofs shown work, you aren't seeing the many, many proofs that don't work.

If you try to say "from prems show foo by simp," say, and it doesn't work, you have to replace the "by simp" clause with a proof block. To get it to accept the keyword proof, you also must remove the "from prems" clause!!!

```
lemma [simp]: !t e. valif (normyf b t e) env =
                        valif (IF b t e) env (is ?P b)
proof (induct b)
  fix v show ?P (CIF v) by simp
next
  fix x show ?P (VIF x) by simp
next
```

fix *x y z* **assume** *?P x ?P y ?P z*
from *prems* **show** *?P* (*IF x y z*) **by** *simp*
qed

theorem *valif* (*normy x*) *env* = *valif x env* (**is** *?P x*)
proof (*induct x*)
fix *v* **show** *?P* (*CIF v*) **by** *simp*
next
fix *n* **show** *?P* (*VIF n*) **by** *simp*
next
fix *b t e* **assume** *?P b ?P t ?P e*
show *?P* (*IF b t e*) (**is** *?Q b*)
— None of the single-step automatic methods seem to work here. Applying simp experimentally shows that it is splitting the equality relation on truth-valued expressions into a pair of implications. That's overkill, since we want a purely formal (i.e. syntactic) proof. Apparently we need to guide it:
proof (*cases b*)
fix *v* **assume** *b* = *CIF v* **from** *prems* **show** *?Q b* **by** *simp*
next
fix *n* **assume** *b* = *VIF n* **from** *prems* **show** *?Q b* **by** *simp*
next
fix *x y z* **assume** *b* = *IF x y z*
from *prems* **show** *?Q b* **by** *simp*
qed
qed

end

5 BasicTypes

theory *BasicTypes* = *Main*:

Beginning section 2.5.

consts *sum*:: *nat* => *nat*
primrec
sum 0 = *0*
sum (*Suc n*) = *Suc n* + *sum n*

lemma *sum n* + *sum n* = *n*∗(*Suc n*) **by** (*induct n*, *auto*)

Hey, that wasn't so bad!
Hmm, here comes one written as a rule.

lemma [| ~ *m*<*n* ; *m* < *n*+*1* |] ==> *m* = *n* **by** *auto*

lemma *min i* (*max j* (*k*∗*k*)) = *max* (*min* (*k*∗*k*) *i*) (*min i* (*j*::*nat*)) **by** *arith*

Wow, arith had to think about that for a while. I would too: I'm not even sure what it means.

lemma *n∗n = n −−> n=0 | n=1* (**is** *?P n −−> ?Q n* **is** *?R n*)
proof (*induct n*)
  show *?R 0* **by** *simp*
  fix *n* **assume** *?R n* **show** *?R* (*Suc n*) **by** (*cases n*, *auto*)
qed

This seems to answer the question of how you handle proving an implication by showing that the premis is false in some cases and the conclusion true in others. Get it down to the cases and let auto deal with the inference rules.

end

6 Simplification

theory *Simplification = Main*:

Beginning chapter 3.
Simplification in Isar is a different breed of cat from that in Isabelle. In Isabelle you can try to simplify a goal, see what redults, and then either work on the result or retract the simplification.
In Isar, a (sub) goal must be stated more or less explicitly and then simplification either proves it or it doesn't. If it doesn't, you have a problem.

lemma [| *xs @ zs = ys @ xs*; [] *@ xs* = [] *@* [] |] ==> *ys = zs*
  by *simp*

lemma !*x*. *f x = g* (*f* (*g x*)) ==> *f* [] = *f* [] *@* [] **by** (*simp* (*no-asm*))

Let's see how to do that one stepwise.

lemma !*x*. *f x = g* (*f* (*g x*)) ==> *f* [] = *f* [] *@* []
proof −
  show *?thesis* **by** *simp*
qed

Just introducing the proof block does not change the goal state in any way. Saying "show ?thesis", however, strips away all the assumptions (everything to the left of the meta-implication) leaving a goal that does not lead the simplifier into a rabbit hole.

This example reveals something important about the semantics of the Isar proof language. If a statement appears in the premises of a goal you can move it to the "prems" theorem register using "assume". You can then strip it out of the local goal state by entering a "show" or "have" context and recover it when needed by mentioning "prems" appropriately.

I hate to say it, but this is starting to make sense.

constdefs *xor*:: *bool* => *bool* => *bool*
  xor A B == (*A* & ~*B*) | (~*A* & *B*)

lemma *xor A* (~*A*) **by** (*simp add*: *xor-def*)

Try opening the proof above by saying "proof (simp only: xor_def)." To finish from there you would have to utter a "show" clause that reiterates the "simplified" goal, a real pain in the tush. Isar really wants you to reason forwards.

On the other hand:

lemma (*let xs* = [] *in xs*@*ys*@*xs*) = *ys*
proof (*simp only*: *Let-def*)
 show [] @ *ys* @ [] = *ys* **by** *simp*
qed

That was pretty painless. Of course it could have been done as a one-liner.

lemma *hd-Cons-tl* [*simp*]: *xs* ~= [] ==> *hd xs* # *tl xs* = *xs*
 by (*cases xs*, *simp*, *simp*)

lemma *xs* ~= [] ==> *hd*(*rev xs*)#*tl*(*rev xs*) = *rev xs* **by** *simp*

Aside: you can get a lot of practice with the subgoal proof stuff by misspelling a couple of words in your theorem statement.

The following lemma is solved in one step by simp, but let's break it down anyway. Without the "intro" method you get a single goal that is a conjunction of implications, a real booger to work with!

lemma !*xs*. *if xs* = [] *then rev xs* = [] *else rev xs* ~= []
proof (*split split-if*, *intro*)
  fix *xs* **assume** *xs* = [] **from** *prems* **show** *xs* = [] **by** *simp*
next — Now this is interesting! It wants us to show False, which is how it says we need to show that the premises are mutually contradictory. Hmmmmm... Getting the right combination of incantations for this one is tricky!
  fix *xs* **assume** *xs* ~= [] *rev xs* = []

from *prems* **show** *False* **by** *simp*
qed

To keep the unification happy you have to assume all the premises you need verbatim, then hack around on them in the subgoal. No shortcuts allowed by assuming something "obviously" equivalent but in a more convenient form!

I have a feeling I'm missing something really important, and it probably has something to do with the "rule" method.

I'm skipping the little section on arithmetic. ProofGeneral users can turn on tracing (as well as most of the other uses of the "ML" directive) from the menu. Try the example with simplification tracing (in "Isabelle/Isar→Settings") turned on:

lemma *rev* [*a*] = [] **proof** *simp* **oops**

After the simp step you would expect to see a trace. Where is it? You have to go to the "Buffers" menu and pick the "isabelle/isar" buffer to see it. The "oops" gets us out of dodge so that the theory document can continue.

Be sure to turn tracing back off, it really slows things down!

end

7 Induction Heuristics

theory *Induction* = *Main* + *Mirrors*:

Beginning section 3.2. I'm including Mirrors so I don't have to retype the background definitions for the exercise.

consts *itrev* :: *'a list* => *'a list* => *'a list*
primrec
itrev [] *ys* = *ys*
itrev (*x*#*xs*) *ys* = *itrev xs* (*x*#*ys*)

Let's do this proof by proving the stronger inductive result as an embedded lemma:

lemma *itrev xs* [] = *rev xs*
proof −
have !*ys*. *itrev xs ys* = (*rev xs*) @ *ys* **by** (*induct xs*, *auto*)

thus *?thesis* **by** *simp*
qed

The only really tricky part about that was to remember to put in the quatifier on ys. Note that Isabelle required it as well.

Exercise 3.2.1

consts *flatten2*:: *'a tree => 'a list => 'a list*

Since the list constructor works right-to-left we need to do a right-to-left in-order traversal.

primrec
flatten2 Tip acc = acc
flatten2 (Node l a r) acc = flatten2 l (a # (flatten2 r acc))

Now it gets interesting. This is where I started to get irritated with Isabelle proof scripts, though I see now that it isn't really all that bad if you ask the question in the right way:

theorem !*acc. flatten2 t acc = (flatten t)@acc*
apply (*induct-tac t*, *auto*)
done

So let's do a pretty Isar proof:

theorem *flatten2 t [] = flatten t*
proof −
  have !*acc. flatten2 t acc = (flatten t)@acc* **by** (*induct t*, *auto*)
  thus *?thesis* **by** *simp*
qed

end

8 Expressions

theory *Expressions = Main*:

Beginning section 3.3

types *'v binop = 'v => 'v => 'v*

datatype *('a,'v)expr = Cex 'v*
| Vex 'a

```
                | Bex 'v binop ('a,'v)expr ('a,'v)expr

consts value:: ('a,'v)expr => ('a => 'v) => 'v
primrec
value (Cex v) env = v
value (Vex a) env = env a
value (Bex f e1 e2) env = f (value e1 env) (value e2 env)

datatype ('a,'v)instr = Const 'v
                      | Load 'a
                      | Apply 'v binop

consts exec:: ('a,'v)instr list => ('a => 'v) => 'v list => 'v list
```

In other words, currying exec with a program (list of instructions) and an environment gives you a stack transformation function. I've taken the liberty of renaming some of the variables in the definition.

```
primrec
exec [] env stk = stk
exec (i#is) env stk = (case i of
    Const v => exec is env (v#stk)
  | Load a  => exec is env ((env a)#stk)
  | Apply f => exec is env ( (f (hd stk) (hd(tl stk))) # (tl(tl stk)) ))

consts comp:: ('a,'v)expr => ('a,'v)instr list
primrec
comp (Cex v) = [Const v]
comp (Vex a) = [Load a]
comp (Bex f e1 e2) = (comp e2)@(comp e1)@[Apply f]



lemma sequence[simp]: !ys stk. exec (xs@ys) env stk =
                      exec ys env (exec xs env stk) (is ?P xs)
proof (induct xs)
  show ?P [] by simp
next
  fix x xs assume ?P xs from prems show ?P (x#xs)
       by (simp split: instr.split)
qed


theorem exec (comp e) env [] = [value e env]
```

proof −
have !*stk*. *exec* (*comp e*) *env stk* = (*value e env*)#*stk* (**is** *?P e*)
proof (*induct e*)
fix *v* **show** *?P* (*Cex v*) **by** *simp*
next
fix *a* **show** *?P* (*Vex a*) **by** *simp*
next
fix *f e1 e2* **assume** *?P e1 ?P e2*
from *prems* **show** *?P* (*Bex f e1 e2*) **by** *simp*
qed
thus *?thesis* **..**
qed

Trying to write the simplification lemma inline reveals a real limitation to that approach. It lets you express and prove the lemma, name it, and use it. But it does not universalize the free variable the way a top-level lemma declaration does, so you can only use it once! If you try to universally quantify xs in the lemma statement you can't prove it. Interesting.

end

9 Recursion

theory *Recursion* = *Main*:

Beginning with section 3.4.1

datatype *'a aexp* = *IF 'a bexp 'a aexp 'a aexp*
| *Sum 'a aexp 'a aexp*
| *Diff 'a aexp 'a aexp*
| *Var 'a*
| *Num nat*
and *'a bexp* = *Less 'a aexp 'a aexp*
| *And 'a bexp 'a bexp*
| *Neg 'a bexp*

consts *evala*:: *'a aexp* => (*'a* => *nat*) => *nat*
evalb:: *'a bexp* => (*'a* => *nat*) => *bool*

primrec
evala (*IF b a1 a2*) *env* =
(*if evalb b env then evala a1 env else evala a2 env*)
evala (*Sum a1 a2*) *env* = *evala a1 env* + *evala a2 env*
evala (*Diff a1 a2*) *env* = *evala a1 env* − *evala a2 env*
evala (*Var a*) *env* = *env a*

```
evala (Num n) env = n

evalb (Less a1 a2) env = (evala a1 env < evala a2 env)
evalb (And b1 b2) env  = (evalb b1 env & evalb b2 env)
evalb (Neg b) env      = (~ evalb b env)

consts substa:: ('a => 'b aexp) => 'a aexp => 'b aexp
       substb:: ('a => 'b aexp) => 'a bexp => 'b bexp
```

Perhaps there are too many a's and b's running around here.

```
primrec
substa s (IF b a1 a2) =
   IF (substb s b) (substa s a1) (substa s a2)
substa s (Sum a1 a2)  = Sum (substa s a1) (substa s a2)
substa s (Diff a1 a2) = Diff (substa s a1) (substa s a2)
substa s (Var a)      = s a
substa s (Num n)      = Num n

substb s (Less a1 a2) = Less (substa s a1) (substa s a2)
substb s (And b1 b2)  = And (substb s b1) (substb s b2)
substb s (Neg b)      = Neg (substb s b)
```

```
theorem
evala (substa s a) env = evala a (%x. evala (s x) env) &
 evalb (substb s b) env = evalb b (%x. evala (s x) env) (is ?P a b)
   by (induct a and b, auto)
```

A one-line proof for a two-line theorem! Since there is nothing but structural recursion going on in substitution it is nice that theorems about it can be proved by pure structural induction without a lot of rigamarole. The conditional constructs in the operationally defined functions are what make proofs about evaluation hard(er).

Exercise 3.4.1, normalizing binary expressions

```
consts normala:: 'a aexp => bool
       normalb:: 'a bexp => bool
primrec
normala (IF b a1 a2) = (normalb b & normala a1 & normala a2)
normala (Sum a1 a2)  = (normala a1 & normala a2)
normala (Diff a1 a2) = (normala a1 & normala a2)
normala (Var a)      = True
normala (Num n)      = True

normalb (Less a1 a2) = (normala a1 & normala a2)
```

normalb (And b1 b2) = False
normalb (Neg b) = False

Remark – it would be interesting to take this farther by defining "canonical" as having all the IFs confined to the outermost layer of the construct, so that all interior arithmetic expressions are IF-free.

consts *norma*:: *'a aexp => 'a aexp*
normif:: *'a bexp => 'a aexp => 'a aexp => 'a aexp*

primrec
norma (IF b a1 a2) = normif b (norma a1) (norma a2)
norma (Sum a1 a2) = Sum (norma a1) (norma a2)
norma (Diff a1 a2) = Diff (norma a1) (norma a2)
norma (Var a) = Var a
norma (Num n) = Num n

normif (Less a1 a2) t e = IF (Less (norma a1) (norma a2)) t e
normif (And b1 b2) t e = normif b1 (normif b2 t e) e
normif (Neg b) t e = normif b e t

theorem *normala (norma (a::'a aexp)) &*
(!t e. normala (normif (b::'a bexp) t e)
= (normala t & normala e))
by (*induct a* **and** *b*, *auto*)

OK, I think I see what was happening. I'd make a mistake in the theory or the theorem statement, then when a "by" proof attempt failed I'd try to expand it.

Eventually I'd get down to a subgoal that I could see was broken and I'd go fix the theory. Then the expanded proof would go through, but I didn't know how to formulate the one-line incantation and get rid of all the intermediate stuff.

I will keep going, since I have not yet reached the point where I gave up on Isabelle scripts.

Section 3.4.2, Nested Recursion.

datatype *('v,'f)term = Var 'v | App 'f ('v,'f) term list*

consts
subst :: *('a => ('b,'f)term) => ('a,'f)term => ('b,'f)term*
substs:: *('a => ('b,'f)term) => ('a,'f)term list => ('b,'f)term list*

primrec

subst s (Var x) = s x
subst-App:
subst s (App f ts) = App f (substs s ts)

substs s [] = []
substs s (t#ts) = subst s t # substs s ts

lemma *subst Var t = (t ::('v,'f)term) &*
substs Var ts = (ts ::('v,'f)term list)
by (*induct-tac t* **and** *ts*, *auto*)

Hmmm.... induct_tac works but induct does not! Something about the types having incompatible induction rules. Let's see if we can prove it without resorting to the tactic.

Or at least see what it is doing. How do I get this thing to show me the rules it is trying to use?

lemma *subst Var t = (t::('v,'f)term) &*
substs Var ts = (ts::('v,'f)term list)
(**is** *?P t & ?Q ts* **is** *?R t ts*)
proof (*induct-tac t* **and** *ts*)
fix *v* **show** *?P (Var v)* **by** *simp*
next
show *?Q []* **by** *simp*
next
fix *f ts* **assume** *?Q ts*
show *?P (App f ts)* **by** *simp*
next
fix *t ts* **assume** *?P t ?Q ts*
from *prems* **show** *?Q (t#ts)* **by** *simp*
qed

OK, I had to go home, read ahead in the tutorial, and sleep, but I got it done (using *induct-tac*.) Isar is extremely picky about what you try to assume. For example, if you try to assume *?R t ts* in the last step, the show does not go on!.

Note for introductory remarks:

A proposition containing ==> is a claim that the right-hand-side can be proved from the left hand side. A proposition based on −−> on the other hand is a *predicate* and you are claiming that in all cases (in the quantification context) the predicate evaluates to *True*. In Isabelle implications are pretty much automatically elevated to meta-implications but in Isar you should try to say what you mean in the first place.

Second note: some not-so-old proverbs, part one:

Isabelle proofs are unreadable. Isar proofs are unwritable.

You can have anything you want but you can only show a pending goal at Isar's Restaurant.

You can't even show a pending goal if you've assumed something or fixed something that Isar doesn't like or failed to fix something that Isar does like. How do you know it didn't like your assumption? The show gives a warning. Then when you finish proving the claim you get a really cryptic error message and apparent progress stops.

More meta-musings:

You use *rule* to *undo* the effect of a rule. This is because we're fundamentally working backwards.

The fundamental rule for proof by contradiction is called *classical*. The classical reasoner is called *clarify* if you don't want it to simplify. If you do it's called *clarsimp*. So far so good, but if you want to allow goal splitting they are called *safe* and *auto*, respectively. The rules for contrapositives don't rewrite an implication in your goal – they re-arrange the list of hypotheses, swapping one of them for the goal. To turn an implication into the equivalent disjunction or its contrapositive you need another rule, the name of which escapes me at the moment. But it is a safe bet you'd never guess it correctly in less than three tries!

A rule, by the way, is nothing more than a proved proposition involving schematic variables and where the outermost operator is meta-implication. Applying a rule forwards means that if you can match up the left-hand-side of the rule with stuff in the left-hand-side of your goal then you can add the right-hand- side of the rule to the left-hand side of your goal.

Got that? Good. Applying a rule *backwards* means that if the right-hand-side of your goal can be matched with the right-hand-side of the rule, the right-hand-side of your goal is *replaced* by the left-hand-side of the rule. There is some hocus-pocus having to do with matching up pieces on the left too, but the take-home from that is if you don't like what *rule* does, try *frule*.

But wait, there's more! If the rule you're trying to use backwards is one that, if applied frontwards, would introduce a symbol that already appears in your goal you can use *rule*. If on the other hand you want to convert your goal to one that uses a particular symbol, you use the *elimination* rule for that symbol with *erule*.

There, piece of cake. I don't understand why people are intimidated by this stuff.

Fundamentally, the objective in either Isabelle or Isar is to eliminate goals. A goal is eliminated most basically by the *assumption* method, which works only if the right-hand-side appears among the terms on the left-hand-side

verbatim. Since matching a theorem against your goal allows you to insert the RHS into the LHS, *by rule thm-foo* is the next-most-fundamental way of eliminating a goal. All *by* does is run the given method, try to eliminate all pending (sub)goals by *assumption*, and fail if the goals don't go away. Once the list of goals is empty you get to say *qed* in Isar or *done* in Isabelle, except that *by* says that for you.

Now Exercise 3.4.2

The goal is to show that substitustion and a certain kind of composition commute. The first challenge is to cleanly model the composition. Since $g\ a$ is an expression it can't be an argument to f. On the other hand, we don't want the supposed composition to be trivially equivalent to nested substitution.

Let's try this:

```
constdefs subcomp:: ('b => ('v,'f) term)
                    => ('a => ('b,'f) term)
                    => ('a => ('v,'f) term)
          subcomp f g == (%a. subst f (g a))

theorem subst (subcomp f g) (t ::('a,'f)term) =     ( subst f ( subst g t))
      & substs (subcomp f g) (ts::('a,'f)term list)=(substs f (substs g ts))
        (is ?P t & ?Q ts)
by (simp add: subcomp-def, induct-tac t and ts, auto)
```

OK, so that's a three-line Isabelle proof collapsed into a one-line Isar proof in an unsubtle manner. Sue me, once you've fixed *induct* so it works here.

Exercise 3.4.3

As the tutorial suggests, it is very hard to get it to accept a functional definition directly. Let's just do it the easy way:

```
consts trev:: ('v,'f)term => ('v,'f)term
       trevs:: ('v,'f)term list => ('v,'f)term list
primrec
trev (Var v) = Var v
trev (App f ts) = App f (trevs ts)
trevs [] = []
trevs (t#ts) = (trevs ts)@[trev t]

lemma trevs-map: trevs ts = map trev (rev ts)
by (induct ts, auto)

theorem trev (trev t)   = (t::('v,'f)term) &
        trevs (trevs ts) = (ts::('v,'f)term list)
```

(**is** *?P t* & *?Q ts*)
by (*induct-tac t* **and** *ts*, *auto simp*: *trevs-map*)

Is it obvious yet that I've been reading ahead into chapter four?

datatype ('*a*,'*i*)*bigtree* = *Tip* | *Br* '*a* '*i* => ('*a*,'*i*)*bigtree*

consts *map-bt*:: ('*a* => '*b*) => ('*a*,'*i*)*bigtree* => ('*b*,'*i*)*bigtree*
primrec
map-bt f Tip = *Tip*
map-bt-rdef:
map-bt f (*Br a F*) = *Br* (*f a*) (%*i*. *map-bt f* (*F i*))

lemma *map-bt* (*g o f*) *T* = *map-bt g* (*map-bt f T*) (**is** *?P T*)
proof (*induct T*) — At last, a case where *induct* works!
show *?P Tip* **by** *simp*
next
fix *a fun* **show** *?P* (*Br a fun*)
apply (*simp add*: *o-def*)
apply (*intro*) — I want *lambdaI* here, but that's not the right name. Fortunately, *intro* alone does what I want.
apply (*cases fun i*)
oops

Well, that was fun. That theorm is provable in one step. What magic incantation is missing from the more explicit text? The world may never know ...

lemma *map-bt* (*g o f*) *T* = *map-bt g* (*map-bt f T*)
by (*induct T*, *auto*)

end

10 Tries

theory *Tries* = *Main*:

ML *set quick-and-dirty*

An aside on managing a large project:

Once a theory file is acceptable to the interactive prover you need to get it typset. That's where the file ROOT.ML and root.tex come in. But the documentation building process can generate new errors, since the text portions are not checked during interactive processing.

Since the order in which individual theory files is processed is controlled by ROOT.ML while their order in the document is controlled by root.tex, it is possible to compile the (independent) theory files in any order. Put the newest files as high up in the list as possible to minimize latency in the ((modify compile correct)* inspect correct)* cycle.

All about problems 3.4.4-5

[The early parts of this theory file were written before I began working with Isar. I'm not going to rewrite them now.]

I begin by focusing on the maintenance of the associative lists. Here I solve the whole problem by defining a lookup and update function pair that realize a concrete associative set protocol.

With associative lists in hand, implmenting tries becomes an exercise in structural recursion. [Ha! Don't I wish this optimistic forward-looking assertion had been true!]

Define the lookup function, `assoc, and field extractors.`

consts *assoc* :: $('k*'v)list => 'k => 'v\ option$
akey :: $('k*'v) => 'k$
aval :: $('k*'v) => 'v$
primrec
akey $(kk,vv) = kk$

primrec
aval (*kk*,*vv*) = *vv*
primrec
assoc-base:
assoc [] *x* = *None*
assoc-rdef:
assoc (*p*#*ps*) *k* = (*if akey p* = *k then Some* (*aval p*) *else assoc ps k*)

The associative list update function is named `arep, it uses the helper function thisrep.`

consts *arep* :: (′*k*∗′*v*)*list* => ′*k* => ′*v option* => (′*k*∗′*v*)*list*
thisrep :: (′*k*∗′*v*) => ′*k* => ′*v option* => (′*k*∗′*v*)*list*
primrec
arep-base:
arep [] *k vo* = (*case vo of None* => [] | *Some v* => [(*k*,*v*)])
arep-rdef:
arep (*p*#*ps*) *k vo* = (*thisrep p k vo*) @ (*arep ps k vo*)
primrec
thisrep p k None = (*if akey p* = *k then* [] *else* [*p*])
thisrep p k (*Some v*) = (*if akey p* = *k then* [(*k*,*v*)] *else* [*p*])

The main theorem (for associativ lists) is given in two parts, the first establishing the independence of keys and the second establising, in essence, that the result of a lookup is the value of the most recent (outermost) update operation. This is strictly true only if arep and

`are the only constructors allowed. It would be entertaining to go ahead and cast all this as an axiomatic data type, but I need to sleep.`

`As recommended in the tutorial, we begin by expanding the set of default inference.`

declare *Let-def*[*simp*] *option.split*[*split*]

theorem *arep-indep*: *k1* ~= *k2* ==> *assoc* (*arep d k2 vo*) *k1* = *assoc d k1*
apply (*induct-tac d*)
apply (*force*)
apply (*simp only*: *arep-rdef*)
apply (*case-tac vo*)
apply (*auto*)
done

theorem *assoc-rep*[*simp*]: *assoc* (*arep d k2 vo*) *k1* =

(*if k1 = k2 then vo else assoc d k1*)
apply (*auto*)
apply (*induct-tac d*, *auto*)
apply (*case-tac vo*, *auto*)
apply (*rule arep-indep*, *simp*)
done

Note that this theorem cannot be established without either formalizing the idea of a well-formed associative list or having `arep delete extraneous entries. After spending a very interesting couple of days playing with the former, I bailed out and went with the latter.`

`That last proof directive, by the way, is the direct way to get a lemma applied to the current goal.` [1] `Now for the tries; the beginning is verbatim from the tutorial.`

datatype ('*a*,'*v*)*trie* = *Trie* '*v option* ('*a* ∗ ('*a*,'*v*)*trie*)*list*

consts *value* :: ('*a*,'*v*)*trie* => '*v option*
alist :: ('*a*,'*v*)*trie* => ('*a* ∗ ('*a*,'*v*)*trie*)*list*
primrec *value-def*:
value (*Trie ov al*) = *ov*
primrec *alist-def*:
alist (*Trie ov al*) = *al*

consts *lookup* :: ('*a*,'*v*)*trie* => '*a list* => '*v option*
primrec
lookup-base:
lookup t [] = *value t*
lookup-rdef:
lookup t (*a*#*as*) = (*case assoc* (*alist t*) *a of*
None => *None*
| *Some at* => *lookup at as*)

lemma [*simp*]: *lookup* (*Trie None* []) *as* = *None*
apply (*case-tac as*, *auto*)
done

Now I need to use a modified update. The one for problem 4 is just like the original, but uses the `arep operator.`

consts *update* :: ('*a*,'*v*)*trie* => '*a list* => '*v* => ('*a*,'*v*)*trie*

primrec

[1] This interjection was writen relatively early in my journey, I'm keeping it here for journal-keeping integrity reasons.

update t [] v = Trie (Some v) (alist t)
update-rdef:
update t (a#as) v = (let tt = (case assoc (alist t) a of
None => Trie None [] | Some x => x)
in Trie (value t) (arep (alist t) a (Some (update tt as v))))

This update is leak-free, since the underlying update operation is leak-free, and still satisfies the correctness property theorem:

theorem *!t v bs. lookup (update t as v) bs =*
(if as = bs then Some v else lookup t bs)
apply(*induct-tac as, auto*)
apply(*case-tac*[!] *bs, auto*)
done

To build a trie update operation that implements deletion, we procede along the same lines as the general *arep*, representing deletion by *None*. The details, however, are tricky: since intererior nodes can't be deleted just because they carry no data, it may be necessary to do some garbage collection when deleting a leaf node. Fully developing this would involve showing that a deletion in a well-formed trie leaves it well-formed. I'm not going to attack that. Instead, all I show is that the function I define also satisfies the correctness property theorem.

consts *u2* :: *('a,'v)trie => 'a list => 'v option => ('a,'v)trie*
tfix :: *('a,'v)trie option => ('a,'v)trie*
tgc :: *('a,'v)trie => ('a,'v)trie option*

primrec
tfix-none:
tfix None = Trie None []
tfix-some:
tfix (Some t) = t
recdef *tgc* {}
tgc-none:
tgc (Trie None []) = None
tgc-some:
tgc t = Some t

consts *u2core*:: *'v option => ('k*('k,'v)trie)list => 'k list => 'v option*
=> ('k,'v)trie option

defs
u2-def:
u2 t s vo == tfix (u2core (value t) (alist t) s vo)

primrec
u2core-base:
u2core ovo al [] *vo* = *tgc* (*Trie vo al*)
u2core-rdef:
u2core ovo al (*x*#*xs*) *vo* = *tgc* (*Trie ovo*
(*let tt* = *tfix* (*assoc al x*) *in*
arep al x
(*u2core* (*value tt*) (*alist tt*) *xs vo*)))

This update is leak-free, since the underlying list update operation is leak-free, and since (I claim) it trims away non-terminal nodes when they become leaves. I will prove that `u2 also satisfies the correctness property theorem. Attempting the proof in one stage motivates the following lemmas:`

OK, this is where it all fell apart on my first pass. Clean slate time.
I had written and proved the following

```
lemma string_diff1: "s ~= t & s ~= [] & t ~= [] ==>
                     hd s ~= hd t | tl s ~= tl t"
```

but it never helped as much as I thought it would. Now that I know more it seems that the conjunctions were standing in the way. The difference between an "&" and a ";" in a theorem statement does not look very important to the beginner's eye, given how quickly one is exchanged for the other in proving a theorem. But it makes a big difference when you go to use a rule!

lemma *string-diff1*[*intro*]: [| *s* ~= *t* ; *s* ~= [] ; *t* ~= [] |]
==> *hd s* ~= *hd t* | *tl s* ~= *tl t*
by (*cases s*, *simp*, *cases t*, *simp-all*)

Again I've just taken the original 3-line Isabelle proof and repackaged it with "by".
A typo I made when reformatting the lemma led me to discover something worth noting. Markus Wenzel sent me a nice email with lots of little example proofs, one of which turned me on to using the *case* construct rather than assumptions to drive a proof. Unfortunately you have to use the name of the constructor to designate the case, so if you usually use a sugared form (like [], which is really []) you may not know how to write down the name of the case!
Aha! There is a "show me cases" thingy under the Isabell/Isar help menu! It only works when you are inside a cases construct, but that's still a good thing.
Let's expand this next one to illustrate the syntax:

theorem *case-some*:(*case a of None => None | Some v => f v*) ~= *None*
==> *a* ~= *None* (**is** *?P a ==> ?Q a*)
proof (*cases a*)
case *None* — this just adds the assumption *a = None*
assume *?P a* **from** *prems* **show** *?Q a* **by** *simp*
next
case *Some* — This is more intersting: it introduces a fixed variable for a to be Some of.
assume *?P a* **from** *prems* **show** *?Q a* **by** *simp*
qed

theorem *case-none*:*a = Some v &*
(*case a of None => None | Some x => f x*) = *None*
==> *f v = None*
by (*case-tac a*, *auto*)

theorem *case-one*:(*case a of None => None | Some x => f x*) = *Some v*
==> *? x. a = Some x & f x = Some v*
by (*case-tac a*, *auto*)

Here are some lemmas about my trie ¡–¿ trie option conversion helper functions.

lemma *tfix-id*[*simp*]: *tfix* (*tgc t*) = *t*
apply (*case-tac t*)
apply (*case-tac option*, *case-tac list*)
apply (*auto*)
done

lemma *tgc-id*: *tfix* (*tgc t*) = *tfix* (*Some t*)
by (*auto*)

lemma *lookup0* [*simp*]: *lookup* (*u2 t* [] *vo*) *x* =
(*if x* = [] *then vo else lookup t x*)
apply (*case-tac x*)
apply (*simp-all add*: *u2-def u2core-base*)
done

To make the main proofs tractible, it turns out we need to be able to talk about the recursive structure of the trie and not just about values of lookup.

constdefs *subtrie*:: ('*k*,'*v*)*trie* => '*k* => ('*k*,'*v*)*trie option*
subtrie t k == *assoc* (*alist t*) *k*

lemma *lookup1*: *lookup t* (*a*#*s*) = *lookup* (*tfix* (*subtrie t a*)) *s*
apply (*simp only*: *subtrie-def lookup-rdef*)

apply (*case-tac subtrie t a*)
apply (*auto*)
done

lemma *lookup2*: *a ~= b --> subtrie (u2 t (a#s) vo) b = subtrie t b*
apply (*simp only*: *u2-def u2core-rdef Let-def subtrie-def*)
apply (*auto*)
done

lemma *subtrie-dist*: *tfix (subtrie (u2 t (a#s) vo) a)*
= u2 (tfix (subtrie t a)) s vo
apply (*simp only*: *u2-def u2core-rdef*)
apply (*simp*)
apply (*simp only*: *subtrie-def alist-def assoc-rep*)
apply (*simp*)
done

All of that could have been done with one *simp add*: but breaking it up makes it possible to walk through it interactively.

At this point I became frustrates trying to prove the main theorem and started backing off to more and more abstract theorems about predicates defined on pairs of strings that may or may not be equal. At every stage my proof ends up failing because the recursion hypothesis has the wrong structure. The classical proof for these situations is to induct over the length of the shorter key, which involves a case distinction. But I can't get that proof to work, since the automatically generated goal does not simulate the stripping of the lead character on the longer string!

What seems to be needed is a specialized deduction rule for induction on a pair of unequal strings. Let's try to set one up.

constdefs *pair-reductive*:: *('a list => 'a list => 'b) => bool*
pair-reductive f ==
(!a s t. f (a#s) (a#t) = f s t)
pair-symmetric:: *('a list => 'a list => 'b) => bool*
pair-symmetric f == (!s t. f s t = f t s)
pair-inductive:: *('a list => 'a list => bool) => bool*
pair-inductive f ==
(!s t ss tt. f ss [] & f [] tt &
(f ss tt --> f (s#ss) (t#tt)))

Here I'm using some more of the tricks I got in my email. Cool!

lemma *ind-sym*: *[|pair-reductive f ;*
pair-inductive f |]
==> pair-symmetric f

proof −
  assume *red*: *pair-reductive f*
  assume *ind*: *pair-inductive f*
  show *?thesis*
  proof (*simp only*: *pair-symmetric-def*, *rule allI*, *rule allI*)
    fix *s t* **show** *f s t* = *f t s*
    proof (*cases s*)
      case *Nil* **from** *ind* **show** *f s t* = *f t s*
oops

Well Shoot. Maybe I don't need that theorem. It turns out to be true, since *pair-inductive* forces f to be a predicate which is always true. Let's just prove that, if we can.

theorem *pair-induction*: *pair-inductive f* ==> *f x y*
proof −
  assume *ind*: *pair-inductive f*
  from *ind* **have** *lshort*: !!*t*. *f* [] *t*
    by (*simp add*: *ind pair-inductive-def*)
  moreover from *ind* **have** *rshort*: !!*s*. *f s* []
    by (*simp add*: *ind pair-inductive-def*)
  moreover have *samesame*: !!*s*. *f s s*
apply (*cases* (*x*,*y*))
apply (*cases x*)
sorry
  have *sym*: !!*x y*. *f x y* = *f y x* — I want to prove it this way since otherwise I end up having to prove it twice. Eliminating (using "introduction" rules!) the predicate-level universal quantifier on two variables found in the definition of *pair-symmetric* yields two different subgoals that are identical down to renaming the meta-universally quantified variables. Foo.
sorry
  from *sym* **show** *?thesis* **sorry**
  — try *by* (*simp add*: *pair-symmetric-def*)
oops

I bailed out of that attempt for the following News Flash:

I had posted a version of the string pair induction problem to the mailing list for help. Jeremy@discus.anu.edu.au sent me a proof which appears to be in some hindbrain dialect not accepted by Isabelle99-2/Isar/HOL:

```
goal List.thy
"( (!a ss tt. P ss tt --> P (a#ss) (a#tt)) &
   (!a b ss tt. a ~= b --> P (a#ss) (b#tt)) &
   (!s. P s []) & (!s. P [] s) ) --> P x y";
```

```
br impI 1;
by (res_inst_tac[("x","y")] spec 1);
by (induct_tac "x" 1);
by (Fast_tac 1);
br allI 1;
by (case_tac "x" 1)
by (Fast_tac 1)
by (case_tac "a=aa");
by (Fast_tac 1);
by (Fast_tac 1);
```

Let's see if we can figure out what it's saying.

theorem *pair-induct*:
((!*a ss tt*. *P ss tt* −−> *P* (*a*#*ss*) (*a*#*tt*)) &
(!*a b ss tt*. *a* ~= *b* −−> *P* (*a*#*ss*) (*b*#*tt*)) &
(!*s*. *P s* []) & (!*s*. *P* [] *s*)) −−> *P x y* (**is** *?H* −−> *P x y*)

apply(*rule impI*)
— The application of "spec" here seems to be both unnecessary and impossible.
apply (*induct-tac x*)
apply (*fast*)
oops

Well, Shoot.

Digging around in the Isabelle reference manual shows at least some information about these constructs, but they are disabled when Isar is enabled. I can't easily switch to non-Isar Isabelle in the middle of a file, and the chapter on converting scripts (the last section of the Isar Reference Manual) is daunting.

Attempting to read the unreadable proof, it seems that it is essentially the induct x, cases y proof that solves the lookup-update theorem for the leaky implementation given in the tutorial. Using spec on a pair to simultaneously fix both x and y is a neat trick, but it doesn't seem to work out in Isar.

Maybe somebody will send me an Isar proof. At least I know that it's a theorem! On the authority of a random piece of email I'll fake a proof of the pair induction principle and use it to end this nightmare.

theorem *pair-induct*:
[| !!*a ss tt*. *P ss tt* ==> *P* (*a*#*ss*) (*a*#*tt*) ;
!!*a b ss tt*. *a* ~= *b* ==> *P* (*a*#*ss*) (*b*#*tt*) ;
!!*s*. *P s* [] ; !!*s*. *P* [] *s* |] ==> *P x y*
sorry

OK, let's see how a fancy rule like that can be used to turn an impossible proof into a cakewalk.

The following is a concrete example of the pair induction problem that I was unable to solve directly. It should go through easily using the new pair induction principle.

consts *str-common*:: (*'a list* * *'a list*) => *'a list*
recdef *str-common measure* (%(*ss*,*tt*). *length ss*)
str-common(*s*#*ss*,*t*#*tt*) = (*if s* = *t then s*#*str-common* (*ss*,*tt*) *else* [])
str-common(*ss*,*tt*) = []

I could have added a case for identical arguments, but it would be cleaner to prove it as a simplification lemma.

lemma [*simp*]: *str-common*([],*s*) = [] **by** *simp*

lemma [*simp*]: *str-common*(*s*,[]) = [] **by** (*cases s*, *auto*)

Fascinating: I had to give it cases on the second lemma. What is the difference???

I had to come back and name the following, since I needed to use them in preference to all other simplification rules in a tight spot farther down.

lemma *comm-cat-right*[*simp*]: *str-common*(*s*,*s*@*t*) = *s*
by (*induct s*, *auto*)
lemma *comm-cat-left* [*simp*]: *str-common*(*s*@*t*,*s*) = *s*
by (*induct s*, *auto*)

consts *str-uncommon*:: (*'a list* * *'a list*) => *'a list*
recdef *str-uncommon measure* (%(*s*,*t*). *length s*)
str-uncommon([],*tt*) = []
str-uncommon(*ss*,[]) = *ss*
str-uncommon(*s*#*ss*,*t*#*tt*) = (*if s* = *t then str-uncommon*(*ss*,*tt*) *else s*#*ss*)

lemma *str-com-rec*: *hd* (*str-common*(*a*#*s*,*a*#*t*)) = *a* &
tl (*str-common*(*a*#*s*,*a*#*t*)) = *str-common*(*s*,*t*)
by *simp*

lemma *str-com-stop*: *s* ~= *t* ==> *str-common*(*s*#*ss*,*t*#*tt*) = []
by *simp*

lemma *str-com-go*: *s* = *t* ==> *str-common*(*s*#*ss*,*t*#*tt*) = *s*#(*str-common*(*ss*,*tt*))
by *simp*

even with the foregoing I could not make progress on the following:

```
lemma str_com_prefix: "(!t. str_common(s,t) = str_common(t,s))
```

```
                    & (!s. str_common(s,t) = str_common(t,s))" (is "?P s t")
proof (induct ?P s and t)
proof (cases "s=t", simp, cases s, simp, cases t, simp)
  fix x xs y ys assume "s=x#xs" "t=y#ys" "?P xs ys"
  from prems show "?P s t"
```

Let's try again, without having to strengthen the real goal:

lemma *str-com-prefix*: *str-common*(*s*,*t*) = *str-common*(*t*,*s*) (**is** *?P s t*)
by (*rule pair-induct*, *auto*)

Hurrah! I'm finally getting past the beginner stage.

lemma *uncommon-prefix*[*simp*]:
    str-common(*s*,*t*)@*str-uncommon*(*s*,*t*) = *s* (**is** *?P s t*)
proof (*rule pair-induct* [*of ?P*])
— Without the of clause it fails miserably. Sigh.
  fix *s* **show** *?P* [] *s* **by** (*cases s*, *auto*)
next
  fix *s* **show** *?P s* [] **by** (*cases s*, *auto*)
next
  fix *a ss tt* **assume** *?P ss tt* **show** *?P* (*a*#*ss*) (*a*#*tt*) **by** *simp*
next
  fix *a b ss tt* **assume** (*a*::′*a*) ~= (*b*::′*a*)
— Apparently = doesn't imply type compatibility! Interestingly, you can't use a new free type variable here. In fact, you only really need the type specifier on *b*.
  from *prems* **show** *?P* (*a*#*ss*) (*b*#*tt*) **by** *auto*
qed

At this point I would go back and insert the type constraint into the original rule by changing the way the inequality is written, but inspecting the stored theorem shows that the constraint is there. Somehow it is being lost in the shuffle.

The only thing not solved by the induction idiom followed by auto, interestingly, are the two base cases. Adding list.split doesn't help. You'd think adding the definition of *str-uncommon* would help, but it doesn't. There's a comment in the tutorial about priving trivial-looking simplification lemmas for functions defined by recdef. This is where that comes in. Might as well add them now:

lemma [*simp*]: *str-uncommon*(*s*,[]) = *s* **by** (*cases s*, *auto*)
lemma [*simp*]: *str-uncommon*([],*s*) = [] **by** (*cases s*, *auto*)

Note that a string not being empty is the minimum condition for saying anything at all about its head or tail.

New revelation: not not equal is not the same thing as equal. For this reason,

the following lemma is quite a challenge to prove. Negating the thesis buys you nothing!

Fortunately enough, assuming the equality seems to be an acceptable step.

lemma *str-split1*:
  [|
    str-uncommon(*s*,*t*) ~= [] ;
    str-uncommon(*t*,*s*) ~= []
  |] ==>
  hd (*str-uncommon*(*s*,*t*)) ~= *hd* (*str-uncommon*(*t*,*s*))
proof — Default intro rule turns it into a proof by contradiction
  assume *tails*: *str-uncommon*(*s*,*t*) ~= [] **and**
      tailt: *str-uncommon*(*t*,*s*) ~= [] **and**
      sameheads: *hd* (*str-uncommon*(*s*,*t*)) = *hd* (*str-uncommon*(*t*,*s*))
— Note the use of the connective "and" when assuming a batch of individually named propositions.
  def *prefix*: *p* == *str-common*(*s*,*t*)
  def *pivots*: *c* == *hd* (*str-uncommon*(*s*,*t*))
  from *this* **and** *sameheads* **have** *pivott*: *c* = *hd*(*str-uncommon*(*t*,*s*)) **by** *simp*
  def *ttail*: *tt* == *tl* (*str-uncommon*(*t*,*s*))
  def *stail*: *ss* == *tl* (*str-uncommon*(*s*,*t*))
  from *prefix pivots stail tails* **have** *scomp*: *s* = *p*@[*c*]@*ss* **by** *simp*
— Alas, here is one of those places where we must guide the simplifier step by bloody step.
  from *pivott* **have** *p*@[*c*]@*tt* = *p*@([*hd* (*str-uncommon*(*t*,*s*))]@*tt*) **by** *simp*
  also from *ttail tailt* **have** ... = *p*@(*str-uncommon*(*t*,*s*)) **by** *simp*
  also from *prefix* **and** *str-com-prefix*
    have ... = (*str-common*(*t*,*s*))@(*str-uncommon*(*t*,*s*)) **by** *force*
  finally have *tcomp*: *t* = *p*@[*c*]@*tt* **by** *simp*
— Well that finally worked. Now to show a contradiction.
  from *scomp* **have** *s* = (*p*@[*c*])@*ss* **by** *simp*
  from *this* **have** *killer1*: *str-common*(*s*,*p*@[*c*]) = *p*@[*c*]
    by (*simp only*: *comm-cat-left*)
— getting that to go through was a bear. At some earlier stage I had a theorem that *str-common* curried with one string was idempotent. I seem to have deleted it. Sigh.
  from *tcomp* **have** *t* = (*p*@[*c*])@*tt* **by** *simp*
  from *this* **have** *killer2*: *str-common*(*t*,*p*@[*c*]) = *p*@[*c*]
    by (*simp only*: *comm-cat-left*) — OK, now I'm just wandering around in the wilderness. If I'd proved this on paper first I would know exactly what contradiction to show here. Of course on paper I would have probably just written "obvious" ;-)
  have *killer3*: *str-common*(*p*@[*c*]@*ss*,*p*@[*c*]@*tt*) ~= *p*
   by (*induct p*, *auto*)

from *killer3 prefix scomp tcomp* **show** *False* **by** *simp*
qed

Now that was easy, once I figured out how to prove the silly thing. New proverb:
Automatic theorem proving is mighty fine, but don't forget how to develop a proof strategy. You'll need one when the automation fails.
Exercise: figure out how much shorter that proof could have been if I'd had the strategy clearly worked out ahead of time.

Well, I've popped my lemma stack all the way back to the main theorem on my updatable tries!

theorem !*t*. *lookup* (*u2 t as vo*) *bs* =
(*if as* = *bs then vo else lookup t bs*) (**is** *?P as bs*)
proof (*cases as* = *bs*)

oops

Stopping for the evening. Need to do laundry.

```
   case True [simplified]
   show "?P as bs" sorry
   proof (simp only: split: split_if, simp only: prems, induct bs)
      case Nil

apply, induct as)
proof (rule pair_induct [of ?P])
   fix s show "?P [] s" by force
next
   fix s show "?P s []" -- {* Note the operands are not used symmetrically!
     by (cases s, auto simp: u2_def)
next
   fix a b ss tt assume "(a::'a) ~= (b::'a)"
   show "?P (a#ss) (b#tt)"
   apply (auto simp: u2_def lookup_rdef)
apply (simp only: prems lookup_rdef u2_def)
apply clarify


apply (intro)
apply (case_tac "as = bs", split split_if, safe)
-- {* This has isolated the case lookup (update t as v) as = v, which
we attack by induction on the key and case splitting on the value option.
For the case where the result is none we need another recursion.  The
```

```
Some case is inconsistent with a None result from assoc, which is enough
to finish that branch of the recursion.
apply (induct_tac as, simp)
apply (case_tac vo)
prefer 3
apply (split split_if)
apply (simp)
apply (simp only: u2_rdef)
apply (simp only: Let_def)
apply (simp only: lookup_rdef)
apply (clarify)
apply (case_tac t)
apply (force)
apply (simp add: Let_def)
apply (split split_if)
apply (clarify)
back
back
back
back
back
back
apply (induct_tac as)
   -- {* At this point there are three goals, the
first corresponding to as=bs=[], which the simplifier solves.
apply (simp)
-- {* The second is the case as=bs ~= [].
apply (split split_if, safe)
-- {* that mouthful makes if in the conclustion go away,
apply (rule assoc_rep)
apply (induct_tac as, simp)
-- {* The basis case is disposed of by simp.  Now it gets interesting!
simp on the recursive case returns a goal the size of Dallas. The
key is to exploit a lemma about list inequality:
apply (subgoal_tac "")
apply (simp only: lookup_rdef)
apply (force)
apply (insert assoc_rep, simp)

apply (force)
done

end
```