FIT - From's Isabelle Tutorial

Verification of Quicksort

Andreas Halkjær From

DTU Compute
Department of Applied Mathematics and Computer Science

Contents

Section 1

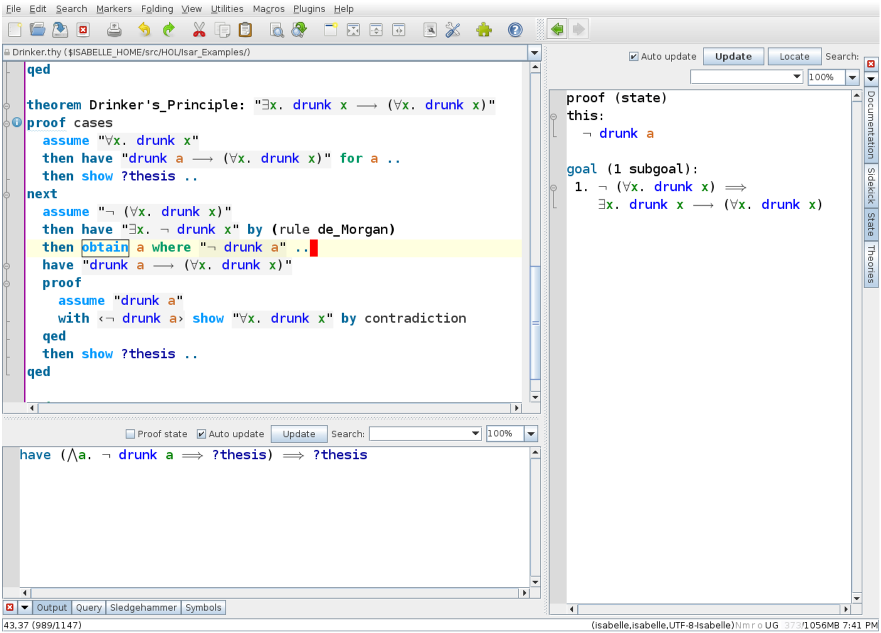
Introduction

Isabelle I

Source: `isabelle.in.tum.de/overview.html`:

> *Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus.*
>
> *The main application is the formalization of mathematical proofs and in particular formal verification, which includes* ***proving the correctness of computer hardware or software*** *and proving properties of computer languages and protocols.*

Isabelle II


File Edit Search Markers Folding View Utilities Macros Plugins Help
Drinker.thy ($ISABELLE_HOME/src/HOL/Isar_Examples/)
qed
theorem Drinker's_Principle: "∃x. drunk x ⟶ (∀x. drunk x)"
proof cases
assume "∀x. drunk x"
then have "drunk a ⟶ (∀x. drunk x)" for a ..
then show ?thesis ..
next
assume "¬ (∀x. drunk x)"
then have "∃x. ¬ drunk x" by (rule de_Morgan)
then obtain a where "¬ drunk a" ..
have "drunk a ⟶ (∀x. drunk x)"
proof
assume "drunk a"
with ‹¬ drunk a› show "∀x. drunk x" by contradiction
qed
then show ?thesis ..
qed
Proof state
Auto update
Update
Search:
100%
have (⋀a. ¬ drunk a ⟹ ?thesis) ⟹ ?thesis
Output
Query
Sledgehammer
Symbols
43,37 (989/1147)
(isabelle,isabelle,UTF-8-Isabelle)Nmro UG 373/1056MB 7:41 PM
Auto update
Update
Locate
Search:
100%
proof (state)
this:
¬ drunk a
goal (1 subgoal):
1. ¬ (∀x. drunk x) ⟹
∃x. drunk x ⟶ (∀x. drunk x)
Documentation
Sidekick
State
Theories

Isabelle III

Isabelle/HOL includes powerful specification tools, e.g. for (co)datatypes, (co)inductive definitions and recursive functions with complex pattern matching.

Proofs are conducted in the structured proof language Isar, allowing for proof text naturally understandable for both humans and computers.

For proofs, Isabelle incorporates some tools to improve the user's productivity. In particular, Isabelle's classical reasoner can perform long chains of reasoning steps to prove formulas. ***The simplifier can reason with and about equations.*** *Linear arithmetic facts are proved automatically, various algebraic decision procedures are provided. External first-order provers can be invoked through sledgehammer.*

Slides

> Verified code has a grey background.
> These snippets are generated directly by Isabelle after verification.

Extra information , esp. proof state, has red names.

Commands are sometimes explained like this.

Cues for myself look like this.

Data types

Algebraic data types like Standard ML.

datatype *mynat* = *Zero* | *Succ mynat*

datatype *'a mylist* = *Nil* | *Cons 'a ‹'a mylist›*

Restrictions on recursive positions: e.g. recursion only allowed to the right of $\Rightarrow$.
No empty types ala:

datatype 'a stream = Cons 'a ‹'a stream›

Need codatatypes.

Primitive recursion

Recursion only allowed on direct arguments of constructor.

primrec *plus* :: ‹*mynat* ⇒ *mynat* ⇒ *mynat*› **where**
  ‹*plus Zero m* = *m*›
| ‹*plus* (*Succ n*) *m* = *Succ* (*plus n m*)›

primrec *len* :: ‹'*a mylist* ⇒ *mynat*› **where**
  ‹*len Nil* = *Zero*›
| ‹*len* (*Cons x xs*) = *Succ* (*len xs*)›

primrec *app* :: ‹'*a mylist* ⇒ '*a mylist* ⇒ '*a mylist*› **where**
  ‹*app Nil ys* = *ys*›
| ‹*app* (*Cons x xs*) *ys* = *Cons x* (*app xs ys*)›

Our first proof I

We can state properties about the programs directly:

theorem *len-app*: ‹$len\ (app\ xs\ ys) = plus\ (len\ xs)\ (len\ ys)$›

xs and *ys* are automatically universally quantified.
Proof by induction:

proof ($induct\ xs$)

Splits the goal into a case for each constructor of *xs*.

Nil $len\ (app\ Nil\ ys) = plus\ (len\ Nil)\ (len\ ys)$

Cons $len\ (app\ (Cons\ x\ xs)\ ys) = plus\ (len\ (Cons\ x\ xs))\ (len\ ys)$

Our first proof II (Nil)

$$\textcolor{red}{?case}\ len\ (app\ Nil\ ys) = plus\ (len\ Nil)\ (len\ ys)$$

Equational reasoning.

```
case Nil
have ‹len (app Nil ys) = len ys›
  by simp
also have ‹... = plus Zero (len ys)›
  by simp
also have ‹... = plus (len Nil) (len ys)›
  by simp
finally show ?case
  by simp
```

have states intermediary facts.
also chains them together.
finally completes the chain.

Our first proof III (Cons)

?case *len (app (Cons x xs) ys) = plus (len (Cons x xs)) (len ys)*

IH *len (app xs ys) = plus (len xs) (len ys)*

```
case (Cons x xs)
have ‹len (app (Cons x xs) ys) = len (Cons x (app xs ys))›
  by simp
also have ‹... = Succ (len (app xs ys))›
  by simp
also have ‹... = Succ (plus (len xs) (len ys))›
  using Cons by simp
```

```
also have ‹... = plus (len (Cons x xs)) (len ys)›
  by simp
finally show ?case
  by simp
qed
```

Our first proof IV

Alternatively:

```
theorem ‹len (app xs ys) = plus (len xs) (len ys)›
proof (induct xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs)
  then show ?case
    by simp
qed
```

Shorter:

```
theorem ‹len (app xs ys) = plus (len xs) (len ys)›
  by (induct xs) simp-all
```

Definitions

Definitions are non-recursive and introduce a layer of indirection.

> **definition** *double* :: ‹*mynat* ⇒ *mynat*› **where**
> ‹*double n* ≡ *plus n n*›

Indirection removed by unfolding:

> **corollary** ‹*len* (*app xs xs*) = *double* (*len xs*)›
> **unfolding** *double-def* **by** (*simp add*: *len-app*)

Alternatively:

> **corollary** ‹*len* (*app xs xs*) = *double* (*len xs*)›
> **unfolding** *double-def* **using** *len-app* **by** *blast*

using makes the stated fact(s) available to the proof method.

Proof methods I

Modify the proof state.
Some simple methods:

rule r, replace current goal with assumptions of *r* if its conclusion unifies with the goal.

"*.*", abbreviation for "**by** *this*"
Used when the goal unifies directly with the stated fact (possibly after unfolding etc.).

Isabelle also includes automatic proof methods.

Proof methods II

simp, the simplifier, rewrites terms using various rules and contextual information.

- May loop.
- Functions, definitions, lemmas can give rise to rewrite rules ([*simp*] attribute).

auto combines the simplifier with classical reasoning among other things.

lemma $\langle (xs = app\ xs\ ys) = (ys = Nil) \rangle$
  by $(induct\ xs)$ *auto*

force performs a “rather exhaustive search” using “many fancy proof tools”

theorem *Cantor*: $\langle \nexists f :: nat \Rightarrow nat\ set.\ \forall A.\ \exists x.\ f\ x = A \rangle$
  by *force*

Proof methods III

blast is a classical tableau prover.

- Does not use the simplifier.
- Written to be very fast.
- Proof is reconstructed in Isabelle afterwards.

If everyone that is not rich has a rich father, then some rich person must have a rich grandfather.

lemma $\langle(\forall x.\ (\neg\ r(x) \longrightarrow r(f(x)))) \longrightarrow (\exists x.\ (r(x) \wedge r(f(f(x)))))\rangle$
by *blast*

Proof methods IV

fast uses sequent-style proving.

- Breadth-first search strategy.
- Constructs an Isabelle proof directly.
- *fastforce* combines it with the simplifier.

Any list built by concatenating another one with itself has even length.

lemma ‹$\forall\, xs \in A.\ \exists\, ys.\ xs = app\ ys\ ys \Longrightarrow us \in A \Longrightarrow$
$\exists\, n.\ len\ us = plus\ n\ n$›
using *len-app* **by** *fast*

blast and *fast* use classical reasoning, *iprover* uses only intuitionistic logic.

Proof methods V

metis implements ordered paramodulation.

- Very powerful.
- Does not use the simplifier.

If a number acts as the identity for plus, it must be zero.

lemma ‹$\forall x.\ plus\ x\ y = x \Longrightarrow y = Zero$›
using *plus.simps*(*1*) **by** *metis*

A suitable proof method can be found with **try0**.

Our second proof I

Reverse a list in linear time in the size of the list:

$$\textbf{primrec}\ \mathit{rev}' :: \langle {}'a\ \mathit{mylist} \Rightarrow {}'a\ \mathit{mylist} \Rightarrow {}'a\ \mathit{mylist}\rangle\ \textbf{where}$$
$$\langle \mathit{rev}'\ \mathit{Nil}\ \mathit{acc} = \mathit{acc}\rangle$$
$$\mid \langle \mathit{rev}'\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{acc} = \mathit{rev}'\ \mathit{xs}\ (\mathit{Cons}\ x\ \mathit{acc})\rangle$$

Hide details of accumulator behind a definition:

$$\textbf{definition}\ \mathit{rev} :: \langle {}'a\ \mathit{mylist} \Rightarrow {}'a\ \mathit{mylist}\rangle\ \textbf{where}$$
$$\langle \mathit{rev}\ \mathit{xs} = \mathit{rev}'\ \mathit{xs}\ \mathit{Nil}\rangle$$

Our second proof II

Goal: Prove the length is preserved. First attempt:

lemma ‹*len* (*rev xs*) = *len xs*›
proof (*induct xs*)
  case *Nil*
  then show *?case*
    unfolding *rev-def* **by** *simp*
next

  case (*Cons x xs*)
  have ‹*len* (*rev* (*Cons x xs*)) = *len* (*rev′* (*Cons x xs*) *Nil*)›
    unfolding *rev-def* **by** *blast*
  also have ‹... = *len* (*rev′ xs* (*Cons x Nil*))›
    by *simp*
  show *?case* **sorry**
qed

IH *len (rev xs) = len xs*

unfolded *len (rev' xs **Nil**) = len xs*

Our second proof III

We need to apply the induction hypothesis to an *arbitrary* accumulator. Second attempt:

```
lemma ‹len (rev' xs acc) = plus (len xs) (len acc)›
proof (induct xs arbitrary: acc)
```

```
  case Nil
  then show ?case
    by simp
next
```

Our second proof IV

?case *len (rev' (Cons x xs) acc) = plus (len (Cons x xs) (len acc)*

IH *len (rev' xs* ***?acc****) = plus (len xs) (len* ***?acc****)*

```
  case (Cons x xs)
  have ‹len (rev′ (Cons x xs) acc) = len (rev′ xs (Cons x acc))›
    by simp
  also have ‹. . . = plus (len xs) (len (Cons x acc))›
    using Cons by blast
```

Cons x is in the wrong place. . . Rewrite:

```
  also have ‹. . . = Succ (plus (len xs) (len acc))›
    by (induct xs) simp-all
  also have ‹. . . = plus (len (Cons x xs)) (len acc)›
    by simp
  finally show ?case .
qed
```

Our second proof V

Simplifier rule:

lemma *plus-right-succ* [*simp*]:
  ‹*plus n* (*Succ m*) = *Succ* (*plus n m*)›
  by (*induct n*) *simp-all*

Automatic proof:

lemma *len-rev′*: ‹*len* (*rev′ xs acc*) = *plus* (*len xs*) (*len acc*)›
  by (*induct xs arbitrary*: *acc*) *simp-all*

Our second proof VI

Another fact about addition:

lemma *plus-right-zero* [*simp*]: ‹*plus n Zero* = *n*›
  by (*induct n*) *simp-all*

Finally we can relate it to the definition:

lemma *len-rev*: ‹*len* (*rev xs*) = *len xs*›
  unfolding *rev-def* **by** (*simp add*: *len-rev*′)

DTU

Section 2

Quicksort

Simple functional version

Given an input list l. If l is empty it is already sorted. Otherwise l has shape $x \# xs$:

1. Split xs into as and zs where
   - $\forall a \in \mathrm{set}\ as.\ a \leq x$
   - $\forall z \in \mathrm{set}\ zs.\ x < z$.
2. Recursively sort as and zs into as' and zs'
3. Return $as' \mathbin{@} x \# zs'$

where @ appends two lists.

Termination: Base case is covered and as and zs are strictly smaller than l so the recursion is well-founded.

Isabelle formulation I

I will show the entire theory, in order, here.

theory *Quicksort* **imports** *HOL−Library.Multiset* **begin**

abbreviation *le* :: ‹('a::linorder) ⇒ 'a ⇒ bool› **where**
‹le x y ≡ y ≤ x›

Intended for partial application:
Predicate *le x* holds for all elements less than or equal to x.

Isabelle formulation II

```
fun quicksort :: ‹('a::linorder) list ⇒ 'a list› where
  ‹quicksort [] = []›
| ‹quicksort (x # xs) =
   (let (as, zs) = partition (le x) xs
    in quicksort as @ x # quicksort zs)›
```

Termination automatically proven.

Unit test

```
lemma ‹quicksort [8,1,5,2,0,9,1,4 :: int] = [0,1,1,2,4,5,8,9]›
  by eval
```

Goal

Properties for sort

properties_for_sort: $mset\ ?ys = mset\ ?xs \Longrightarrow sorted\ ?ys \Longrightarrow sort\ ?xs = ?ys$

where $?ys = quicksort\ ?xs$

So, proving for all xs:

Permutation $mset\ (quicksort\ xs) = mset\ xs$

Sorting $sorted\ (quicksort\ xs)$

Gives us a proof that

$sort\ xs = quicksort\ xs$

Multisets

Unordered collections of elements, e.g.:

$$\{a, a, b\} = \{a, b, a\}$$

$$\{a, a, b\} \neq \{a, b\}$$

Also known as *bags*.

Library support in Isabelle:

$(+)$ :: *'a multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *'a multiset*

mset :: *'a list* $\Rightarrow$ *'a multiset*

set-mset :: *'a multiset* $\Rightarrow$ *'a set*

set-mset-mset: *set-mset (mset ?xs)* $=$ *set ?xs*

Permutation I

Induction over the recursive calls by the algorithm:

lemma *quicksort-permutes* [*simp*]:
  ‹*mset* (*quicksort xs*) = *mset xs*›
proof (*induct xs rule*: *quicksort.induct*)

?case (1) *mset (quicksort []) = mset []*

  case *1*
  show *?case* **by** *simp*
next

Permutation II

?case (2) $\mathit{mset}\ (\mathit{quicksort}\ (x\ \#\ \mathit{xs})) = \mathit{mset}\ (x\ \#\ \mathit{xs})$

IH (as) $(\mathit{as},\mathit{zs}) = \mathit{partition}\ (\mathit{le}\ x)\ \mathit{xs} \Longrightarrow \mathit{mset}\ (\mathit{quicksort}\ \mathit{as}) = \mathit{mset}\ \mathit{as}$

IH (zs) $(\mathit{as},\mathit{zs}) = \mathit{partition}\ (\mathit{le}\ x)\ \mathit{xs} \Longrightarrow \mathit{mset}\ (\mathit{quicksort}\ \mathit{zs}) = \mathit{mset}\ \mathit{zs}$

Compare to

$\mathit{mset}\ (\mathit{quicksort}\ \mathit{xs}) = \mathit{mset}\ \mathit{xs}$

Difficult to relate to as and zs.

Permutation III

?case (2) $mset\ (quicksort\ (x \# xs)) = mset\ (x \# xs)$

```
    case (2 x xs)
    moreover obtain as zs where ‹(as, zs) = partition (le x) xs›
      by simp
```

```
    moreover from this have ‹mset as + mset zs = mset xs›
      by (induct xs arbitrary: as zs) simp-all
    ultimately show ?case
      by simp
  qed
```

from includes facts by name while
moreover accumulates them until
ultimately uses them.
obtain eliminates an existential.

Permutation IV

Corollary for regular sets:

corollary *set-quicksort* [*simp*]: ‹*set* (*quicksort xs*) = *set xs*›
using *quicksort-permutes set-mset-mset* **by** *metis*

Question of what to add to the simplifier. Balance:

- Clutter at use-site.
- Justification of proof steps.
- Dependency visibility.

Sorted I

Proof by induction over recursive calls again:

```
lemma quicksort-sorts [simp]: ‹sorted (quicksort xs)›
proof (induct xs rule: quicksort.induct)
```

?case (1) *sorted (quicksort [])*

```
  case 1
  show ?case by simp
next
```

The empty case is trivial.
Hurray for the simplifier figuring out the details.

Sorted II

?case (2) *sorted (quicksort (x # xs))*

```
case (2 x xs)
obtain as zs where *: ‹(as, zs) = partition (le x) xs›
  by simp
then have
  ‹∀ a ∈ set (quicksort as). ∀ z ∈ set (x # quicksort zs). a ≤ z›
  by fastforce
```

```
  then have ‹sorted (quicksort as @ x # quicksort zs)›
    using * 2 set-quicksort
    by (metis linear partition-P sorted-Cons sorted-append)
  then show ?case
    using * by simp
qed
```

then $\equiv$ **from** *this*.

sledgehammer can find the *metis* proof.

Sorted III

$\text{ext:}\ (\bigwedge x.\ ?f\ x = ?g\ x) \Longrightarrow ?f = ?g$

theorem *sort-quicksort*: ‹*sort* = *quicksort*›
  using *properties-for-sort* **by** (*rule ext*) *simp-all*

Thank you

```
end
```

Questions?

References:

- The Isabelle/Isar Reference Manual, Makarius Wenzel
- Miscellaneous Isabelle/Isar examples, Makarius Wenzel
- Defining Recursive Functions in Isabelle/HOL, Alexander Krauss

Thanks to Jørgen Villadsen for help refining the quicksort example and to him, Anders Schlichtkrull and John Bruntse Larsen for feedback on a written version of this talk.