FIT - From’s Isabelle Tutorial

Verification of Quicksort

Andreas Halkjær From
Contents

• Introduction
  • Data types
  • Primitive recursion
  • Our first proof
  • Definitions
  • Proof methods
  • Our second proof

• Quicksort
  • Formulation
  • Multisets
  • Permutation
  • Sorted
  • References
Section 1

Introduction
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.
Introduction

Isabelle II

Theorem Drinker's Principle: \( \exists x. \text{drunk } x \rightarrow (\forall x. \text{drunk } x) \)

Proof cases

- Assume \( \forall x. \text{drunk } x \)
- Then have \( \text{drunk } a \rightarrow (\forall x. \text{drunk } x) \) for \( a \).
- Then show \( \text{thesis} \).

Next

- Assume \( \neg (\forall x. \text{drunk } x) \)
- Then have \( \exists x. \neg \text{drunk } x \) by (rule de_Morgan)
- Then obtain \( a \) where \( \neg \text{drunk } a \)
- Have \( \text{drunk } a \rightarrow (\forall x. \text{drunk } x) \)
- Proof \( \text{drunk } a \)
- With \( \neg \text{drunk } a \) show \( \exists x. \text{drunk } x \) by contradiction
- Qed
- Then show \( \text{thesis} \).
- Qed

Have \( (\forall a. \neg \text{drunk } a \rightarrow \text{thesis}) \rightarrow \text{thesis} \)
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.
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.
Introduction

Data types

Algebraic data types like Standard ML.

```plaintext
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 ⇒.
No empty types ala:

```plaintext
datatype 'a stream = Cons 'a 'a stream
```

Need codatatypes.
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)⟩
```
We can state properties about the programs directly:

```
theorem len-app: \langle \text{len} (\text{app} \; \text{xs} \; \text{ys}) = \text{plus} \; (\text{len} \; \text{xs}) \; (\text{len} \; \text{ys}) \rangle
```

\text{xs} and \text{ys} are automatically universally quantified.

Proof by induction:

```
proof (induct xs)
```

Splits the goal into a case for each constructor of \text{xs}.

- \text{Nil} \quad \text{len} (\text{app} \; \text{Nil} \; \text{ys}) = \text{plus} \; (\text{len} \; \text{Nil}) \; (\text{len} \; \text{ys})
- \text{Cons} \quad \text{len} (\text{app} \; (\text{Cons} \times \text{xs}) \; \text{ys}) = \text{plus} \; (\text{len} \; (\text{Cons} \times \text{xs})) \; (\text{len} \; \text{ys})
Our first proof II (Nil)

\[ \text{case \ len (app Nil ys) = plus (len Nil) (len ys)} \]

Equational reasoning.

\[
\begin{align*}
\text{case Nil} \\
\text{have } \langle \text{len (app Nil ys)} = \text{len ys}\rangle \\
\text{by simp} \\
\text{also have } \langle \ldots = \text{plus Zero (len ys)}\rangle \\
\text{by simp} \\
\text{also have } \langle \ldots = \text{plus (len Nil) (len ys)}\rangle \\
\text{by simp} \\
\text{finally show } \text{?case} \\
\text{by simp}
\end{align*}
\]

\textbf{have} states intermediary facts.  
\textbf{also} chains them together.  
\textbf{finally} completes the chain.
Our first proof III (Cons)

\[ \text{case } \text{len (app (Cons x xs) ys)} = \text{plus (len (Cons x xs)) (len ys)} \]

IH \[ \text{len (app xs ys)} = \text{plus (len xs) (len ys)} \]

\begin{verbatim}
case (Cons x xs)
  have \langle \text{len (app (Cons x xs) ys)} = \text{len (Cons x (app xs ys))} \rangle
      by simp
  also have \langle \ldots = \text{Succ (len (app xs ys))} \rangle
      by simp
  also have \langle \ldots = \text{Succ (plus (len xs) (len ys))} \rangle
      using Cons by simp

  also have \langle \ldots = \text{plus (len (Cons x xs)) (len ys)} \rangle
      by simp
  finally show ?case
      by simp
  qed
\end{verbatim}
Introduction

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 are non-recursive and introduce a layer of indirection.

**definition** `double :: ∀mynat ⇒ mynat` where

\[ double \ n ≡ plus \ n \ n \]

Indirection removed by unfolding:

**corollary** \[ \text{len} \ (\text{app} \ xs \ xs) = double \ (\text{len} \ xs) \]
unfolding `double-def` by `(simp add: len-app)`

Alternatively:

**corollary** \[ \text{len} \ (\text{app} \ xs \ xs) = double \ (\text{len} \ xs) \]
unfolding `double-def` using `len-app` by `blast`

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

**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 ⟨(xs = app xs ys) = (ys = Nil)⟩
  by (induct xs) auto
```

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

```
theorem Cantor: ⟨!! f :: nat ⇒ nat set. ∀ A. ∃ x. f x = A⟩
  by force
```
\textit{blast} is a classical tableau prover.

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

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

\texttt{lemma} \langle (\forall x. (\neg r(x) \rightarrow r(f(x)))) \rightarrow (\exists x. (r(x) \land r(f(f(x))))) \rangle \text{ by blast}
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 \( \forall xs \in A. \exists ys. xs = \text{app} ys ys \implies us \in A \implies \exists n. \text{len} us = \text{plus} n n \) using \text{len-app} by fast
```

blast and fast use classical reasoning, iprover uses only intuitionistic logic.
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 (∀ x. plus x y = x ⟹ y = Zero)
using plus.simps(1) by metis
```

A suitable proof method can be found with **try0**.
Reverse a list in linear time in the size of the list:

\[ \textbf{primrec } \textit{rev': } \langle 'a \textit{mylist} \Rightarrow 'a \textit{mylist} \Rightarrow 'a \textit{mylist} \rangle \textbf{ where} \]
\[ \langle \textit{rev'} \text{ Nil } \textit{acc} = \textit{acc} \rangle \]
\[ | \langle \textit{rev'} (\textit{Cons } x \textit{ xs}) \textit{ acc} = \textit{rev'} \textit{ xs} (\textit{Cons } x \textit{ acc}) \rangle \]

Hide details of accumulator behind a definition:

\[ \textbf{definition } \textit{rev}: \langle 'a \textit{mylist} \Rightarrow 'a \textit{mylist}\rangle \textbf{ where} \]
\[ \langle \textit{rev} \textit{ xs} = \textit{rev'} \textit{ xs} \textit{ Nil} \rangle \]
Our second proof II

Goal: Prove the length is preserved. First attempt:

\begin{quote}
\begin{verbatim}
lemma \langle \text{len (rev } xs) = \text{len } xs \rangle

proof (induct xs)
  case \text{Nil}
  then show \text{?case}
    unfolding \text{rev-def by simp}
  next

  case \text{(Cons x xs)}
  have \langle \text{len (rev (Cons x xs)) = len (rev' (Cons x xs) Nil)} \rangle
    unfolding \text{rev-def by blast}
  also have \langle \ldots = \text{len (rev' xs (Cons x Nil))} \rangle
    by \text{simp}
  show \text{?case sorry}

qed
\end{verbatim}
\end{quote}

IH \text{len (rev } xs) = \text{len } xs

unfolded \text{len (rev' xs Nil) = len } xs
We need to apply the induction hypothesis to an arbitrary accumulator. Second attempt:

**Lemma** \(\text{len } (\text{rev'} \ xs \ acc) = \text{plus } (\text{len } xs) \ (\text{len } acc)\)

**Proof** \(\text{(induct } xs \ \text{arbitrary: } acc)\)

**Case** Nil

**Then show** ?case

**By** simp

**Next**
Our second proof IV

\[ \text{?case } \text{len } (\text{rev'} (\text{Cons } x \ x) \ \text{acc}) = \text{plus} (\text{len } (\text{Cons } x \ x) \ (\text{len } \text{acc})) \]

IH \[ \text{len } (\text{rev'} x \ \text{acc}) = \text{plus} (\text{len } x \ x) \ (\text{len } \text{acc}) \]

\begin{align*}
\text{case } (\text{Cons } x \ x) \\
\text{have } \langle \text{len } (\text{rev'} (\text{Cons } x \ x) \ \text{acc}) = \text{len } (\text{rev'} x \ (\text{Cons } x \ \text{acc})) \rangle \\
\text{by simp} \\
\text{also have } \langle \ldots = \text{plus} (\text{len } x \ x) \ (\text{len } (\text{Cons } x \ \text{acc})) \rangle \\
\text{using Cons by blast} \\
\end{align*}

Cons \ x \ \text{is in the wrong place... Rewrite:}

\begin{align*}
\text{also have } \langle \ldots = \text{Succ } (\text{plus } (\text{len } x \ x) \ (\text{len } \text{acc})) \rangle \\
\text{by } \langle \text{induct } x \ \text{simp-all} \rangle \\
\text{also have } \langle \ldots = \text{plus } (\text{len } (\text{Cons } x \ x)) \ (\text{len } \text{acc}) \rangle \\
\text{by simp} \\
\text{finally show } \text{?case } . \\
\text{qed} \\
\end{align*}
Introduction

Our second proof V

Simplifier rule:

\textbf{lemma} plus-right-succ \textbf{[simp]}:
\[
\langle \text{plus}\ n\ (\text{Succ}\ m) = \text{Succ}\ (\text{plus}\ n\ m) \rangle
\]
\textbf{by} (induct\ n)\ simp-all

Automatic proof:

\textbf{lemma} len-rev\': \langle len\ (\text{rev}'\ xs\ acc) = plus\ (len\ xs)\ (len\ acc) \rangle
\textbf{by} (induct\ xs\ arbitrary: acc)\ simp-all
Another fact about addition:

\textbf{lemma} plus-right-zero \([\text{simp}]: \langle plus \ n \ \text{Zero} = n \rangle\)
\textbf{by} (induct \ n) simp-all

Finally we can relate it to the definition:

\textbf{lemma} len-rev: \(\langle len \ (rev \ xs) = len \ xs \rangle\)
\textbf{unfolding} rev-def \textbf{by} (simp add: len-rev')
Section 2

Quicksort
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
   - $\forall a \in \text{set } as. \ a \leq x$
   - $\forall z \in \text{set } zs. \ x < z$.

2. Recursively sort $as$ and $zs$ into $as'$ and $zs'$

3. Return $as' \ @ x \# zs'$

where $\@$ appends two lists.

Termination: Base case is covered and $as$ and $zs$ are strictly smaller than $l$ so the recursion is well-founded.
I will show the entire theory, in order, here.

```isar
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` holds for all elements less than or equal to `x`.```
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
Properties for sort

properties_for_sort: \( mset \ ?ys = mset \ ?xs \implies sorted \ ?ys \implies sort \ ?xs = ?ys \)

where \( ?ys = quicksort \ ?xs \)

So, proving for all \( xs \):

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

Sorting \( sorted \ (quicksort \ xs) \)

Gives us a proof that

\( sort \ xs = quicksort \ xs \)
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:

\begin{align*}
(+) & : \text{'a multiset} \Rightarrow \text{'a multiset} \Rightarrow \text{'a multiset} \\
mset & : \text{'a list} \Rightarrow \text{'a multiset} \\
set-mset & : \text{'a multiset} \Rightarrow \text{'a set} \\
set-mset-mset: & \text{set-mset (mset ?xs) = set ?xs}
\end{align*}
Induction over the recursive calls by the algorithm:

**lemma** quicksort-permutes [simp]:
\[\langle mset (quicksort xs) = mset xs \rangle\]
**proof** (induct xs rule: quicksort.induct)

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

  case 1
  show ?case by simp
next
QuickSort

Permutation II

\(?\text{case (2)}\) \( \text{mset}\ (\text{quicksort}\ (x \neq xs)) = \text{mset}\ (x \neq xs) \)

\( \text{IH (as)}\) \( (as,zs) = \text{partition}\ ((\leq x))\ xs \implies \text{mset}\ (\text{quicksort}\ as) = \text{mset}\ as \)

\( \text{IH (zs)}\) \( (as,zs) = \text{partition}\ ((\leq x))\ xs \implies \text{mset}\ (\text{quicksort}\ zs) = \text{mset}\ zs \)

Compare to

\( \text{mset}\ (\text{quicksort}\ xs) = \text{mset}\ xs \)

Difficult to relate to \( as \) and \( zs \).
?case (2) \text{mset} (\text{quicksort} (x \# xs)) = \text{mset} (x \# xs)

\text{case} (2 \times xs)
\text{moreover obtain} \ as \ zs \ 
\text{where} \ \langle as, zs \rangle = \text{partition} \ (le \ x) \ xs
\text{by simp}

\text{moreover from} \ this \ have \ \langle \text{mset} \ as + \text{mset} \ zs = \text{mset} \ xs \rangle
\text{by (induct} \ xs \ \text{arbitrary:} \ as \ zs) \ \text{simp-all}
\text{ultimately show} \ ?\text{case}
\text{by simp}
\text{qed}

\text{from} \ includes \ facts \ by \ name \ while
\text{moreover} \ accumulates \ them \ until
\text{ultimately} \ uses \ them.
\text{obtain} \ eliminates \ an \ existential.
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.
Proof by induction over recursive calls again:

```isar
lemma quicksort-sorts [simp]: \{\text{sorted \( (\text{quicksort} \; \text{xs}) \) \}}
proof \( \text{induct \; \text{xs \; rule: \; quicksort.induct} \) }

\text{?case (1) sorted \( (\text{quicksort} \; [\]) \) }

case 1
show ?case by simp
next
```

The empty case is trivial.
Hurray for the simplifier figuring out the details.
case (2) sorted (quicksort (x ≠ xs))

\[\text{case } (2 \times xs)\]
\[\text{obtain } as \text{ zs where } *: \langle as, zs \rangle = \text{partition } (le x) \text{ xs}\]
\[\text{by simp}\]
\[\text{then have}\]
\[\forall a \in \text{set } (\text{quicksort as}). \forall z \in \text{set } (x \# \text{quicksort zs}). a \leq z\]
\[\text{by fastforce}\]

then have \(\langle \text{sorted } (\text{quicksort as } @ x \# \text{quicksort zs})\rangle\)
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 \textit{metis} proof.
ext: ($\forall x. \ ?f \ x = \ ?g \ x$) $\implies$ $?f = ?g$

**Theorem** sort-quicksort: $\langle \text{sort} = \text{quicksort} \rangle$

**Using** properties-for-sort by (rule ext) simp-all
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.*