

Interactive Formal Verification

Lawrence C Paulson
Computer Laboratory
University of Cambridge

This lecture course introduces interactive formal proof using Isabelle. The lecture notes consist of copies of the slides, some of which have brief remarks attached. Isabelle documentation can be found on the Internet at the URL <http://www.cl.cam.ac.uk/research/hvg/Isabelle/documentation.html>. Documentation is also available from the sidebar when running Isabelle. The first manual shown (*Programming and Proving in Isabelle/HOL*) is the most useful, but the much older *Tutorial on Isabelle/HOL* (available in the sidebar under Old Manuals) is more comprehensive. Please note that the style of proof it presents is now quite outdated.

The other tutorials listed on the documentation page (or sidebar) are specialised and mainly for advanced users.

Interactive Formal Verification

I: Introduction

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Motivation

- Complex systems almost inevitably contain bugs.
- Debugging suffers from diminishing returns. Many critical bugs are *never fixed!*
- Critical systems (avionics, ...) are required to meet a standard of 10^{-9} failures per hour. Testing to such a standard is infeasible.

“Program testing can be used to show the presence of bugs, but never to show their absence!”
— Edsger W. Dijkstra

What is Interactive Proof?

- Prove theorems in a *logical formalism*
 - with precise definitions of concepts
 - a formal deductive system
 - and automatic tools
- Create hierarchies of *definitions and proofs*
 - specifications of components and properties
 - proofs that designs meet their requirements

Interactive Theorem Provers

- Based on higher-order logic
 - Isabelle, HOL (many versions), PVS
- Based on constructive type theory
 - Coq, Twelf, Agda, ...
- Based on first-order logic with recursion
 - ACL2

Here are some useful web links:

Isabelle: <http://www.cl.cam.ac.uk/research/hvg/Isabelle/>
HOL4: <http://hol.sourceforge.net/>
HOL Light: <http://www.cl.cam.ac.uk/~jrh13/hol-light/>
PVS: <http://pvs.cs1.sri.com/>
Coq: <http://coq.inria.fr/>
ACL2: <http://www.cs.utexas.edu/users/moore/acl2/>

The LCF Architecture

- A small *kernel* implements the logic and can generate theorems.
- All specification methods and automatic proof procedures expand to full proofs.
- Unsoundness is less likely with this architecture
- ... but the implementation is more complicated, and performance can suffer.
- Used in Isabelle, HOL, Coq *but not* PVS or ACL2.

Theorem Provers: Key Features

- Logical formalism (higher-order, type theory etc.)
- Control issues:
 - User interface / Proof language
 - Automation
- Libraries of *formalised mathematics*
- Tools: typesetting, library search and more!

Isabelle

- Isabelle is a generic interactive theorem prover, developed by Lawrence Paulson (Cambridge) and Tobias Nipkow (Munich). First released in 1986.
- Integrated tool support for
 - Automated provers
 - Counter-example finding
 - Code generation from logical terms
 - LaTeX document generation

Higher-Order Logic

- First-order logic extended with functions and sets
- Polymorphic types, including a type of truth values
- No distinction between terms and formulas
- ML-style functional programming

“HOL = functional programming + logic”

Basic Syntax of Formulas

formulas A, B, \dots can be written in ASCII as

(A)	$t = u$	$\sim A$
$A \& B$	$A B$	$A \dashrightarrow B$
$A \leftrightarrow B$	$!x.A$	$?x.A$

(Among many others)

Isabelle also supports symbols such as

$\leq \geq \neq \wedge \vee \rightarrow \leftrightarrow \forall \exists$

See the *Tutorial*, section 1.3: “Types, terms and formulae”. The ASCII notation for logical symbols can be used to input them to Isabelle, which will offer to replace them by special symbols.

See the *Tutorial*, section 1.3: “Types, terms and formulae”

Some Syntactic Conventions

In $\forall x. A \wedge B$, the quantifier spans the entire formula

Parentheses are **required** in $A \wedge (\forall x y. B)$

Binary logical connectives associate to the **right**:

$A \rightarrow B \rightarrow C$ is the same as $A \rightarrow (B \rightarrow C)$

$\neg A \wedge B = C \vee D$ is the same as $((\neg A) \wedge (B = C)) \vee D$

See the *Tutorial*, section 1.3: “Types, terms and formulae”

Basic Syntax of Terms

- The typed λ -calculus:
 - constants, c
 - variables, x and *flexible variables*, λx
 - abstractions $\lambda x. t$
 - function applications $t u$
- Numerous infix operators and binding operators for arithmetic, set theory, etc.

Types

- Every term has a type; Isabelle infers the types of terms automatically. We write $t :: \tau$
- Types can be *polymorphic*, with a system of type classes (inspired by the Haskell language) that allows sophisticated overloading.
- A formula is simply a term of type `bool`.
- There are types of ordered pairs and functions.
- Other important types are those of the natural numbers (`nat`) and integers (`int`).

Product Types for Pairs

- (x_1, x_2) has type $\tau_1 \times \tau_2$ provided $x_i :: \tau_i$
- $(x_1, \dots, x_{n-1}, x_n)$ abbreviates $(x_1, \dots, (x_{n-1}, x_n))$
- Extensible record types can also be defined.

Function Types

- Infix operators are curried functions
 - $+ :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$
 - $\& :: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}$
 - Curried function notation: $\lambda x y. t$
- Function arguments can be paired
 - Example: $\text{nat} * \text{nat} \Rightarrow \text{nat}$
 - Paired function notation: $\lambda(x,y). t$

Arithmetic Types

- **nat**: the natural numbers (nonnegative integers)
 - inductively defined: $0, \text{Suc } n$
 - operators include $+ - * \text{div mod}$
 - relations include $< \leq \text{dvd}$ (divisibility)
- **int**: the integers, with $+ - * \text{div mod} \dots$
- **rat, real**: $+ - * / \sin \cos \ln \dots$
- arithmetic constants and laws for these types

Only integer constants are available. Traditional notation for floating point numbers would be inappropriate, but rational numbers can be expressed.

HOL as a Functional Language

```

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list" where
  app Nil ys = ys
| app (Cons x xs) ys = Cons x (app xs ys)

fun rev where
  rev Nil = Nil
| rev (Cons x xs) = app (rev xs) (Cons x Nil)
  
```

recursive data type of lists

recursive functions (types can be inferred)

Recursive data types can be defined as in ML, although with somewhat less generality. Recursive functions can also be declared, provided Isabelle can establish their termination; all functions in higher-order logic are total. Naturally terminating recursive definitions pose no difficulties for Isabelle. In complicated situations, it is possible to give a hint.

Proof by Induction

```

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
  done
  
```

declaring a lemma

use it to simplify other formulas

two steps: induction followed by automation

end of proof

Example of a Structured Proof

- base case and inductive step can be proved explicitly
- Invaluable for proofs that need intricate manipulation of facts

```

lemma "app xs Nil = xs"
proof (induct xs)
  case Nil
  show "app Nil Nil = Nil"
  by auto
  next
  case (Cons a xs)
  show "app (Cons a xs) Nil = Cons a xs"
  by auto
qed
  
```

Interactive Formal Verification 2: Isabelle Theories

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Formal Theories

- Collections of *specifications*: types, constants, functions, sets and relations.
- even *axioms* occasionally, but it is safer to define explicit models satisfying desired properties.

Axiom systems are frequently inconsistent!

- Theories can specify mathematics, formal models or abstract implementations.

See the *Tutorial*, section 1.2 (Theories) and 2.1 (An Introductory Theory).

A Tiny Theory

```
theory BT imports Main begin
datatype 'a bt =
  Lf
| Br 'a "'a bt" "'a bt"
fun reflect :: "'a bt => 'a bt" where
  "reflect Lf = Lf"
| "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"
lemma reflect_reflect_ident: "reflect (reflect t) = t"
  apply (induct t)
  apply auto
  done
end
```

Annotations:

- name of the new theory (points to `theory BT`)
- the theory it builds upon (points to `imports Main`)
- declarations of types, constants, etc (points to `datatype` and `fun`)
- proving a theorem (points to `lemma`)

Notes on Theory Structure

- A theory can *import* any existing theories.
- Types, constants and functions must be *declared before use*.
- The various declarations and proofs may otherwise appear in any order.
- Many declarations can be confined to *local scopes*, which can be nested.
- A finished theory can be imported by others.

Some Fancy Type Declarations

```
typedecl loc --"an unspecified type of locations"
            a new basic type
type_synonym val = nat --"values"
type_synonym state = "loc => val"
type_synonym aexp = "state => val"
type_synonym bexp = "state => bool" --"functions on states"

datatype com = SKIP
            | Assign loc aexp ("_ := _" 60)
            | Semi com com ("_; _" [60, 60] 10)
            | Cond bexp com com ("IF _ THEN _ ELSE _" 60)
            | While bexp com ("WHILE _ DO _" 60)

recursive type of commands
```

concrete syntax for commands

end-of-line comments

See the tutorial, section 2.5.

Notes on Type Declarations

- Type synonyms merely introduce *abbreviations*.
- Recursive data types are less general than in functional programming languages:
 - No recursion into the domain of a function.
 - Mutually recursive definitions can be tricky.
- Recursive types are equipped with proof methods for *induction* and *case analysis*.

Basic Constant Definitions

```
theory Def imports Main begin

definition square :: "nat => nat" where
  "square n = n*n"

definition prime :: "nat => bool" where
  "prime(p::nat) = (1 < p ^ (m dvd p -> m = 1 ^ m = p))"

definition prime :: "nat => bool" where
  "prime p = (1 < p ^ (forall m. m dvd p -> m = 1 ^ m = p))"

end
```

Extra variables on rhs: "m"
The error(s) above occurred in definition:
"prime p = 1 < p ^ (m dvd p -> m = 1 ^ m = p)"

The second one contains an error, which is corrected in the third example.

See the *Tutorial*, Section 2.7.2 Constant Definitions.

Notes on Constant Definitions

- Basic definitions are *not* recursive.
- Every variable on the right-hand side must also appear on the left.
- In proofs, definitions are *not* expanded by default!
 - Defining the constant C to denote t yields the theorem C_def , asserting $C=t$.
- **Abbreviations** can be declared through a separate mechanism.

Extended Example: Lists

- We illustrate data types and functions using a reduced Isabelle theory (one without lists).
- The standard Isabelle environment has a *comprehensive list library*:
 - Functions # (cons), @ (append), map, filter, nth, take, drop, takeWhile, dropWhile, ...
 - Cases: (case xs of [] => [] | x#xs => ...)
- Over 600 theorems!

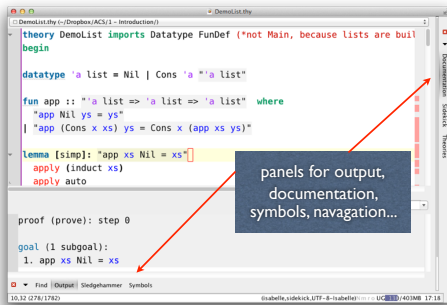
List Induction Principle

To show $\varphi(xs)$, it suffices to show the *base case* and *inductive step*:

- $\varphi(\text{Nil})$
- $\varphi(xs) \Rightarrow \varphi(\text{Cons}(x,xs))$

The principle of case analysis is similar, expressing that any list has one of the forms Nil or Cons(x,xs) (for some x and xs).

Isabelle/jEdit



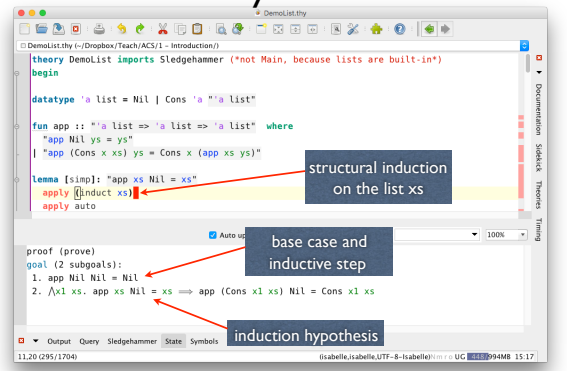
Isabelle's user interface is the work of Makarius Wenzel. The entire proof document is processed as far as possible, errors and all.

Isabelle/jEdit allows inspection of proof states and the declarations of identifiers, symbols and even Isabelle keywords.

All documentation is accessible from the sidebar.

Launch using the command "isabelle jedit FILENAME"

Proof by Induction



See the tutorial, section 2.3 (An Introductory Proof). For the moment, there is no important difference between `induct_tac` (used in the tutorial) and `induct` (used above). With both of these proof methods, you name an induction variable and it selects the corresponding structural induction rule, based on that variable's type. It then produces an instance of induction sufficient to prove the property in question.

Finishing a Proof

```
datatype 'a list = Nil | Cons 'a 'a list
fun app :: "'a list => 'a list => 'a list" where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"
lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto
  done
fun rev where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
proof (prove)
  goal:
  No subgoals!
```

By default, Isabelle simplifies applications of recursive functions that match their defining recursion equations. This is quite different to the treatment of non-recursive definitions.

Another Proof Attempt

```
datatype 'a list = Nil | Cons 'a 'a list
fun rev where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
proof (prove)
  goal (2 subgoals):
  1. rev (rev Nil) = Nil
  2.  $\lambda x l xs. rev (rev xs) = xs \implies rev (rev (Cons x l xs)) = Cons x l xs$ 
```

Stuck!

```
datatype 'a list = Nil | Cons 'a 'a list
fun rev where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"
lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
proof (prove)
  goal (1 subgoal):
  1.  $\lambda x l xs. rev (rev xs) = xs \implies rev (app (rev xs) (Cons x l Nil)) = Cons x l xs$ 
```

It isn't clear how to continue, but it is clear that we should be able to do something with terms of the form $rev (app xs ys)$.

Stuck Again!

```
fun rev where
  "rev Nil = Nil"
  | "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

Lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

Lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)

proof (prove)
goal (1 subgoal):
1.  $\forall x1\ xs.$ 
  rev (app xs ys) = app (rev ys) (rev xs)  $\implies$ 
  app (app (rev ys) (rev xs)) (Cons x1 Nil) = app (rev ys) (app (rev xs) (Cons x1 Nil))
```

The subgoal that we cannot prove looks complicated. But when we notice the repeated terms, we see that it is an instance of something simple: the associativity of the function app. This fact does not involve the function rev! We see in this example how crucial it is to prove properties in the most abstract and general form.

The Final Piece of the Jigsaw

```
fun rev where
  "rev Nil = Nil"
  | "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

Lemma [simp]: "app (app xs ys) zs = app xs (app ys zs)"
  apply (induct xs)
  apply auto
  done

Lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)

proof (prove)
goal (2 subgoals):
1. app (app Nil ys) zs = app Nil (app ys zs)
2.  $\forall x1\ xs.$ 
  app (app xs ys) zs = app xs (app ys zs)  $\implies$ 
  app (app (Cons x1 xs) ys) zs = app (Cons x1 xs) (app ys zs)
```

This proof of associativity will be successful, and with its help, the other lemmas are easily proved.

The Finished Proof

```
fun rev where
  "rev Nil = Nil"
  | "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

Lemma [simp]: "app (app xs ys) zs = app xs (app ys zs)"
  apply (induct xs)
  apply auto
  done

Lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

Lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

The lemmas must be proved in the correct order. Each is needed to prove the next.

It is actually more usable to give each lemma a name and to supply the relevant names to auto. The two lemmas proved above, especially the associativity of append, do not look like they would always be useful in simplification, so normally they would be proved without the [simp] attribute.

Interactive Formal Verification

3: Elementary Proof

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Elements of Interactive Proof

- Quite a few theorems can be proved by a combination of *induction* and *simplification*.
- Induction can be a straightforward *structural induction* rule derived from a type declaration, but other induction rules are quite specialised.
- Simplification typically refers to *rewriting* according to the definition of a recursive function...
- but it has many refinements, including automatic *case splitting*, simple logical reasoning and *arithmetic* reasoning.

Goals and Subgoals

- We start with a *goal*: the statement to be proved.
- Proof *tactics* and *methods* typically replace a single subgoal by zero or more new subgoals.
- [But certain methods, notably `auto` and `simp_all`, operate on *all* outstanding subgoals.]
- When no subgoals remain, *the theorem is proved!*

See the Tutorial, 2.3 An Introductory Proof. The list of subgoals is always flat. However, Isabelle supports structured proofs and they are covered later in the course.

Structure of a Subgoal

The screenshot shows a theorem prover interface with a code editor and a subgoal window. The code editor contains the following text:

```

datatype 'a bt =
  Lf
| Br 'a "'a bt" "'a bt"
fun reflect :: "'a bt => 'a bt" where
  "reflect Lf = Lf"
| "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"
lemma reflect_reflect_ident: "reflect (reflect t) = t"
apply [induct t]
apply auto
    
```

The subgoal window displays the following list of items:

1. reflect (reflect Lf) = Lf
2. $\forall a\ t1\ t2.$
 - reflect (reflect t1) = t1 \implies
 - reflect (reflect t2) = t2 \implies
 - reflect (reflect (Br a t1 t2)) = Br a t1 t2

Annotations in the image point to different parts of the subgoal:

- assumptions (two induction hypotheses)**: Points to the first two items in the list.
- parameters (arbitrary local variables)**: Points to the variables a , $t1$, and $t2$ in the second item.
- conclusion**: Points to the final equality in the second item.

Proof by Rewriting

$app\ (Cons\ ?x\ ?xs)\ ?ys \rightarrow Cons\ ?x\ (app\ ?xs\ ?ys)$ ← recursive defs
 $rev\ (Cons\ ?x\ ?xs) \rightarrow app\ (rev\ ?xs)\ (Cons\ ?x\ Nil)$ ← recursive defs
 $rev\ (app\ xs\ ys) \rightarrow app\ (rev\ ys)\ (rev\ xs)$ ← induction hyp (for fixed xs, ys)
 $app\ (app\ ?xs\ ?ys)\ ?zs \rightarrow app\ ?xs\ (app\ ?ys\ ?zs)$ ← lemma about app

$rev\ (app\ (Cons\ a\ xs)\ ys) = app\ (rev\ ys)\ (rev\ (Cons\ a\ xs))$
 $rev\ (app\ (Cons\ a\ xs)\ ys) =$
 $rev\ (Cons\ a\ (app\ xs\ ys)) =$
 $app\ (rev\ (app\ xs\ ys))\ (Cons\ a\ Nil) =$
 $app\ (app\ (rev\ ys)\ (rev\ xs))\ (Cons\ a\ Nil) =$
 $app\ (rev\ ys)\ (app\ (rev\ xs)\ (Cons\ a\ Nil))$

$app\ (rev\ ys)\ (rev\ (Cons\ a\ xs)) =$
 $app\ (rev\ ys)\ (app\ (rev\ xs)\ (Cons\ a\ Nil))$

At each step, the highlighted term is rewritten to something else. Eventually, the left hand side and right hand side of the desired equation have become equal. (This equation is the induction step for our lemma, $rev\ (app\ xs\ ys) = app\ (rev\ ys)\ (rev\ xs)$.)

The equalities on the slide are fully general with the exception of the third one, which (being an induction hypothesis) holds only for a fixed value of xs . Here we indicate the generality of each variable using $?$ -notation, as follows:

$app\ (Cons\ ?x\ ?xs)\ ?ys = Cons\ ?x\ (app\ ?xs\ ?ys)$
 $rev\ (Cons\ ?x\ ?xs) = app\ (rev\ ?xs)\ (Cons\ ?x\ Nil)$
 $rev\ (app\ xs\ ?ys) = app\ (rev\ ?ys)\ (rev\ xs)$
 $app\ (app\ ?xs\ ?ys)\ ?zs = app\ ?xs\ (app\ ?ys\ ?zs)$

An identifier preceded by $?$ is a true variable and can be substituted by any term, while the other identifiers are fixed.

Rewriting with Equivalences

$(x\ dvd\ -y) = (x\ dvd\ y)$
 $(a * b = 0) = (a = 0 \vee b = 0)$ ← introduces a case split on the sign of c
 $(A - B \subseteq C) = (A \subseteq B \cup C)$
 $(a * c \leq b * c) = ((0 < c \rightarrow a \leq b) \wedge (c < 0 \rightarrow b \leq a))$

- Logical equivalencies are just boolean equations.
- They give a clear, simple proof style.
- They can also be written with the syntax $P \leftrightarrow Q$.

Automatic Case Splitting

Simplification will replace

$$P(\text{if } b \text{ then } x \text{ else } y)$$

by

$$(b \rightarrow P(x)) \wedge (\neg b \rightarrow P(y))$$

- By default, this only happens when simplifying the conclusion. But *assumptions* can also be split.
- Other kinds of case splitting can be enabled.

Conditional Rewrite Rules

$$xs \neq [] \Rightarrow \text{hd } (xs @ ys) = \text{hd } xs$$
$$n \leq m \Rightarrow (\text{Suc } m) - n = \text{Suc } (m - n)$$
$$[| a \neq 0; b \neq 0 |] \Rightarrow b / (a * b) = 1 / a$$

- *First* match the left-hand side, then **recursively** prove the conditions by simplification.
- If successful, applying the resulting rewrite rule.

Termination Issues

- *Looping*: $f(x) = h(g(x))$, $g(x) = f(x+2)$
- *Looping*: $P(x) \Rightarrow x=0$
 - `simp` will try to use this rule to simplify its own precondition!
- $x+y = y+x$ is actually okay!
 - *Permutative rewrite rules* are applied but only if they make the term “lexicographically smaller”.

See the **Tutorial**, 3.1 Simplification. This section describes the options and possibilities thoroughly.

The Methods `simp` and `auto`

- `simp` performs *rewriting* (along with simple arithmetic simplification) on the *first* subgoal
- `auto` simplifies *all subgoals*, not just the first.
- `auto` also applies all obvious *logical steps*
 - Splitting conjunctive goals and disjunctive assumptions
 - Performing obvious quantifier removal

Variations on `simp` and `auto`

The diagram shows several variations of the `simp` and `auto` methods, with red arrows pointing from callout boxes to specific parts of the code:

- `simp add: app_assoc` is annotated with "using another rewrite rule" pointing to `app_assoc` and "omitting a certain rule" pointing to `add:`.
- `simp del: rev_rev (no_asm_simp)` is annotated with "omitting a certain rule" pointing to `del:`.
- `simp (no_asm)` is annotated with "not simplifying the assumptions" pointing to `(no_asm)`.
- `simp_all (no_asm_simp) add: ... del: ...` is annotated with "not simplifying the assumptions" pointing to `(no_asm_simp)` and "ignoring all assumptions" pointing to `(no_asm_simp)`.
- `auto simp add: ... simp del: ...` is annotated with "do simp for all subgoals" pointing to `simp` and "auto with options" pointing to `auto`.

Rules for Arithmetic

- An identifier can denote a *list* of lemmas.
 - `add_ac` and `mult_ac`: associative/commutative properties of addition and multiplication
 - `algebra_simps`: for multiplying out polynomials
 - `field_simps` and `divide_simps`: for multiplying out denominators
- Example:* `auto simp add: field_simps`

These identifiers denote lists of theorems that work together well as rewrite rules for performing various simplification tasks.

Basics of Proof by Induction

- State the desired theorem using “lemma”, with its name and optionally [simp]
- Identify the *induction variable*
 - Its type should be some datatype (incl. nat)
 - It should appear as the *argument of a recursive function*.
- Complicating issues include **unusual recursions** and **auxiliary variables**.

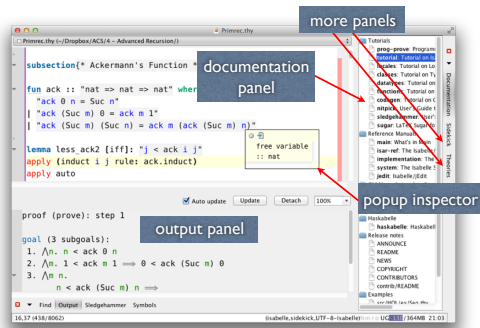
Completing the Proof

- Apply “induction” with the chosen variable.
- The first subgoal will be the base case, and it should be trivial using “simp”.
- Other subgoals will involve induction hypotheses and the proof of each may require several steps.
- Naturally, the first thing to try is “auto”, but much more is possible.

Basics of Isabelle/jEdit

- Based on the jEdit text editor.
- Isabelle automatically processes the entire visible window, errors and all, using parallel threads.
- Identifiers and other elements can be *inspected* using hover-click.
- *Dockable panels* give access to the Isabelle output, theory structure, manuals, symbols, etc.

A View of Isabelle/jEdit



Interactive Formal Verification

4: Advanced Recursion, Induction and Simplification

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Why does Induction Fail?

In a formal proof—like in a program—even trivial errors can be fatal. Everything must be set up *exactly* right...

- The statement being proved is **too weak**, so the induction hypothesis is too weak.
- You have chosen an inappropriate induction rule.
- Or maybe you just don't know how to make use of the induction hypotheses.

A Failing Proof by Induction

The screenshot shows a theorem prover interface with the following code and annotations:

```
fun itlen :: "a list => nat"
  "itlen Nil n = n"
  | "itlen (Cons x xs) n = itlen xs (Suc n)"

lemma "itlen xs n = size xs + n"
  apply (induct xs)
  apply auto
  apply

proof (prove) step 2
  sub (1 subgoal):
  1.  $\forall xs. \text{itlen } xs \ n = \text{size } xs + n \implies \text{itlen } xs \ (\text{Suc } n) = \text{Suc } (\text{size } xs + n)$ 
```

Annotations:

- length of a list (tail-recursive)**: points to the `itlen` function definition.
- equivalent to the built-in length function?**: points to the lemma statement.
- Mismatch between induction hypothesis and conclusion!**: points to the induction hypothesis in the proof step.
- May as well give up!**: points to the `apply` command in the proof step.

Generalising the Induction

Insert a universal quantifier

Induction hypothesis holds for all n

```

fun itlen :: "'a list => nat"
  "itlen Nil n = n"
| "itlen (Cons x xs) n = size xs + n"

lemma "∀n. itlen xs n = size xs + n"
  apply (induct xs)
  apply auto
  done

goal (2 subgoals):
  1. ∀n. itlen Nil n = size Nil + n
  2. ∀n xs.
     ∀n. itlen xs n = size xs + n =>
     ∀n. itlen (Cons a xs) n = size (Cons a xs) + n
  
```

The need to generalise the induction formula in order to obtain a more general induction hypothesis is well known from mathematics. Logically, note that the induction formula above has only one free variable: xs . The induction formula on the previous slide has two free variables: xs and n .

Generalising: A Better Way

Designate a variable as "arbitrary"

Induction hypothesis still holds for all n!

```

fun itlen :: "'a list => nat => nat" where
  "itlen Nil n = n"
| "itlen (Cons x xs) n = itlen xs (Suc n)"

lemma "itlen xs n = size xs + n"
  apply (induct xs arbitrary: n)
  apply auto
  done

proof (prove): step 1
goal (2 subgoals):
  1. ∀n. itlen Nil n = size Nil + n
  2. ∀n xs n.
     (∀n. itlen xs n = size xs + n) =>
     itlen (Cons a xs) n = size (Cons a xs) + n
  
```

The approach described above is logically similar to the one on the previous slide, but it avoids the use of a universal quantifier (\forall) in the theorem statement. Because Isabelle is a logical framework, it has meta-level versions of the universal quantifier and the implication symbol, and we generally avoid universal quantifiers in theorems. But it is important to remember that behind the convenience of the method illustrated here is a straightforward use of logic: we are still generalising induction formula. For more complicated examples, see the Tutorial, 9.2.1 Massaging the Proposition.

Unusual Recursions

Two variables in the induction!

Two variables in the recursion!

A specialised induction rule!

The subgoals follow the recursion!

```

fun ack :: "nat => nat => nat" where
  "ack n 0 = Suc n"
| "ack 0 m = ack m 1"
| "ack (Suc n) (Suc m) = ack m (ack (Suc m) n)"

lemma less_ack_iff: "i < ack i j"
  apply (induct i j rule: ack.induct)
  apply auto
  done

proof (prove): step 1
goal (3 subgoals):
  1. ∀n. n < ack 0 n
  2. ∀n. 1 < ack m 3 => 0 < ack (Suc m) 0
  3. ∀m n. [n < ack (Suc m) n; ack (Suc m) n < ack m (ack (Suc m) n)]
     => Suc n < ack (Suc m) (Suc n)
  
```

For full documentation, see [Defining Recursive Functions in Isabelle/HOL](#), by Alexander Krauss.

Recursion: Key Points

- Recursion in one variable, following the structure of a datatype declaration, is called *primitive*.
- Recursion in multiple variables, terminating by size considerations, can be handled using `fun`.
 - `fun` produces a special induction rule.
 - `fun` can handle **nested recursion**.
 - `fun` also handles *pattern matching*, which it **completes** (if patterns overlap)

Isabelle provides the command `primrec` for primitive recursion as well. It is closely based on the internal derivation of recursion, and can handle function definitions involving certain complicated features (in particular, higher-order primitive recursion) where `fun` fails. See the *Tutorial, 2.1 An Introductory Theory*. More difficult examples of `primrec` are covered in **3.3 Case Study: Compiling Expressions**.

Specialised Induction Rules

- They follow the function's recursion **exactly**.
- For Ackermann, they reduce $P\ x\ y$ to
 - $P\ 0\ n$, for arbitrary n
 - $P\ (Suc\ m)\ 0$ assuming $P\ m\ 1$, for arbitrary m
 - $P\ (Suc\ m)\ (Suc\ n)$ assuming $P\ (Suc\ m)\ n$ and $P\ m\ (ack\ (Suc\ m)\ n)$, for arbitrary m and n
- **Usually** they do what you want. Trial and error is tempting, but ultimately you will need to think!

The Ackermann example proves several lemmas using the special rule, but several others using ordinary mathematical induction!

Another Unusual Recursion

```

fun merge :: "'a list => 'a list => 'a list"
where
  "merge (x#xs) (y#ys) =
    (if x <= y then x # merge xs (y#ys) else y # merge (x#xs) ys)"
| "merge xs [] = xs"
| "merge [] ys = ys"

lemma set_merge [simp]: "set (merge xs ys) = set xs U set ys"
apply(induct xs ys rule: merge.induct)
apply auto
done

1.  $\forall x\ y\ ys.
   (x \leq y \implies \text{set}(\text{merge } xs\ (y\ \# \text{ys})) = \text{set } xs\ \cup\ \text{set } (y\ \# \text{ys})) \implies
   (\neg x \leq y \implies \text{set}(\text{merge } (x\ \# \text{xs})\ \text{ys}) = \text{set } (x\ \# \text{xs})\ \cup\ \text{set } \text{ys}) \implies
   \text{set}(\text{merge } (x\ \# \text{xs})\ (y\ \# \text{ys})) = \text{set } (x\ \# \text{xs})\ \cup\ \text{set } (y\ \# \text{ys})
2.  $\forall xs. \text{set}(\text{merge } xs\ []) = \text{set } xs\ \cup\ \text{set } []$ 
3.  $\forall va. \text{set}(\text{merge } []\ (v\ \# \text{va})) = \text{set } []\ \cup\ \text{set } (v\ \# \text{va})$$ 
```

Again, see *Defining Recursive Functions in Isabelle/HOL*. Each induction hypothesis can only be used if the corresponding condition is provable.

Proof Outline

```
set (merge (x#xs) (y#ys)) = set (x # xs) U set (y # ys)
  set (if x ≤ y then x # merge xs (y#ys)
        else y # merge (x#xs) ys) = ...
  =
  (x ≤ y → set(x # merge xs (y#ys)) = ...) &
  (¬ x ≤ y → set(y # merge (x#xs) ys) = ...)
  =
  (x ≤ y → {x} U set(merge xs (y#ys)) = ...) &
  (¬ x ≤ y → {y} U set(merge (x#xs) ys) = ...)
  =
  (x ≤ y → {x} U set xs U set (y # ys) = ...) &
  (¬ x ≤ y → {y} U set (x # xs) U set ys = ...)
```

The first rewriting step in the proof unfolds the definition of merge. The second one is a case-split involving if. This step introduces a conjunction of implications, creating contexts that exactly match the induction hypotheses. But first, the definition of set (a function that maps a list to the finite set of its elements) must be unfolded. The last step highlighted above applies the induction hypotheses. The remaining steps, not shown, prove the equality between the set expressions just produced and the right-hand side of the original subgoal.

The Case Expression

- Similar to that found in the functional language ML.
- Automatically generated for every datatype.
- The simplifier can (upon request!) perform case-splits analogous to those for "if".
- Case splits in *assumptions* (not the conclusion) never happen unless requested.

Case-Splits for Lists

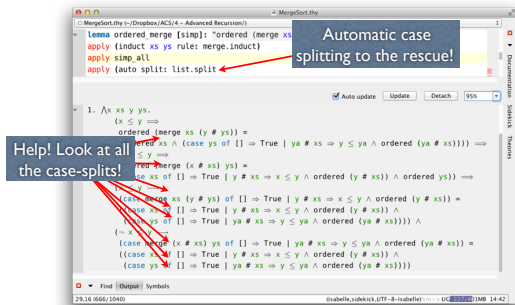
```
fun ordered :: "'a list => bool"
  where
    "ordered [] = True"
  | "ordered (x#l) =
    (case l of [] => True
     | Cons y xs => (x ≤ y & ordered (y#xs)))"
```

The definition shown on the slide describes the same function as the following one:

```
fun ordered :: "'a list => bool"
  where
    "ordered [] = True"
  | "ordered [x] = True"
  | "ordered (x#y#xs) = (x <= y & ordered (y#xs))"
```

The version adopted in this example works better in the simplifier. Two logically equivalent definitions may behave very differently with respect to formal proof.

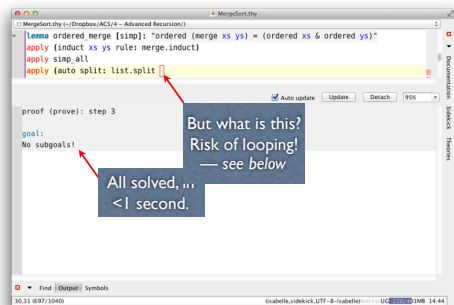
Case-Splitting in Action



There isn't room to show the full subgoal, but the second part of the conjunction (beginning with $\neg x \leq y$) has a similar form to the first part, which is visible above.

Note that the last step used was `simp_all`, rather than `auto`. The latter would break up the subgoal according to its logical structure, leaving us with 14 separate subgoals! Simplification, on the other hand, seldom generates multiple subgoals. The one common situation where this can happen is indeed with case splitting, but in our example, case splitting completely proves the theorem.

Completing the Proof



The identifier `ordered.simps` refers to the two equations that make up the definition of the function `ordered`. The suffix (2) selects the second of these. Now "`simp del: ordered.simps(2)`" tells `auto` to ignore this equation. Otherwise, the call will run forever.

Case Splitting for Lists

Simplification will replace

$$P(\text{case } xs \text{ of } [] \Rightarrow a \mid \text{Cons } x \ l \Rightarrow b \ x \ l)$$

by

$$(xs = [] \rightarrow P(a)) \wedge (\forall x \ l. xs = x \# \ l \rightarrow P(b \ x \ l))$$

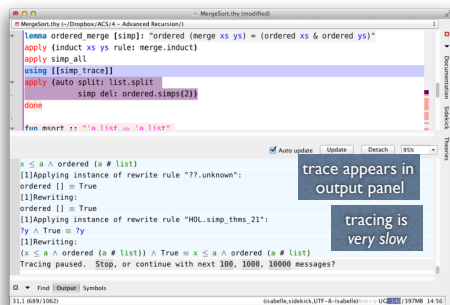
- It creates a case for each datatype constructor.
- Here it causes the simplifier to loop if combined with the second rewrite rule for `ordered`.

Specifically, a case split will create an instance where the list has the form `a#l`, and therefore `ordered(a#l)` will rewrite to another instance of case, *ad infinitum*.

Summary

- Many forms of recursion are available.
- You are given a specialised induction rule, which often leads to simple proofs.
- The “case” operator can often be dealt with using automatic case splitting...
- but complex simplifications can run forever!

How to Trace the Simplifier



```
Lemma ordered_merge [simp]: "ordered (merge xs ys) = (ordered xs & ordered ys)"
apply (induct xs ys rule: merge.induct)
apply simp_all
using [[simp_trace]]
apply (auto split: list.split
      simp del: ordered.simps())
done

fun merge :: "'a list * 'a list => 'a list"
  x <= a & ordered (a # list)
  [[Applying instance of rewrite rule "??,unknown":
  ordered [] = True
  [[Rewriting:
  ordered [] = True
  [[Applying instance of rewrite rule "HOL.simp_thm_21":
  ?y & True = ?y
  [[Rewriting:
  (? <= a & ordered (a # list)) & True = x <= a & ordered (a # list)
  Tracing paused. Stop, or continue with next 100, 1000, 10000 messages?

trace appears in
output panel
tracing is
very slow
```

An experimental new tracing panel now exists, invoked via Plugins > Isabelle > Simplifier Trace in conjunction with the declaration using `[[simp_trace_new mode=full]]`

This opens its own panel where the trace is displayed along with filtering options. Either way, simplifier tracing requires quite a bit of patience.

Interactive Formal Verification

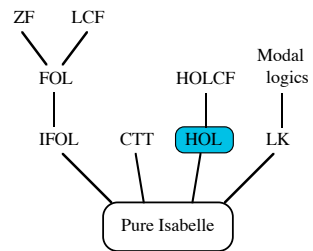
5: Logic in Isabelle

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Isabelle as a Logical Framework

- A formalism to represent other formalisms
- Support for *natural deduction*
- A common basis for **implementations**
- *Type theories* are commonly used, but Isabelle uses a simple meta-logic whose main primitives are
 - \Rightarrow (implication)
 - \wedge (universal quantification)

Isabelle's Family of Logics



Natural Deduction Basics

- Proof is done using mainly *inference rules* rather than axioms.
- For each logical symbol, there are rules to *introduce* and *eliminate* it.
- Assumptions can be *introduced* and *discharged*.
- Contrast with *Hilbert-style* proof systems, where typically the main inference rule is *modus ponens*...
- and there are many cryptic axioms, each combining a number of logical symbols.

Natural Deduction in Isabelle

$$\frac{P \quad Q}{P \wedge Q} \quad P \Rightarrow (Q \Rightarrow P \wedge Q)$$

$$\frac{P \wedge Q}{P} \quad P \wedge Q \Rightarrow P$$

$$\frac{P \wedge Q}{Q} \quad P \wedge Q \Rightarrow Q$$

$$\frac{P \rightarrow Q \quad P}{Q} \quad P \rightarrow Q \Rightarrow (P \Rightarrow Q)$$

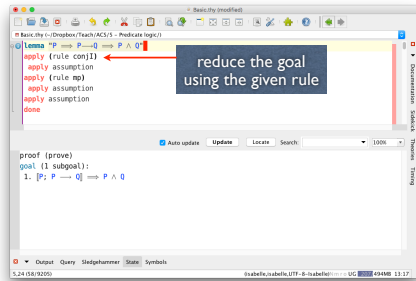
See the Tutorial, Chapter 5: The Rules of the Game. The first of these is an introduction rule, `conjI` in Isabelle. The following three are elimination rules: `conjunct1`, `conjunct2`, and `mp`. Isabelle parlance, these three are actually destruction rules because they lack the general form of an elimination rule in natural deduction.

Meta-implication

- The symbol \Rightarrow (or \implies) expresses the relationship between premise and conclusion
- ... and between subgoal and goal.
- It is distinct from \rightarrow , which is not part of Isabelle's underlying logical framework.
- $P \Rightarrow (Q \Rightarrow R)$ is abbreviated as $\llbracket P; Q \rrbracket \Rightarrow R$

The distinction between meta- and object-connectives is a common source of confusion among students. This distinction is inherent in the use of a logical framework. There is no reason why an object-logic would have an implication symbol at all. Isabelle gives a special significance to \Rightarrow , in particular for expressing the structure of inference rules, as shown on previous slide. This would be impossible if we make no distinction between \Rightarrow and \rightarrow .

A Trivial Proof

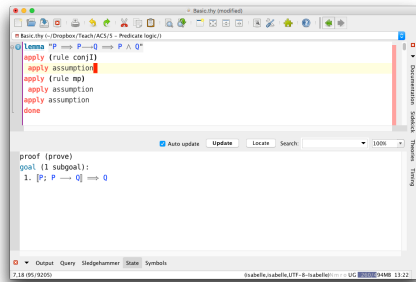


The method “rule” is fundamental. It matches the conclusion of the supplied rule with that of the a subgoal, which is replaced by new subgoals: the corresponding instances of the rule’s premises. See the Tutorial, 5.7 Interlude: the Basic Methods for Rules.

Normally, it applies to the first subgoal, though a specific goal number can be specified; many other proof methods follow the same convention.

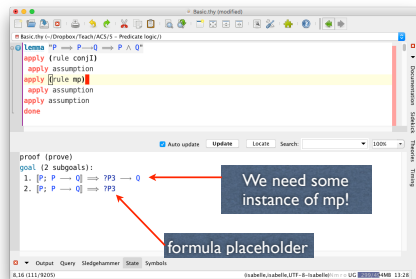
If automation doesn’t help you, then single step proof construction using the `rule` method and its variants (`drule`, `erule`) may be the best way forward.

Proof by Assumption



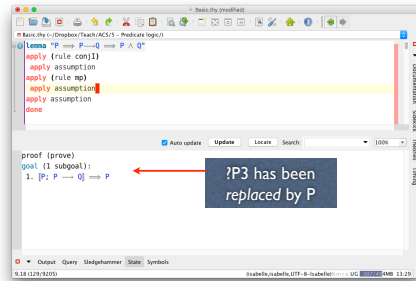
The method “assumption” proves (and deletes!) a subgoal if it can unify the subgoal’s conclusion with one of its premises.

Unknowns in Subgoals



Isabelle includes a class of variables whose names begin with the `?` character. They are called unknowns or schematic variables. Logically, they are no different from ordinary free variables, but Isabelle treats them differently: it allows them to be replaced by other expressions during unification. Isabelle rewrite rules and inference rules contain many such variables, but we normally suppress the question marks to make them easier to read. For example, the rule `conjI` is really $?P \implies (?Q \implies ?P \ \& \ ?Q)$.

Unknowns and Unification



Proving $P \rightarrow Q$ from the assumption $P \rightarrow Q$ performs unification, and the variable $?P3$ is updated. All occurrences of the variable are updated. In this way, proving one subgoal can make another subgoal impossible to prove. Sometimes there are multiple choices and only one will allow the proof to go through.

Discharging Assumptions

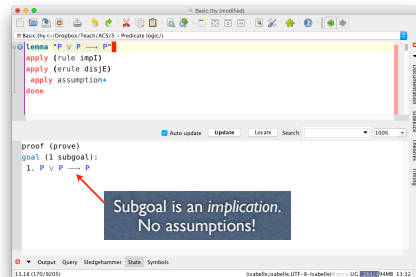
$$\frac{\begin{array}{c} [P] \\ \vdots \\ Q \end{array}}{P \rightarrow Q} \quad (P \Rightarrow Q) \Rightarrow P \rightarrow Q$$

$$\frac{\begin{array}{c} [P] \quad [Q] \\ \vdots \quad \vdots \\ P \vee Q \quad R \quad R \end{array}}{R} \quad \llbracket P \vee Q; P \Rightarrow R; Q \Rightarrow R \rrbracket \Rightarrow R$$

Such rules take derivations that depend upon particular assumptions (written as $[P]$ and $[Q]$ above) and “discharge” those assumptions, which means that the conclusion is not regarded as depending on them. The backwards interpretation is more natural: to prove $P \rightarrow Q$, it suffices to assume P and prove Q .

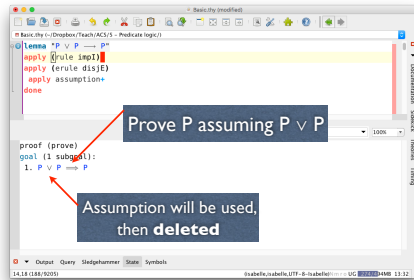
Meta-level implication (\Rightarrow) expresses the discharging of assumptions as well as the relationship between premises and conclusion.

A Proof using Assumptions

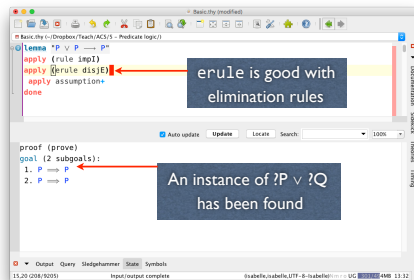


A full list of the predicate calculus rules for higher-order logic is available in Isabelle’s Logics: HOL, an old but still useful reference manual.

After Implies-Introduction

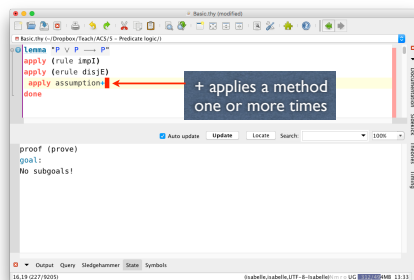


Disjunction Elimination



The point of the `erule` method is to apply inference rules (one at a time) where the first premise of the rule participates in the matching. There is also `drule`, which in effect performs forward reasoning, matching the first premise of a rule and adding the conclusion as a new assumption.

The Final Step



Quantifiers

$$\frac{P(t)}{\exists x. P(x)} \quad P(x) \Rightarrow \exists x. P(x)$$

$$\frac{\exists x. P(x) \quad \begin{array}{c} [P(x)] \\ \vdots \\ Q \end{array}}{Q} \quad [[\exists x. P(x); \wedge x. P(x) \Rightarrow Q]] \Rightarrow Q$$

meta-universal quantifier
states the variable condition

Isabelle's logical framework includes the typed lambda calculus, so quantifiers can be declared as constants of appropriate type. Variable-binding syntax can also be specified.

A Tiny Quantifier Proof

```

lemma "( $\exists x. P (f x) \wedge Q x \implies \exists x. P x$ )"
  apply (erule exE)
  apply (erule conjE)
  apply (rule exI)
  apply assumption
  done
  
```

proof (prove)
goal (1 subgoal):
1. $\exists x. P (f x) \wedge Q x \implies \exists x. P x$

conjE is an alternative to the conjunct1 and conjunct2. It has the standard elimination format (like disjE for disjunction elimination) so it can be used with the method erule.

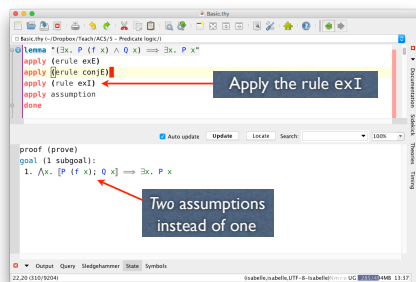
Conjunction Elimination

```

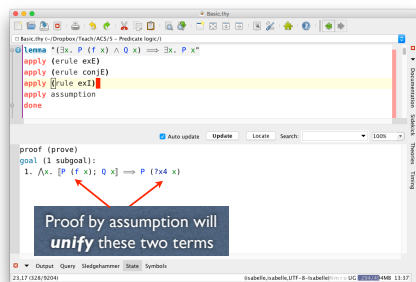
lemma "( $\exists x. P (f x) \wedge Q x \implies \exists x. P x$ )"
  apply (erule exE)
  apply (erule conjE)
  apply (rule exI)
  apply assumption
  done
  
```

proof (prove)
goal (1 subgoal):
1. $\exists x. P (f x) \wedge Q x \implies \exists x. P x$

Now for \exists -Introduction

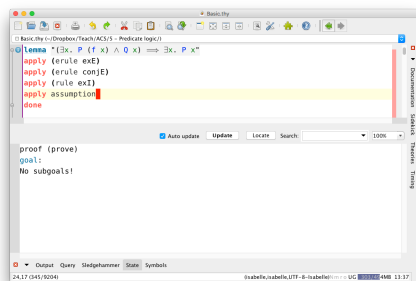


An Unknown for the Witness



A proof of existence normally requires a witness, namely a specific term satisfying the required property. Isabelle allows this choice to be deferred. The structure of the term, in this case `?x4 x`, holds information about which bound variables may appear in the witness. Here, `x` may appear in the witness.

Done!



Final Remarks

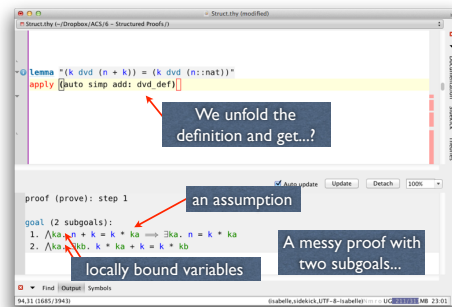
- There are analogous rules for universal quantifiers.
 - Higher-order unification respects bound variables, ensuring that quantifier reasoning is sound.
 - The examples above illustrate how things work, but typically most logical reasoning is automatic.
 - When automation fails, you can try single-step reasoning using methods such as `rule` and `erule`.
-

Interactive Formal Verification 6: Structured Proofs

Lawrence C Paulson
Computer Laboratory
University of Cambridge

A Proof about “Divides”

Note: $b \text{ dvd } a \Leftrightarrow (\exists k. a = b \times k)$



Capturing the Structure in Proofs

- Isabelle provides many tactics that refer to bound variables and assumptions.
- Assumptions are often found by matching.
- Bound variables can be referred to by name, but these names are fragile.
- *Structured proofs* provide a robust means of referring to these elements by name.
- Structured proofs are typically verbose but much more readable than linear `apply`-proofs.

The old-fashioned tactics mentioned above, have names like `rule_tac` and are described in the old *Tutorial*, particularly from section 5.7 onwards.

A Structured Proof

```
lemma "(k dvd (n + k)) = (k dvd (n::nat))"  
proof (simp only: dvd_def, safe)  
  fix m  
  assume "n + k = k * m"  
  then have "m = k * (n - 1)"  
  by (metis diff_add_inverse diff_mult_distrib2 nat_add_commute nat_mult_1_right)  
  then show "∃m'. n = k * m'"  
  by blast  
next  
fix m  
show "∃m'. k * n + k = k * m'"  
by (metis mult_suc_right nat_add_commute)  
qed
```

using this:
n + k = k * m
goal (1 subgoal):
1. n = k * (n - 1)

But how do you write them?

The Elements of Isar

- A *proof context* holds a local variables and assumptions of a subgoal.
- In a context, the variables are free and the assumptions are simply theorems.
- Closing a context yields a theorem having the structure of a subgoal.
- The Isar language lets us state and prove intermediate results, express inductions, etc.

Structured proofs can be tricky to write at first, and the *Tutorial* has little to say about them. This course now recommends more modern documentation, reserving the *Tutorial* as documentation for more old-fashioned but still useful commands. It also contained a great many extended examples.

Getting Started

```
lemma "(k dvd (n + k)) = (k dvd (n::nat))"  
proof (simp only: dvd_def, safe) []  
  
proof (state: step 1)  
goal (2 subgoals):  
1.  $\lambda k a. n + k = k * ka \implies \exists ka. n = k * ka$   
2.  $\lambda k a. \exists kb. k * ka + k = k * kb$ 
```

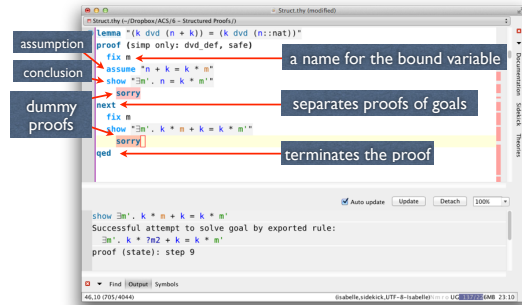
indicates the start of a structured proof (we avoid auto)

The simplest way to get started is as shown: applying auto with any necessary definitions. The resulting output will then dictate the structure of the final proof.

This style is actually rather fragile. Potentially, a change to auto could alter its output, causing a proof based on this precise output to fail. There are two ways of reducing this risk. One is to use a proof method less general than auto to unfold the definition of the divides relation and to perform basic logical reasoning. The other is to encapsulate the proofs of the two subgoals as local lemmas that can be passed to auto; this approach is more robust as it does not depend on the exact output.

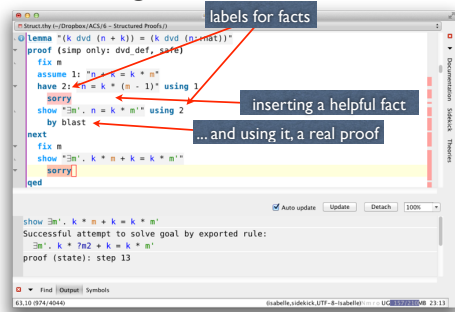
Replacing “auto” by “simp only: dvd_def, safe” produces a more robust proof, since these methods are much simpler and more stable than auto.

The Proof Skeleton



We have used sorry to omit the proofs. These dummy proofs allow us to construct the outer shell and confirm that it fits together. We use show to state (and eventually prove for real!) the subgoal's conclusion. Since we have renamed the bound variable ka to m, we must rename it in the assumption and conclusions. The context that we create with fix/assume, together with the conclusion that we state with show, must agree with the original subgoal. Otherwise, Isabelle will generate an error message, "Local statement will fail to refine any pending goal".

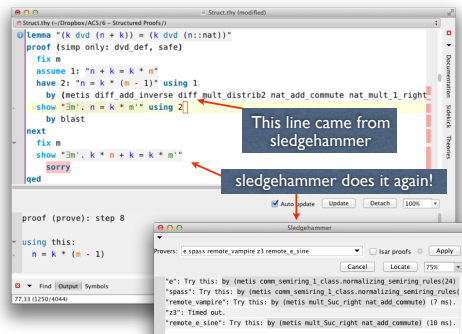
Fleshing Out that Skeleton



Looking at the first subgoal, we see that it would help to transform the assumption to resemble the body of the quantified formula that is the conclusion. Proving that conclusion should then be trivial, because the existential witness ($m-1$) is explicit. We use sorry to obtain this intermediate result, then confirm that the conclusion is provable from it using blast. Because it is a one line proof, we write it using "by". It is permissible to insert a string of "apply" commands followed by "done", but that looks ugly. The beauty of Isar is that it provides a continuum between fully structured proofs and fully linear apply-style reasoning.

We give labels to the assumption and the intermediate result for easy reference. We can then write "using 1", for example, to indicate that the proof refers to the designated fact. However, referring to the previous result is extremely common, and soon we shall streamline this proof to eliminate the labels. Also, labels do not have to be integers: they can be any Isabelle identifiers.

Completing the Proof



We have narrowed the gaps, and now sledgehammer can fill them. Replacing the last "sorry" completes the proof.

There is of course no need to follow this sort of top-down development. It is one approach that is particularly simple for beginners.

Streamlining the Proof

```

assume 1: "n + k = k * m"
have 2: "n = k * (m - 1)" using 1
sorry
show "∃m'. n = k * m'" using 2

```

→

```

assume "n + k = k * m"
then have "n = k * (m - 1)"
sorry
then show "∃m'. n = k * m'"

```

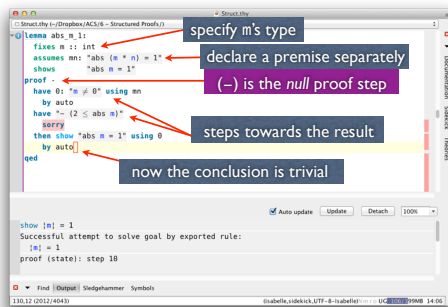
using the previous fact without mentioning labels

- then have or hence
- then show or thus

There are numerous other tricks of this sort!

Avoiding the contracted forms “hence” and “thus” may be better for readability, emphasising the role of “then”, which uses the previous fact.

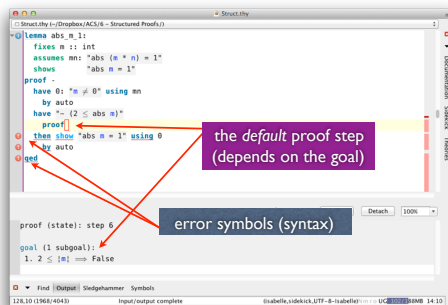
Another Proof Skeleton



This is an example of an obvious fact whose proof is not obvious. Clearly $m \neq 0$, since otherwise $m \cdot n = 0$. If we can also show that $|m| \geq 2$ is impossible, then the only remaining possibility is $|m| = 1$.

In this example, auto can do nothing. No proof steps are obvious from the problem’s syntax. So the Isar proof begins with “-”, the null proof. This step does nothing but insert any “pending facts” from a previous step (here, there aren’t any) into the proof state. It is quite common to begin with “proof -”.

Starting a Nested Proof



To begin with “proof” (not to be confused with “proof -”) applies a default proof method. In theory, this method should be appropriate for the problem, but in practice, it is often unhelpful. The default method is determined by elementary syntactic criteria. For example, the formula “ $\neg (2 \leq \text{abs } m)$ ” begins with a negation sign, so the default method applies the corresponding logical inference: it reduces the problem to proving False under the assumption $2 \leq \text{abs } m$.

A Nested Proof Skeleton

```

lemma abs_m_1:
  fixes m n: int
  assumes mn: "abs (m * n) = 1"
  shows "abs m = 1"
proof -
  have 0: "m ≠ 0" using mn
  by auto
  have "2 ≤ abs m"
  proof
    assume "2 ≤ abs m"
    then show "False"
    sorry
  qed
  then show "abs m = 1" using 0
  by auto
qed
  
```

Proofs can be nested to any depth. The assumptions and conclusions of each nested proof are independent of one another. The usual scoping rules apply, and in particular the facts mn and 0 are visible within this inner scope.

A Complete Proof

```

lemma abs_m_1:
  fixes m n: int
  assumes mn: "abs (m * n) = 1"
  shows "abs m = 1"
proof -
  have 0: "m ≠ 0" "n ≠ 0" using mn
  by auto
  have "2 ≤ abs m"
  proof
    assume "2 ≤ abs m"
    then have "2 * abs n ≤ abs m * abs n"
    by (simp add: mult mono)
    then have "2 * abs n ≤ abs (m*n)"
    by (simp add: abs_mult)
    then have "2 * abs n ≤ 1"
    by (auto simp add: mn)
    then show "False" using 0
    by auto
  qed
  then show "abs m = 1" using 0
  by auto
qed
  
```

This example is typical of a structured proof. From the assumption, $2 \leq \text{abs } m$, we deduce a chain of consequences that become absurd. We connect one step to the next using “hence”, except that we must introduce the conclusion using “thus”.

Note that we have beefed up the fact “0” from simply $m \neq 0$ to include as well $n \neq 0$, which we need to obtain a contradiction from $2 \times \text{abs } n \leq 1$. In fact, “0” here denotes a list of facts.

Calculational Proofs

```

proof (prove) step 15
goal (1 subgoal):
1. |m * n| = 1
  
```

The chain of reasoning in the previous proof holds by transitivity, and in normal mathematical discourse would be written as a chain of inequalities and equalities. Isar supports this notation.

The Next Step

```

assume "2 ≤ abs n"
then have "2 * abs n ≤ abs n * abs n"
  by (simp add: mult_mono 0)
also have "... = abs (n*n)"
  by (simp add: abs_mult)
also have "... = 1"
  by (simp add: mn)
finally have "2 * abs n ≤ 1" .
then show "False" using 0

```

proof (prove) step 12

goal (1 subgoal):

- $|n| * |n| = |n * n|$

... refers to the previous right-hand side

The Internal Calculation

```

assume "2 ≤ abs n"
then have "2 * abs n ≤ abs n * abs n"
  by (simp add: mult_mono 0)
also have "... = abs (n*n)"
  by (simp add: abs_mult)
also have "... = 1"
  by (simp add: mn)
finally have "2 * abs n ≤ 1" .
then show "False" using 0

```

structure of a calculation

calculation: $2 * |n| \leq 1$

proof (chain) step 17

picking this:

 $2 * |n| \leq 1$

The calculation is displayed for also and finally

Use “also” to attach a new link to the chain, extending the calculation. Use “finally” to refer to the calculation itself. It is usual for the proof script merely to repeat explicitly what this calculation should be, as shown above. If this is done, the proof is trivial and is written in Isar as a single dot (.).

We could instead avoid that repetition and reach the contradiction directly as follows:

```

also have "... = 1"
  by (simp add: mn)
finally show "False" using 0
  by auto

```

Internally, this proof is identical to the previous one. It merely differs in appearance, not bothering to note that $2 \times \text{abs } n \leq 1$ has been derived.

Ending the Calculation

```

assume "2 ≤ abs n"
then have "2 * abs n ≤ abs n * abs n"
  by (simp add: mult_mono 0)
also have "... = abs (n*n)"
  by (simp add: abs_mult)
also have "... = 1"
  by (simp add: mn)
finally have "2 * abs n ≤ 1" .
then show "False" using 0

```

(.) denotes a trivial proof

have $2 * |n| \leq 1$

proof (state) step 19

this:

 $2 * |n| \leq 1$

We have deduced $2 \times \text{abs } n \leq 1$

Structure of a Calculation

- The first line begins with *have* or *hence*
 - Subsequent lines begin with
 also have “... = “
 - *Any* transitive relation may be used. New ones may be declared.
 - The concluding line begins with
 finally have or *show*
 - It may *restate* the final result, terminating with (.)
-

Interactive Formal Verification

7: Sets

Lawrence C Paulson
Computer Laboratory
University of Cambridge

See the *Tutorial*, section 6.1 Sets.

Set Notation in Isabelle

- Set notation is crucial to mathematical discourse.
 - *Basics*: Union, intersection, power set
 - *Functions*: image, inverse image, composition
 - *Relations*: transitive closure, image
 - *Descriptions*: picking elements of sets by their properties
- A set in higher-order logic is similar to a *boolean-valued map*: in other words, to a logical predicate.
- The elements of a set must all have the *same type*!

Set Theory Primitives

- The membership relation: \in
- The subset relation: \subseteq (reflexive)
- Set comprehensions
- The empty set: $\{\}$
- The universal set: UNIV

Type α set is like $\alpha \Rightarrow \text{bool}$, but sets are not functions!

Basic Set Theory Equivalences

$$\begin{aligned}
 e \in \{x. P(x)\} &\iff P(e) \\
 e \in \{x \in A. P(x)\} &\iff e \in A \wedge P(e) \\
 e \in -A &\iff e \notin A \\
 e \in A \cup B &\iff e \in A \vee e \in B \\
 e \in A \cap B &\iff e \in A \wedge e \in B \\
 e \in \text{Pow}(A) &\iff e \subseteq A
 \end{aligned}$$

Please note that we do not write $\{x|P(x)\}$. Isabelle would interpret the $|$ as expressing disjunction and the expression as denoting the singleton set containing the element $x|P(x)$!

The logical equivalences shown above are effectively the definitions of the primitives shown, and any occurrences of the left-hand side formula will be replaced by the right-hand side by Isabelle's simplifier.

Big Union and Intersection

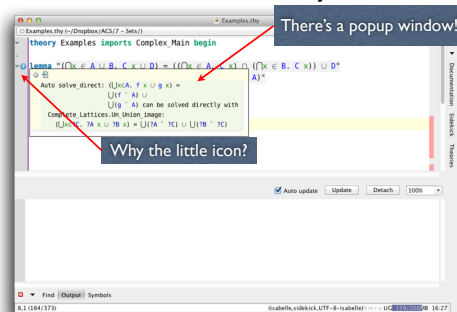
$$\begin{aligned}
 e \in \left(\bigcup x. B(x)\right) &\iff \exists x. e \in B(x) \\
 e \in \left(\bigcup x \in A. B(x)\right) &\iff \exists x \in A. e \in B(x) \\
 e \in \bigcup A &\iff \exists x \in A. e \in x
 \end{aligned}$$

And the analogous forms of intersections...

Once again, the logical equivalences are essentially definitions.

The third form of union is seldom seen, but fundamental.

A Trivial Set Theory Proof



Special symbols can be inserted using the Symbols panel. ASCII can simply be typed; auto-completions for symbols will be offered. More advanced users can type the names of these symbols preceded by a \backslash , if you know what the names are (typically the same as in LaTeX).

The main point of this example is that many such proofs are trivial, using auto or other automatic proof methods.

Also: look for icons in the left-hand "gutter", since they indicate errors, warnings or information.

Functions

$$e \in (f^i A) \iff \exists x \in A. e = f(x)$$

$$e \in (f^{-i} A) \iff f(e) \in A$$

$$f(x:=y) = (\lambda z. \text{if } z = x \text{ then } y \text{ else } f(z))$$

- Also inj, surj, bij, inv, etc. (injective,...)
- Don't re-invent image and inverse image!!

Inverse image is also known as pre-image. Using the actual image primitives gives access to the many theorems proved about them.

Finite Set Notation

$$\{a_1, \dots, a_n\} = \text{insert } a_1 \ (\dots (\text{insert } a_n \ \{\}) \dots)$$

where

$$x \in \text{insert } a \ B \iff x = a \vee x \in B$$

Finite sets can be written explicitly, enumerating their elements in the obvious way.

Finite Sets

A finite set is defined *inductively* in terms of **{}** and **insert**

$$\text{finite}(A \cup B) = (\text{finite } A \wedge \text{finite } B)$$

$$\text{finite } A \implies \text{card}(\text{Pow } A) = 2^{\text{card } A}$$

Defining functions over finite sets is tricky, because your definition has to make sense regardless of the order of the elements and regardless of whether they are repeated or not, because the sets $\{x,y\}$, $\{y,x\}$ and $\{x,y,x\}$ are all equal. The notion of cardinality is built-in.

The right way to define such functions is through special “fold” primitives, which give the desired result provided it’s meaningful. They are analogous to the well-known fold functionals for lists.

Intervals, Sums and Products

```

{..<u} == {x. x < u}
{..u} == {x. x ≤ u}
{1<..} == {x. 1<x}
{1..} == {x. 1≤x}
{1<..<u} == {1<..} ∩ {..<u}
{1..<u} == {1..} ∩ {..<u}

sum f A and prod f A
 $\sum_{i \in I}. f$  and  $\prod_{i \in I}. f$ 

```

Isabelle provides syntax for bounded and unbounded intervals. These are *polymorphic*: they are defined over all types that admit an ordering, and in particular they are applicable to intervals over the natural numbers, integers, rationals or reals.

Sums and products of functions over sets can also be written.

Descriptions

- Min, Max: of a *finite nonempty* set on a suitably ordered type (semilattice)
- Inf, Sup, etc: lower/upper bounds; possibly *limits*, in the case of infinite sets
- Hilbert choice: picking an element satisfying a given predicate (not a set, actually).

SOME $x. f(x) = 0$

Reasoning about such operators can be tricky, so it's important to find the right lemmas. The key property of SOME is $P\ x \implies P\ (SOME\ x. P\ x)$, where the repetition of P (possibly a long formula) causes further complications.

A Harder Proof Involving Sets

```

lemma
  fixes c :: "real"
  shows "finite A  $\implies (\sum_{i \in A}. c * f\ i) = c * (\sum_{i \in A}. f\ i)"
  apply (induct A rule: finite_induct)
  apply auto
  apply (auto simp add: algebra_simps)
  done

proof (prove)
  goal (1 subgoal):
  1. finite A  $\implies (\sum_{i \in A}. c * f\ i) = c * \text{sum } f\ A$$ 
```

This example needs a type constraint because arithmetic concepts such as sum and product are heavily overloaded. A readable way to insert a type constraint is using the `fixes` keyword. However, if you use `fixes`, then you must also use `shows!`

Isabelle's type classes allow this theorem to be proved in an overloaded form, but for simplicity here we restrict ourselves to type `real`.

Outcome of the Induction

```

Lemma
  fixes c :: "real"
  shows "finite A  $\implies \sum_{i \in A}. c * f i = c * \sum_{i \in A}. f i"$ 
  apply (induct A rule: finite_induct)
  apply auto
  apply (auto simp add: algebra_simps)
  done

proof (prove)
goal (2 subgoals):
1.  $\sum_{i \in \{\}}. c * f i = c * \text{sum } f \{\}$ 
2.  $\forall x F. [\text{finite } F; x \notin F; \sum_{i \in F}. c * f i = c * \text{sum } f F]$ 
 $\implies \sum_{i \in \text{insert } x F}. c * f i = c * \text{sum } f (\text{insert } x F)$ 

```

The base case is trivial, because both sides of the equality clearly equal zero. In the induction step, the induction hypothesis (which concerns the set F) will be applicable, because

$$\text{setsum } f (\text{insert } a F) = f a + \text{setsum } f F$$

Note that Isabelle uses a fancy notation for summations, but only if the body of the summation is nontrivial.

Almost There!

```

Lemma
  fixes c :: "real"
  shows "finite A  $\implies \sum_{i \in A}. c * f i = c * \sum_{i \in A}. f i"$ 
  apply (induct A rule: finite_induct)
  apply auto
  apply (auto simp add: algebra_simps)
  done

proof (prove)
goal (1 subgoal):
1.  $\forall x F. [\text{finite } F; x \notin F; \sum_{i \in F}. c * f i = c * \text{sum } f F]$ 
 $\implies c * f x + c * \text{sum } f F = c * (f x + \text{sum } f F)$ 

```

need a distributive law!

Recall that algebra_simps is a list of simplification rules for multiplying out algebraic expressions.

Finished!

```

Lemma
  fixes c :: "real"
  shows "finite A  $\implies \sum_{i \in A}. c * f i = c * \sum_{i \in A}. f i"$ 
  apply (induct A rule: finite_induct)
  apply auto
  apply (auto simp add: algebra_simps)
  done

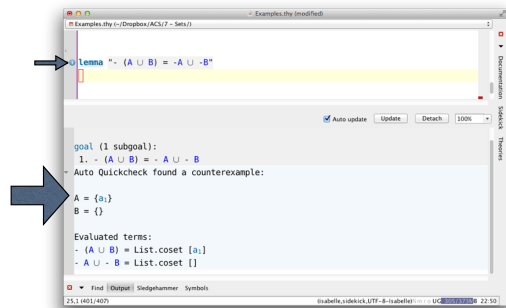
proof (prove)
goal:
No subgoals!

```

Counterexample Finding

- Don't waste time trying to prove false statements!
- Isabelle can find counterexamples quickly...
 - quickcheck: random testing of executable specifications (broadly interpreted)
 - nitpick: a general, SAT-based disprover
 - try: calls both of those (and sledgehammer)
- Type these commands right in the document.

Quickcheck Example



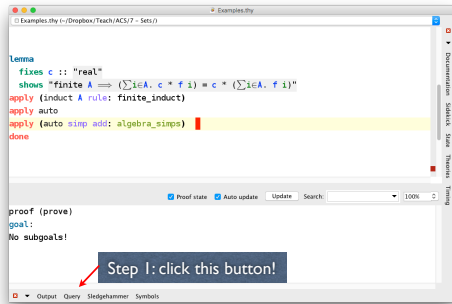
A minimal call to quickcheck is performed automatically. Auto nitpick and even auto sledgehammer can be configured in the plugin options.

They work especially well for functional programs, but work in other domains, as we see here.

Proving Theorems about Sets

- It is not practical to learn all the built-in lemmas.
- Instead, try an automatic proof method:
 - auto
 - force
 - blast
- Each uses the built-in library, comprising hundreds of facts, with powerful heuristics.

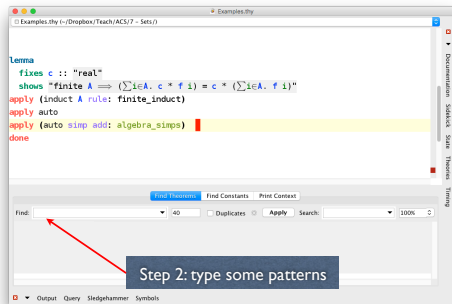
Finding Theorems about Sets



See the *Tutorial*, section **3.1.11 Finding Theorems**. Virtually all theorems loaded within Isabelle can be located using this function. Unfortunately, it does not locate theorems that are proved in external libraries.

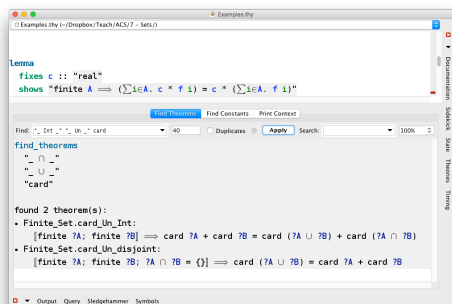
With some versions of Isabelle, the button is labelled “Query”.

Finding Theorems about Sets



The easiest way to refer to infix operators is by entering small patterns, as shown above. More complex patterns are also permitted. The constraints are treated conjunctively: use additional constraints if you get too many results, and fewer constraints if you get no results.

What Theorems Were Found?



The Query panel, like all the other panels, can be detached or docked in various places so that it is always available.

Interactive Formal Verification

8: Inductive Definitions

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Overview

- An introduction to inductive definitions
- Demonstrating their use with finite sets.
- Demonstrating more automation: the `arith` proof method and the *sledgehammer* proof tool.

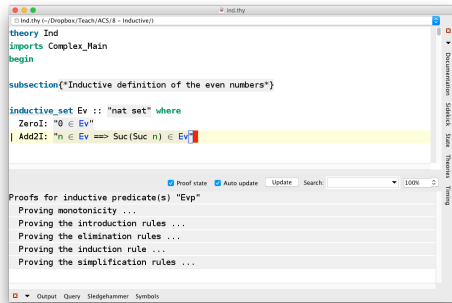
Defining a Set Inductively

- The set of even numbers is the least set such that
 - 0 is even.
 - If n is even, then $n+2$ is even.
- These can be viewed as *introduction rules*.
- We get an *induction principle* to express that no other numbers are even.
- Induction is used throughout mathematics, and to express *programming language semantics*.

See the *Tutorial*, Chapter 7. **Inductively Defined Sets.**

The *Tutorial* discusses precisely the same example, section 7.1.1.

Inductive Definitions in Isabelle



```
theory Ind
  imports Complex_Main
begin

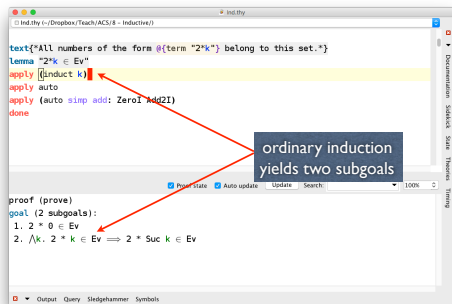
subsection("Inductive definition of the even numbers")

inductive_set Ev :: "nat set" where
  ZeroI: "0 ∈ Ev"
| Add2I: "n ∈ Ev ==> Suc(Suc n) ∈ Ev"

Proof state | Auto update | Update | Search | 100%
```

Proofs for inductive predicate(s) "Ev"
Proving monotonicity ...
Proving the introduction rules ...
Proving the elimination rules ...
Proving the induction rule ...
Proving the simplification rules ...

Even Numbers Belong to Ev

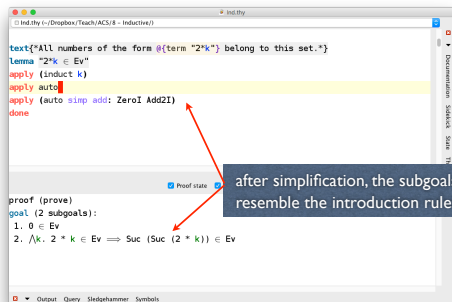


```
text("All numbers of the form @{term \"2*k\"} belong to this set.")
lemma "2*k ∈ Ev"
apply (induct k)
apply auto
apply (auto simp add: ZeroI Add2I)
done

proof (prove)
goal (2 subgoals):
1. 2 * 0 ∈ Ev
2.  $\forall k. 2 * k \in \text{Ev} \implies 2 * \text{Suc } k \in \text{Ev}$ 
```

ordinary induction yields two subgoals

Proving Set Membership



```
text("All numbers of the form @{term \"2*k\"} belong to this set.")
lemma "2*k ∈ Ev"
apply (induct k)
apply auto
apply (auto simp add: ZeroI Add2I)
done

proof (prove)
goal (2 subgoals):
1. 0 ∈ Ev
2.  $\forall k. 2 * k \in \text{Ev} \implies \text{Suc } (\text{Suc } (2 * k)) \in \text{Ev}$ 
```

after simplification, the subgoals resemble the introduction rules

Finishing the Proof

```

text("All numbers of the form 0(term "2*k") belong to this set.")
lemma "2*k ∈ Ev"
apply (induct k)
apply (auto intro: ZeroI Add2I)
done

proof (prove)
goal:
No subgoals!
  
```

The final version eliminates the duplicate “auto” step. In addition, here we see the use of “intro” as an alternative to “simp”. There’s more information on intro/elim/dest in the documentation under *classical reasoning*.

Rule Induction

- Proving a fact about every element of the set.
- It expresses that the inductive set is *minimal*.
- It is sometimes called “induction on derivations”
- There is a *base case* for every non-recursive introduction rule
- ...and an *inductive step* for the other rules.

Ev Has only Even Numbers

```

text("All elements of this set are even.")
lemma "n ∈ Ev ⇒ ∃k. n = 2*k"
apply (induct n rule: Ev.induct)
apply auto
apply arith
done

proof (prove): step 0
goal (1 subgoal):
1. n ∈ Ev ⇒ ∃k. n = 2 * k
  
```

The classic sign that we need rule induction is an occurrence of the inductive set as a premise of the desired result. Of course, sometimes the theorem can be proved by referring to other facts that have been previously proved using rule induction.

An Example of Rule Induction

```
text("All elements of this set are even.")
lemma "n ∈ Ev ⇒ ∃k. n = 2 * k"
apply (induct n rule: Ev.induct)
apply auto
apply arith
done

proof (prove) step 1
goal (2 subgoals):
1. ∃k. 0 = 2 * k
2. ∀n. [n ∈ Ev; ∃k. n = 2 * k] ⇒ ∃k. Suc (Suc n) = 2 * k
```

The auto method provides some support for arithmetic. However, complicated arithmetic arguments require specialised proof methods.

One Tricky Goal Left!

```
proof (prove) step 2
goal (1 subgoal):
1. ∀k. 2 * k ∈ Ev ⇒ ∃ka. Suc (Suc (2 * k)) = 2 * ka
```

The arith Proof Method

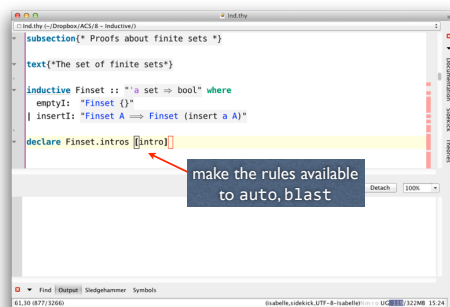
```
proof (prove) step 3
goal:
No subgoals!
```

Aside: Linear Arithmetic

- A *decidable* class of formulas
- auto can solve simple arithmetic problems...
- For the operators + - < ≤ =, and ...
- arith handles logical operators & quantifiers
- multiplication and division by *constants*: ×2, /2
- *Decision procedures are necessary* for proving arithmetic facts with reasonable effort.
- With variations, this class is decidable for the main arithmetic types.

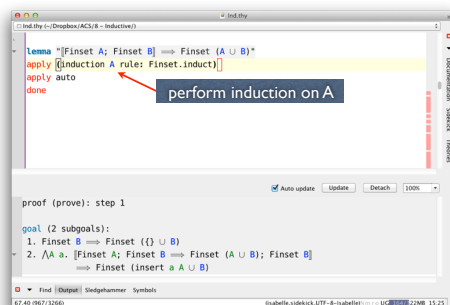
The empty set is finite. Adding one element to a finite set yields another finite set.

Defining Finiteness



```
subsection (* Proofs about finite sets *)
text{The set of finite sets*}
inductive Finset :: "α set ⇒ bool" where
  empty1: "Finset {}"
| insert: "Finset A ⇒ Finset (insert a A)"
declare Finset.intros [intros]
```

The Union of Two Finite Sets

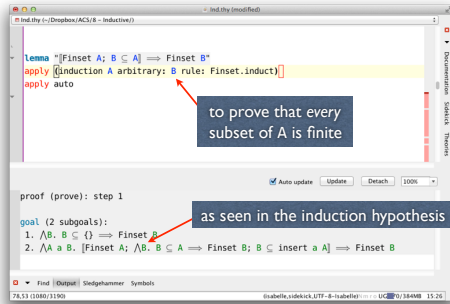


```
lemma "Finset A; Finset B ⇒ Finset (A ∪ B)"
  apply (induction A rule: Finset.induct)
  apply auto
  done

proof (prove): step 1
  goal (2 subgoals):
  1. Finset B ⇒ Finset ({} ∪ B)
  2.  $\forall a. \text{Finset } A; \text{Finset } B \Rightarrow \text{Finset } (A \cup B); \text{Finset } B \Rightarrow \text{Finset } (\text{insert } a \ A \cup B)$ 
```

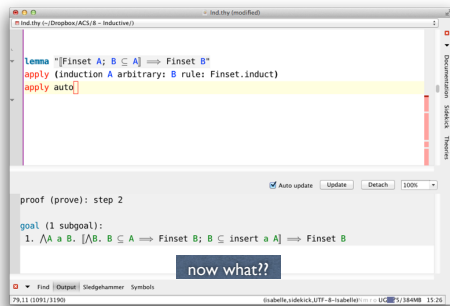
The goals are easily proved by the properties of sets and the introduction rules.

A Subset of a Finite Set



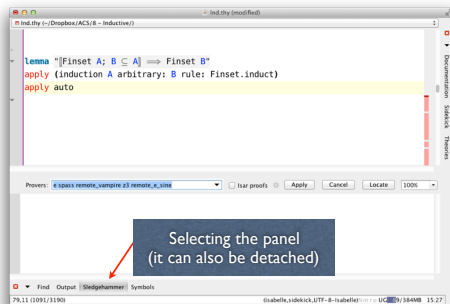
The proof is far more difficult than the preceding one, illustrating advanced techniques, in particular the sledgehammer tool.

A Critical Point in the Proof



None of Isabelle's automatic proof methods (`auto`, `blast`, `force`) have any effect on this subgoal. Informally, we might consider case analysis on whether $a \in B$. This would require using proof tactics that have not been covered. Fortunately, Isabelle provides a general automated tool, `sledgehammer`.

Time to Try Sledgehammer!



`Sledgehammer` calls several automated theorem provers in the background: in other words, Isabelle is still receptive to commands. You can continue to look for a proof manually.

Success!

```

lemma "[Finset A; B ⊆ A] ⇒ Finset B"
  apply (induction A arbitrary: B rule: Finset.induct)
  apply auto
  
```

Provers: spass remote_vampire z3 remote_e_sine

"z3": Try this: by (metis Finset.insertI insert_subset mk_disjoint_insert subset_insert)
"spass": Try this: by (metis (full types) Finset.insertI Set.set_insert insertII insert)
"e": Try this: by (metis (hide lams, no types) Finset.insertI dual_order.trans mk_dis)
"remote_vampire": Try this: by (metis Finset.simps Int.absorb2 Int.insert_right_iff I)
"remote_e_sine": Try this: by (metis Collect.comp Collect.empty_eq Collect_mem_eq Col)
To minimize: sledgehammer_min [remote_e_sine, provers = e spass remote_vampire z3 rem

All of outputs are highlighted. They are live: clicking on either will insert that command into the proof script.

The Completed Proof

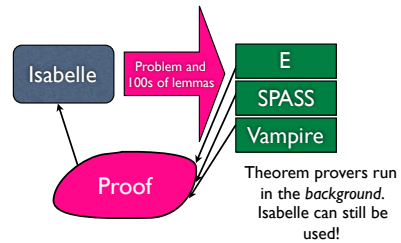
```

lemma "[Finset A; B ⊆ A] ⇒ Finset B"
  apply (induction A arbitrary: B rule: Finset.induct)
  apply auto
  apply (metis Finset.insertI insert_subset mk_disjoint_insert subset_insert)
  done
  
```

proof (prove): step 3

goal:
No subgoals!

How Sledgehammer Works



Notes on Sledgehammer

- First, simplify your goal and break it into pieces.
- It does not directly prove the goal, but returns a call to `metis`, `smt`, `blast`, `auto`, etc. This proof may be horrible, but could be the basis for a better proof.
- Read the manual: many options exist.
- If it doesn't help, think about intermediate properties that may be easier to prove.

Metis is an automatic theorem prover for first order logic, written by Joe Hurd. Sledgehammer calls high-performance theorem provers, such as E and Vampire, using them as relevance filters to select from the thousands of lemmas available in Isabelle. Isabelle problems are translated for these automatic theorem provers using lightweight translations, which do not preserve soundness. For that reason, proofs found by those theorem provers may be incorrect. If that happens, the call to `metis` will generate an error message or fail to terminate. It is possible to force the use of sound translations, but `sledgehammer` seldom finds proofs using those.

The proof returned by Sledgehammer may be ugly and messy. Consider tidying it up, especially for submitted work!

Interactive Formal Verification 9: Structured Induction Proofs

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Structured (Isar) Proofs

- As we've already seen:
 - Structured proofs are clearer than a series of commands, but verbose.
 - The Isar language is rich and complex, supporting a great many proof styles.
 - *But there's more!*
- *Existential* reasoning: naming entities that "exist".
- Syntax for proof by *induction*.
 - No need to write out induction hypotheses.
 - Cases given by *name*; bound variables named.
- And the same syntax works for *case analysis*.

A Proof about Binary Trees

```
datatype 'a bt =
  Lf
| Br 'a "'a bt" "'a bt"

fun reflect :: "'a bt => 'a bt" where
  "reflect Lf = Lf"
| "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect.reflect_ident: "reflect (reflect t) = t"
proof (induction t)
```

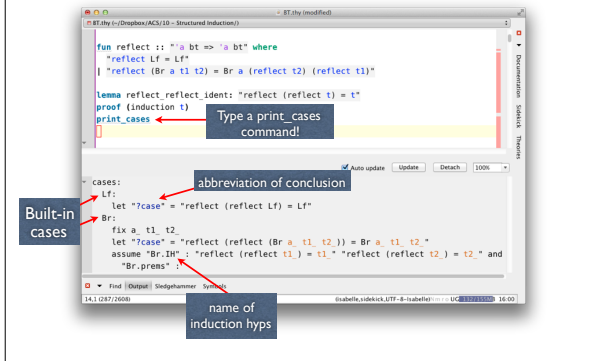
proof (state): step 1

goal (2 subgoals):

1. reflect (reflect Lf) = Lf
2. $\wedge a \ t1 \ t2.$
[reflect (reflect t1) = t1; reflect (reflect t2) = t2]
 \implies reflect (reflect (Br a t1 t2)) = Br a t1 t2

Inductive proofs frequently involve several subgoals, some of them with multiple assumptions and bound variables. Creating an Isar proof skeleton from scratch would be tiresome, and the resulting proof would be quite lengthy.

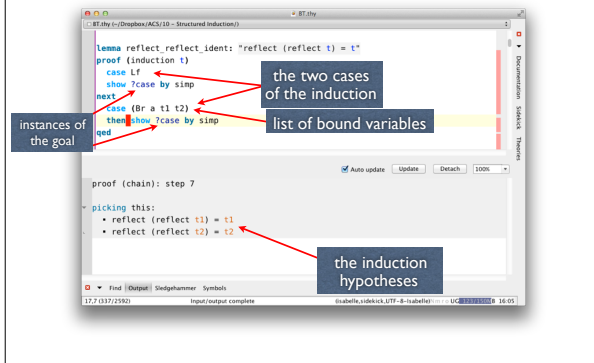
Finding Predefined Cases



Many induction rules have attached cases designed for use with Isar. By referring to such a case, a proof script implicitly introduces the contexts shown above. There are placeholders for the bound variables (specific names must be given). Identifiers are introduced to denote induction hypotheses and other premises that accompany each case. Also, the identifier `?case` is introduced to abbreviate the required instance of the induction formula.

It is possible to type the command `print_cases` right in your document. However, with the latest version of Isabelle, the proof keyword is highlighted: clicking on it inserts the proof skeleton automatically!

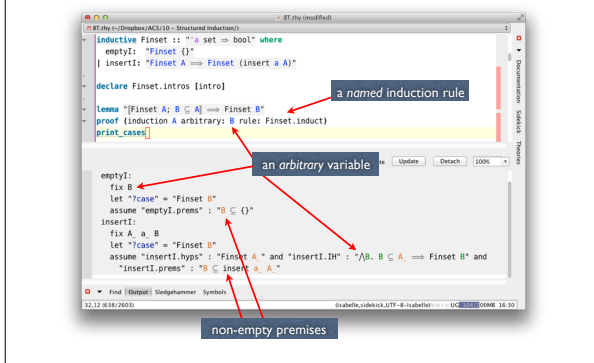
Proof Using Named Cases



With all these abbreviations, the induction formula does not have to be repeated in its various instances. The instances that are to be proved are abbreviated as `?case`; they (and the induction hypotheses) are automatically generated from the supplied list of bound variables.

Observe the use of "then" with "show" in the inductive case, thereby providing the induction hypotheses to the method. In a more complicated proof, these hypotheses can be denoted by the identifier `Br . hyps`.

Induction with a Context



An inductive definition generates an induction rule with one case (correspondingly named) for each introduction rule. This particular proof requires the variable `B` to be taken as arbitrary, which means, universally quantified: it becomes an additional bound variable in each case. This proof also carries along a further premise, $B \subseteq A$, instances of which are attached to both subgoals.

Proving the Base Case

```

inductive Finset : "'a set => bool" where
  empty: "Finset {}"
| insert: "Finset A => Finset (insert a A)"

declare Finset.intros [intro]
lemma "Finset A; B ⊆ A => Finset B"
proof (induction A arbitrary: B rule: Finset.induct)
  case (empty B)
  then show "Finset B"
  by auto
  
```

Annotations in the image:

- "arbitrary" variables can also be given names!
- "then" gives the premise to the next step

The base case would normally be just `emptyI`. But here, there is an additional bound variable. Note that we could have written, for example, `(emptyI C)` and Isabelle would have adjusted everything to use `C` instead of `B`.

A Nested Case Analysis

```

lemma "Finset A; B ⊆ A => Finset B"
proof (induction A arbitrary: B rule: Finset.induct)
  case (empty B)
  then show "Finset B"
  by auto
  next
  case (insertI A a B)
  show "Finset B"
  proof (cases "B ⊆ A")
  case True
  
```

Annotations in the image:

- "arbitrary" variables are (again) assigned names!
- Boolean case analysis on a formula

Here we know $B \subseteq \text{insert } a \text{ } A$, as it is the inherited premise of this case. But do we in fact know $B \subseteq A$?

The Complete Proof

```

  case True
  with insertI show "Finset B"
  by auto
  next
  case False
  have B: "B - {a} ⊆ A" using "B ⊆ insert a A"
  by auto
  with False have "B ⊆ insert a (B - {a})"
  by auto
  with B insertI.IH show "Finset B"
  by (metis Finset.insertI)
qed

```

Annotations in the image:

- reference to the induction hypothesis and premise
- true and false cases
- direct quotation of a fact, using cartouches
- reference to the false case: $B \subseteq A$

Here is an outline of the proof. If $B \subseteq A$, then it is trivial, as we can immediately use the induction hypothesis. If not, then we apply the induction hypothesis to the set $B - \{a\}$. We deduce that $B - \{a\} \in \text{Fin}$, and therefore $B = \text{insert } a \text{ } (B - \{a\}) \in \text{Fin}$.

This proof script contains many references to facts. The facts attached to the case of an inductive proof or case analysis are denoted by the name of that case, for example, `insertI`, `True` or `False`. We can also refer to a theorem by enclosing the actual theorem statement in backward quotation marks. We see this above in the proof of $B - \{a\} \in \text{Fin}$.

Additional Proof Structures

```

case (insertI A a B)
show "Finset B"
proof (cases "B ⊆ A")
case True
show "Finset B" using insertI True
by auto
next
case False
have Ba: "B - {a} ⊆ A" using <B ⊆ insert a A>
by auto
then have "B = insert a (B - {a})" using False
by auto
then show "Finset B"
by (metis Ba Finset.insertI insertI.IH)
qed

```

```

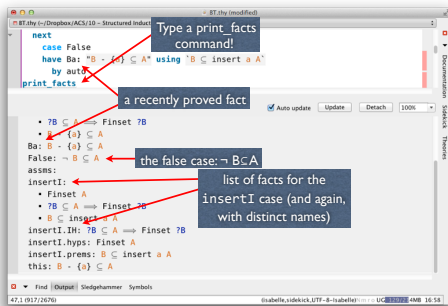
case (insertI A a B)
show "Finset B"
proof (cases "B ⊆ A")
case True
with insertI show "Finset B"
by auto
next
case False
from <B ⊆ insert a A> have Ba: "B - {a} ⊆ A"
by auto
with False have "B = insert a (B - {a})"
by auto
with Ba insertI.IH show "Finset B"
by (metis Finset.insertI)
qed

```

from <facts> ... = ... using <facts>
with <facts> ... = then from <facts> ...

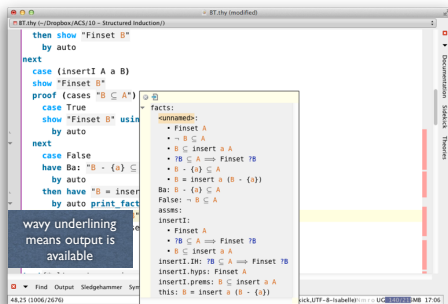
Full details, probably much more than you want at this stage, can be found in *The Isabelle/Isar Reference Manual*, by Makarius Wenzel.

Viewing Available Facts



It is unfortunately necessary to type the command `print_facts` right in your document.

Popups



Simply hover with the mouse over any text where you see wavy underlining.

Existential Claims: “obtain”

```

lemma dvd_trans:
  fixes a::nat
  assumes ab: "a dvd b" and bc: "b dvd c"
  shows "a dvd c"
proof -
  obtain v where "b = a * v"
  by (metis ab dvd_def)
  moreover obtain w where "c = b * w"
  by (metis bc dvd_def)
  ultimately have "c = a * (v * w)"
  by (simp add: mult_assoc)
  
```

proof (prove): step 2
 goal (1 subgoal):
 1. $\forall v. b = a * v \implies thesis \implies thesis$

$b \text{ dvd } a \leftrightarrow (\exists k. a = b \times k)$

Frequently, our reasoning involves quantities (such as j above) that are known to satisfy certain properties. Here, the “divides” premise implies the existence of a divisor, j . Proof attempts involving “obtain” can be difficult to understand, especially when they fail. Isabelle proves a theorem having the general form of an elimination rule, which in the premise introduces one or more bound variables: the variables that we “obtain”.

Chaining Facts: “moreover”

```

lemma dvd_trans:
  fixes a::nat
  assumes ab: "a dvd b" and bc: "b dvd c"
  shows "a dvd c"
proof -
  obtain v where "b = a * v"
  by (metis ab dvd_def)
  moreover obtain w where "c = b * w"
  by (metis bc dvd_def)
  ultimately have "c = a * (v * w)"
  by (simp add: mult_assoc)
  
```

calculation: $b = a * v$
 proof (state): step 4
 this:
 $b = a * v$

Delivering Facts: “ultimately”

```

lemma dvd_trans:
  fixes a::nat
  assumes ab: "a dvd b" and bc: "b dvd c"
  shows "a dvd c"
proof -
  obtain v where "b = a * v"
  by (metis ab dvd_def)
  moreover obtain w where "c = b * w"
  by (metis bc dvd_def)
  ultimately have "c = a * (v * w)"
  by (simp add: mult_assoc)
  
```

calculation:
 $b = a * v$
 $c = b * w$
 proof (chain): step 7
 picking this:
 $b = a * v$
 $c = b * w$

The Finished Proof

```

lemma dvd_trans:
  fixes a::nat
  assumes ab: "a dvd b" and bc: "b dvd c"
  shows "a dvd c"
proof -
  obtain v where "b = a * v"
  by (metis ab dvd_def)
  moreover obtain w where "c = b * w"
  by (metis bc dvd_def)
  ultimately have "c = a * (v * w)"
  by (simp add: mult_assoc)
  then show ?thesis
  by (rule dvdI)
qed
  
```

A Simpler Proof

```

lemma dvd_trans:
  fixes a::nat
  assumes "a dvd b" "b dvd c"
  shows "a dvd c"
proof -
  from assms obtain v where "b = a * v" "c = b * w"
  by (metis dvd_def)
  then have "c = a * (v * w)"
  by (simp add: mult_assoc)
  then show ?thesis ..
qed
  
```

Annotations:

- "assms" is the list of assumptions
- ".." is the default proof step (here, dvdI)
- can obtain multiple vars, facts in one step

Any proof can be written in a variety of different ways. The concluding step is surprising. The mysterious .. symbol denotes the default proof step, which in this case happens to be a rule called dvdI. This rule exactly matches the given premise and conclusion. In practice, however, default proof steps are seldom used.

moreover versus labels

```

notepad
begin

have l1: "fact1" sorry
have l2: "fact2" sorry
have l3: "fact3" sorry
from l1 l2 l3

oops

proof (chain): step 7

picking this:
• fact1
• fact2
• fact3
  
```

```

notepad
begin

have "fact1" sorry
moreover
have "fact2" sorry
moreover
have "fact3" sorry
ultimately

oops

calculation:
• fact1
• fact2
• fact3

proof (chain): step 9
  
```

These two keywords are useful when the conclusion is derived from a series of facts. The need for labels is eliminated (assuming that there are no other references to those facts) and the overall structure becomes much clearer. Here we also see the notepad construct, which is handy for typing in experimental proofs.

Cases via “consider”

```
fix i :: int
consider (even) "i mod 2 = 0" | (odd) "i mod 2 = 1"
  using not_mod_2_eq_0_1 by blast
then have "is_nice i"
proof cases
case even
then show ?thesis sorry
next
case odd
then show ?thesis sorry
qed

proof (prove)
using this:
  i mod 2 = 0
goal (1 subgoal):
  i. is_nice i
```

This even generalises “obtain”, as the separate cases can introduce bound variables. See the Isabelle/Isar reference manual.

Local Proof Blocks

```
notepad
begin
have "P ∨ Q ∨ R" sorry
moreover (assume "P" have "S" sorry)
moreover (assume "Q" have "S" sorry)
moreover (assume "R" have "S" sorry)
ultimately have "S" by blast
end

calculation:
• P ⇒ S
• Q ⇒ S
• R ⇒ S
proof (chain): step 21
```

Here we see a three-way case distinction. Local blocks have many other uses.

Interactive Formal Verification 10: Operational Semantics

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Overview

- The operational semantics of programming languages can be given *inductively*.
- Type checking
- Expression evaluation
- Command execution, including concurrency
- Properties of the semantics are frequently proved by induction.
- *Running example*: an abstract WHILE-language

Language Syntax

```
typedecl loc
(*an unspecified type of locations: addresses of variables*)

type_synonym val = nat (*values, here just natural numbers*)
type_synonym state = "loc => val"
type_synonym aexp = "state => val"
type_synonym bexp = "state => bool"
(* arithmetic and boolean expressions are not modelled explicitly here:
they are just functions on states *)

datatype
com = SKIP
  | Assign loc aexp (infix "==" 80)
  | Semi com com (infix ";" 70)
  | Cond bexp com com ("IF _ THEN _ ELSE _" [0, 90, 90] 91)
  | While bexp com ("WHILE _ DO _" [0, 91] 90)
```

For simplicity, this example does not specify arithmetic or boolean expressions in any detail. Although this approach is unrealistic, it allows us to illustrate key aspects of formalised proofs about programming language semantics.

A Big-Step Semantics

$$\begin{array}{c}
 \langle \text{skip}, s \rangle \rightarrow s \qquad \langle x := a, s \rangle \rightarrow s[x := a \ s] \\
 \\
 \frac{\langle c_0, s \rangle \rightarrow s'' \quad \langle c_1, s'' \rangle \rightarrow s'}{\langle c_0; c_1, s \rangle \rightarrow s'} \\
 \\
 \frac{b \ s \quad \langle c_0, s \rangle \rightarrow s' \quad \neg b \ s \quad \langle c_1, s \rangle \rightarrow s'}{\langle \text{if } b \ \text{then } c_0 \ \text{else } c_1, s \rangle \rightarrow s'} \\
 \\
 \frac{\neg b \ s}{\langle \text{while } b \ \text{do } c, s \rangle \rightarrow s} \quad \frac{b \ s \quad \langle c, s \rangle \rightarrow s'' \quad \langle \text{while } b \ \text{do } c, s'' \rangle \rightarrow s'}{\langle \text{while } b \ \text{do } c, s \rangle \rightarrow s'}
 \end{array}$$

In a big step semantics, the transition $\langle c, s \rangle \rightarrow s'$ means, executing the command c starting in the state s can terminate in state s' . Nothing is said about intermediate stages of the computation. Additionally

Formalised Language Semantics

```

inductive
  evalc :: "[com, state, state] => bool" ("[" _ ", _ ] / ~> _" [0, 0, 60] 60)
where
  Skip:    "[SKIP, s] ~> s"
  Assign:  "[x ::= a, s] ~> s[x := a s]"
  Semi:    "[c0, s] ~> s'' => [c1, s''] ~> s' => [c0; c1, s] ~> s'"
  IfTrue:  "b s => [c0, s] ~> s' => [IF b THEN c0 ELSE c1, s] ~> s'"
  IfFalse: "¬ b s => [c1, s] ~> s' => [IF b THEN c0 ELSE c1, s] ~> s'"
  WhileFalse: "¬ b s => [WHILE b DO c, s] ~> s"
  WhileTrue: "b s => [c, s] ~> s'' => [WHILE b DO c, s''] ~> s'"
           => [WHILE b DO c, s] ~> s'"
  Lemmas evalc.intros [intros] (*use those rules in automatic proofs*)
  
```

In the previous lecture, we used a related declaration, `inductive_set`. There is no intrinsic difference between a set and a one-argument predicate. However, formal semantics generally requires a predicate three or four arguments, and the corresponding set of triples is a little more difficult to work with. Attaching special syntax, as shown above, also requires the use of a predicate. Therefore, formalised semantic definitions will generally use `inductive`.

A Non-Termination Proof

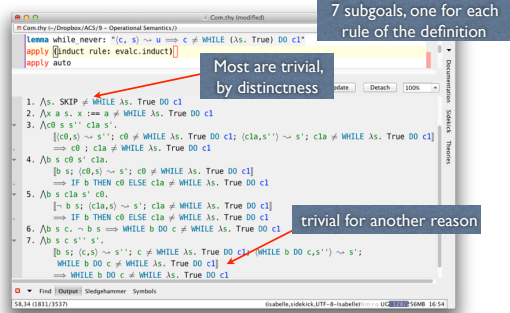
$$\langle \text{while true do } c, s \rangle \not\rightarrow s'$$

This formula is not provable by induction!

$$\langle c, s \rangle \rightarrow s' \Rightarrow \forall c'. c \neq (\text{while true do } c')$$

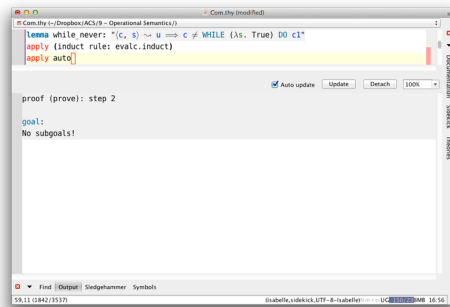
The inductive version considers all possible commands

A Formal Proof of Looping



This really is a trivial proof. I timed this call to auto and it needed only 6 ms.

Done!

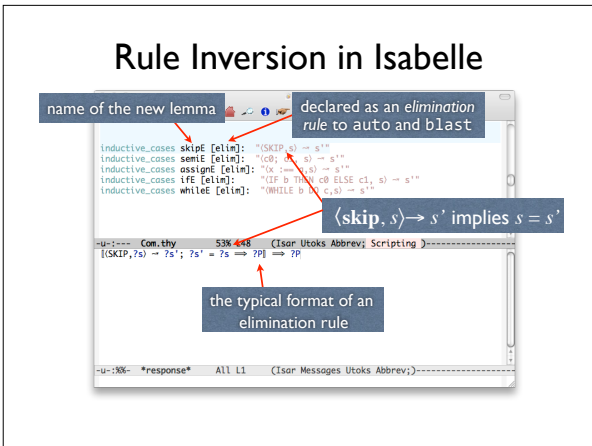


A New Principle: Rule Inversion

- When $\langle \text{skip}, s \rangle \rightarrow s'$ we know $s = s'$
- When $\langle \text{if } b \ \text{then } c_0 \ \text{else } c_1, s \rangle \rightarrow s'$ we know
 - b and $\langle c_0, s \rangle \rightarrow s'$, or...
 - $\neg b$ and $\langle c_1, s \rangle \rightarrow s'$
- This sort of case analysis is easy in Isabelle.

Rule inversion refers to case analysis on the form of the induction, matching the conclusions of the introduction rules (those making up the inductive definition) with a particular pattern. It is useful when only a small percentage of the introduction rules can match the pattern. This type of reasoning is common in informal proofs about operational semantics. It would not be useful in the inductive definitions covered in the previous lecture, where the conclusions of the rules had little structure.

Rule Inversion in Isabelle

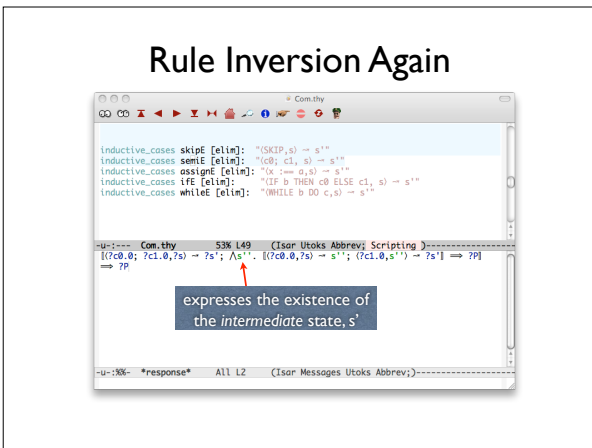


The pattern for each rule inversion lemma appears in quotation marks. Isabelle generates a theorem and gives it the name shown. Each theorem is also made available to Isabelle's automatic tools.

It is possible to write elim! rather than just elim; the exclamation mark tells Isabelle to apply the lemma aggressively. However, this must not be done with the theorem whileE: it expands an occurrence of

$\langle \text{while } b \text{ do } c, s \rangle \rightarrow s'$ and generates another formula of essentially the same form, thereby running for ever.

Rule Inversion Again



A detail: it's actually possible to use the slightly stronger elim! attribute in the first four of these. The difference between elim and elim! is that the former allows backtracking and the latter does not. Or in other words, the stronger version expresses the commitment that the specified theorem is the *only* way of proving the given conclusion.

Determinacy

$$\frac{\langle c, s \rangle \rightarrow t \quad \langle c, s \rangle \rightarrow u}{t = u}$$

If a command is executed in a given state, and it terminates, then this final state is *unique*.

Determinacy in Isabelle...

allow the other state to vary

trivial by rule inversion

```

theorem con_det: "c,s ~ t ~> (c,s) ~ u ~> u = t"
  apply (induct arbitrary: u rule: evalc.induct)
  apply blast
  
```

1. $\forall s\ u. (\text{SKIP } s) \rightsquigarrow u \implies u = s$
2. $\forall a\ s\ u. (a \text{ := } u\ s) \rightsquigarrow u \implies u = s (a := b\ s)$
3. $\forall c\ b\ s\ s'. (c\ b\ s) \rightsquigarrow s' \implies u = s (c := b\ s)$
 $\wedge u. (c_1\ s') \rightsquigarrow u \implies u = s'; (c\ b, c_1\ s) \rightsquigarrow u \implies u = s'$
4. $\forall b\ s\ c\ b\ s'. (c\ b\ s) \rightsquigarrow s'; \wedge u. (c\ b, s) \rightsquigarrow u \implies u = s'; (\text{IF } b \text{ THEN } c\ b \text{ ELSE } c_1\ s) \rightsquigarrow u \implies u = s'$
 $[\neg b\ s; (c_1\ s) \rightsquigarrow s'; \wedge u. (c_1, s) \rightsquigarrow u \implies u = s'; (\text{IF } b \text{ THEN } c\ b \text{ ELSE } c_1, s) \rightsquigarrow u]$
5. $\forall b\ s\ c\ u. [b\ s; (\text{WHILE } b \text{ DO } c\ s) \rightsquigarrow u] \implies u = s$
 $[\neg b\ s; (c_1\ s) \rightsquigarrow s'; \wedge u. (c_1, s) \rightsquigarrow u \implies u = s'; (\text{IF } b \text{ THEN } c\ b \text{ ELSE } c_1, s) \rightsquigarrow u]$
6. $\forall b\ s\ c\ u. [b\ s; (\text{WHILE } b \text{ DO } c\ s) \rightsquigarrow u] \implies u = s$
 $[\neg b\ s; (c_1\ s) \rightsquigarrow s'; \wedge u. (c_1, s) \rightsquigarrow u \implies u = s'; (\text{IF } b \text{ THEN } c\ b \text{ ELSE } c_1, s) \rightsquigarrow u]$
7. $\forall b\ s\ c\ s'. [b\ s; (\text{WHILE } b \text{ DO } c, s') \rightsquigarrow u] \implies u = s'$
 $[\neg b\ s; (c_1\ s) \rightsquigarrow s'; \wedge u. (c_1, s) \rightsquigarrow u \implies u = s'; (\text{IF } b \text{ THEN } c\ b \text{ ELSE } c_1, s) \rightsquigarrow u]$

The proof method `blast` uses introduction and elimination rules, combined with powerful search heuristics. It will not terminate until it has solved the goal. Unlike `auto` and `force`, it does not perform simplification (rewriting) or arithmetic reasoning. These subgoals are mostly trivial: rule inversion, which we set up previously, expresses precisely what we need: that if the given commands have executed, then corresponding intermediate states have been reached. The induction hypothesis allow us to assume the determinacy of the sub-commands.

Proved by Rule Inversion

call blast multiple times (here auto is very slow)

```

proof (prove) step 2
goal:
No subgoals!
  
```

The proof involves a long, tedious and detailed series of rule inversions. Apart from its length, the proof is trivial. This proof needed only 32 ms.

Semantic Equivalence

We can even define the infix syntax

It is trivially shown to be an equivalence relation

```

text("Two commands are equivalent if they allow the same transitions.")
definition
  equiv_c :: "com => com => bool" (infix "~" 50)
where
  "c ~ c' = (forall s'. ((c, s) ~ s') = ((c', s) ~ s'))"
lemma equiv_refl:
  "c ~ c"
by (auto simp add: equiv_c_def)
lemma equiv_sym:
  "c1 ~ c2 ==> c2 ~ c1"
by (auto simp add: equiv_c_def)
lemma equiv_trans:
  "c1 ~ c2 ==> c2 ~ c3 ==> c1 ~ c3"
by (auto simp add: equiv_c_def)
  
```

The screenshot shows the definition of semantic equivalence `equiv_c` and the proof of its properties. The text "We can even define the infix syntax" points to the definition of `equiv_c`. The text "It is trivially shown to be an equivalence relation" points to the lemmas `equiv_refl`, `equiv_sym`, and `equiv_trans`.

More Semantic Equivalence!

```

lemma equiv_semi:
  "(c1 ~ c1' ==> c2 ~ c2' ==> (c1; c2) ~ (c1'; c2'))"
  by (force simp add: equiv_c_def)

lemma equiv_if:
  "(c1 ~ c1' ==> c2 ~ c2' ==> (IF b THEN c1 ELSE c2) ~ (IF b THEN c1' ELSE c2'))"
  by (force simp add: equiv_c_def)

proof (prove): step 0
goal (1 subgoal):
1. WHILE b DO c ~ IF b THEN (c ; WHILE b DO c) ELSE SKIP
  
```

congruence laws: constructors preserve equivalence

by: gives a one-line proof

The properties shown here establish that semantic equivalence is a congruence relation with respect to the command constructors `Semi` and `Cond`. The proofs are again trivial, providing we remember to unfold the definition of semantic equivalence, `equiv_c`. Proving the analogous congruence property for `While` is harder, requiring rule induction with an induction formula similar to that used for another proof about `While` earlier in this lecture.

Method `force` is similar to `auto`, but it is more aggressive and it will not terminate until it has proved the subgoal it was applied to. In these examples, `auto` will give up too easily.

And More!!

```

lemma unfold_while:
  "(WHILE b DO c) ~ (IF b THEN (c ; WHILE b DO c) ELSE SKIP)"
  by (force simp add: equiv_c_def)

lemma triv_if:
  "(IF b THEN c ELSE c) ~ c"
  by (auto simp add: equiv_c_def)
  
```

Somehow, `force` will not solve the second theorem. Sometimes you just have to try different approaches.

Note that a proof consisting of a single proof method can be written using the command `by`, which is more concise than writing `apply` followed by `done`. It is a small matter here, but structured proofs (which we are about to discuss) typically consist of numerous one line proofs expressed using `by`.

Intro-Rule for Equivalence

$$\frac{\langle c, s \rangle \rightarrow s' \iff \langle c', s \rangle \rightarrow s'}{c \sim c'} \quad s \text{ and } s' \text{ not free...}$$

```

lemma equivI [intro!]:
  "(!s s'. (c, s) ~ s' ==> (c', s) ~ s') ==> c ~ c'"
  by (auto simp add: equiv_c_def)

lemma commute_if:
  "(IF b1 THEN (IF b2 THEN c11 ELSE c12) ELSE c2) ~ (IF b2 THEN (IF b1 THEN c11 ELSE c2) ELSE (IF b1 THEN c12 ELSE c2))"
  by blast
  
```

declared like this

formalised like this

used implicitly in blast/auto

Giving the attribute `intro!` to a theorem informs Isabelle's automatic proof methods, including `auto`, `force` and `blast`, that this theorem should be used as an introduction rule. In other words, it should be used in backward-chaining mode: the conclusion of the rule is unified with the subgoal, continuing the search from that rule's premises. It is now unnecessary to mention this theorem when calling those proof methods. The theorem shown can now be proved using `blast` alone. We do not need to refer to `equivI` or to the definition of `equiv_c`. The approach used to prove other examples of semantic equivalence in this lecture do not terminate on this problem in a reasonable time. The proof shown only requires 12 ms.

The exclamation mark (!) tells Isabelle to apply the rule aggressively. It is appropriate when the premise of the rule is equivalent to the conclusion; equivalently, it is appropriate when applying the rule can never be a mistake. The weaker attribute `intro` should be used for a theorem that is one of many different ways of proving its conclusion. Recall the discussion of `elim` versus `elim!` above.

Final Remarks on Semantics

- *Small-step semantics* can be treated similarly.
- *Variable binding* is crucial in larger examples, and should be formalised using the *nominal package*.
 - choosing a *fresh* variable
 - *renaming* bound variables consistently
- Serious proofs will be complex and difficult!

Documentation on the nominal package can be downloaded from <https://nms.kcl.ac.uk/christian.urban/Nominal/>

Many examples are distributed with Isabelle. See the directory HOL/Nominal/Examples.

Other relevant publications are available from Christian Urban's website: <https://nms.kcl.ac.uk/christian.urban/publications.html>

Interactive Formal Verification

//: Modelling Hardware

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Outline

- General modelling techniques
- Hardware verification in higher-order logic
- Additional elements of the Isar language, for instantiating theorems

Basic Principles of Modelling

- Define *mathematical abstractions* of the objects of interest (systems, hardware, protocols,...).
- Whenever possible, use *definitions* — not axioms!
- Ensure that the abstractions capture enough detail.
 - Unrealistic models have unrealistic properties.
 - Inconsistent models will satisfy *all* properties.

All models involving the real world are *approximate*!

Constructing models using definitions exclusively is called the *definitional approach*. A purely definitional theory is guaranteed to be consistent. Axioms are occasionally necessary in abstract models, where the behaviour is too complex to be captured by definitions. However, a system of axioms can easily be inconsistent, which means that they imply every theorem. The most famous example of an inconsistent theory is Frege's, which was refuted by Russell's paradox. A surprising number of Frege's constructions survived this catastrophe. Nevertheless, an inconsistent theory is almost worthless.

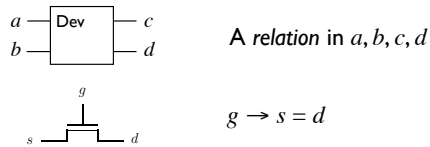
Useful models are abstract, eliminating unnecessary details in order to focus on the crucial points. The frictionless surfaces and pulleys found in elementary physics problems are a well-known example of abstraction. Needless to say, the real world is not frictionless and this particular model is useless for understanding everyday physics such as walking. But even models that introduce friction use abstractions, such as the assumption that the force of friction is linear, which cannot account for such phenomena as slipping on ice. Abstraction is always necessary in models of the real world, with its unimaginable complexity; it is often necessary even in a purely mathematical context if the subject material is complicated.

Hardware Verification

- Pioneered by Prof. M.J. C. Gordon and his students, using successive versions of the HOL system.
- Works *hierarchically* from arithmetic units and memories right down to flip-flops and transistors.
- Used to model complete hardware designs:
 - VIPER verification, by Avra Cohn (1988)
 - ARM6 processor, by Anthony Fox (2003)
- Crucially uses *higher-order logic*, modelling signals as boolean-valued functions over time.

The material in this lecture is based on Prof Gordon's lecture notes for *Specification and Verification II*, which are still available on the Internet at <http://www.cl.cam.ac.uk/~mjc/Lectures/SpecVer2/>

Devices as Relations

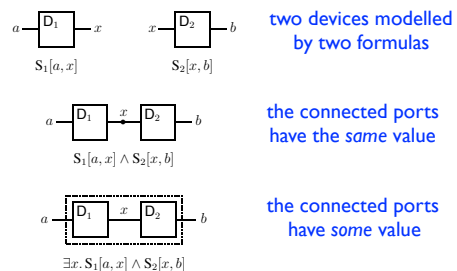


The relation describes the possible combinations of values on the ports.

Values could be bits, words, signals (functions from time to bits), etc

The second device on the slide above is an N-type field effect transistor, which can be conceived as a switch: when the gate goes high, the source and drain are connected. The logical implication shown next to the transistor formalises this behaviour. Note that the connection between the source and drain is *bidirectional*, with no suggestion that information flows from one port to the other.

Relational Composition



The diagrams are taken from Prof Gordon's lecture notes.

Because we model devices by relations, connecting devices together must be modelled by relational composition. Syntactically, we specify circuits by logical terms that denote relations and we express relational composition using the existential quantifier. The quantifier creates a local scope, thereby hiding the internal wire.

Specifications and Correctness

- The *implementation* of a device in terms of other devices can be expressed by composition.
- The *specification* of the device's intended behaviour can be given by an abstract formula.
- Sometimes the implementation and specification can be proved *equivalent*: $Imp \Leftrightarrow Spec$.
- The property $Imp \Rightarrow Spec$ ensures that every behaviour of the *Imp* is permitted by *Spec*.

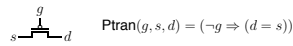
Impossible implementations satisfy all specifications!

The implementation describes a circuit, while the specification should be based on mathematical definitions that were established prior to the implementation. A limitation of this approach is that impossible implementations can be expressed: in the most extreme case, implementations that identify the values true and false. In hardware, this represents a short circuit connecting power to ground, possibly a short circuit that only occurs when a particular combination of values appears on other wires, activating an unfortunate series of transistors. In the real world, short circuits have catastrophic effects, while in logic, identifying true with false allows anything to be proved. Therefore, absence of short circuits needs to be established somehow if this relational approach is to be used safely.

For combinational circuits (those without time), both the implementation and the specification express truth tables with no concept of a "don't care" entry, so logical equivalence should be provable. Sequential circuits involve time, and frequently the specification samples the clock only a specific intervals, ignoring the situation otherwise. Specifications can involve many other forms of abstraction. In general, we cannot expect to prove logical equivalence.

Proving the logical equivalence of the implementation with the specification does not prove the absence of short circuits, but it does prove that the short circuits coincide with inconsistencies in the specification itself. Needless to say, a correct specification should be free of inconsistencies, but there is no way in general to guarantee this. How then do we benefit from using logic? Specifications

The Switch Model of CMOS



$$Ptran(g, s, d) = (\neg g \Rightarrow (d = s))$$



$$Ntran(g, s, d) = (g \Rightarrow (d = s))$$



$$Gnd\ g = (g = F)$$



$$Pwr\ p = (p = T)$$

subsection[* Specification of CMOS primitives *]

text[* P and N transistors *]

definition "Ptran = $\lambda(g, a, b). (\neg g \Rightarrow a = b)$ "

definition "Ntran = $\lambda(g, a, b). (g \Rightarrow a = b)$ "

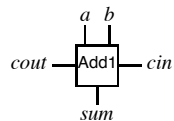
text[* Power and Ground*]

definition "Pwr p = (p = True)"

definition "Gnd p = (p = False)"

CMOS (complementary metal oxide semiconductor) technology combines P- and N-type transistors on a chip to make gates and other devices. The slide shows primitive concepts: the two types of transistors, ground (modelled by the value False) and power (model by the value True). The corresponding Isabelle definitions are easily expressed. Lambda-notation is a convenient way to express a function is argument is a triple.

Full Adder: Specification



$$2 \times cout + sum = a + b + cin$$

text[* 1-bit full adder specification *]

text[* Convert boolean to number (0 or 1) *]

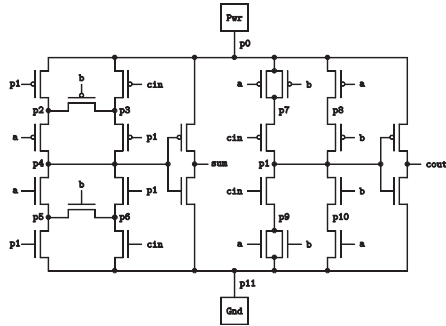
definition bit_val :: "bool \Rightarrow nat" where

"bit_val p = (if p then 1 else 0)"

definition "Add1Spec = $\lambda(a, b, cin, sum, cout). 2 * (bit_val\ cout) + bit_val\ sum = bit_val\ a + bit_val\ b + bit_val\ cin$ "

A full adder forms the sum of three one-bit inputs, yielding a two-bit result. The higher-order output bit is called "carry out", and it will typically be connected to the "carry in" of the next stage. Because we typically use True and False to designate hardware bit values, the obvious conversion to 1 and 0 is necessary in order to express arithmetic properties. Even with this small step, expressing the specification in higher-order logic is trivial. The identifier denotes the abstract relation satisfied by a full adder, namely the legal combinations of values on the various ports.

Full Adder: Implementation



A full adder is easily expressed at the gate level in terms of exclusive-OR (to compute the sum) and other simple gating to compute the carry. The diagram above, again from Prof Gordon's notes, expresses a full adder as would be implemented directly in terms of transistors.

Full Adder in Isabelle

```

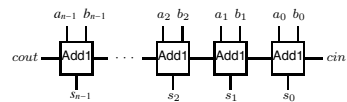
definition "AddImp = (λ(a,b,cin,s,cout).
  ∃(p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11.
    Ptran(p1,p0,p2) ∧ Ptran(cin,p0,p3) ∧
    Ptran(b,p2,p3) ∧ Ptran(a,p2,p4) ∧
    Ptran(p1,p3,p4) ∧ Ntran(a,p4,p5) ∧
    Ntran(p1,p4,p6) ∧ Ntran(b,p5,p6) ∧
    Ntran(p1,p5,p11) ∧ Ntran(cin,p6,p11) ∧
    Ptran(a,p6,p7) ∧ Ptran(b,p6,p7) ∧
    Ptran(a,p8,p8) ∧ Ptran(cin,p7,p1) ∧
    Ptran(b,p8,p11) ∧ Ntran(cin,p1,p9) ∧
    Ntran(b,p1,p10) ∧ Ntran(a,p9,p11) ∧
    Ntran(b,p9,p11) ∧ Ntran(a,p10,p11) ∧
    Pwr(p0) ∧ Ptran(p4,p8,s) ∧
    Ntran(p4,s,p11) ∧ Gnd(p11) ∧
    Ptran(p1,p0,cout) ∧ Ntran(p1,cout,p11)")"
  
```

$$(\exists b. P \ b) = (P \ \text{True} \vee P \ \text{False})$$

The logical formula above is a direct translation of the diagram on the previous slide. Needless to say, the translation from diagram to formula should ideally be automatic, and better still, driven by the same tools that fabricate the actual chip.

The theorem expresses the logical equivalence between the implementation (in terms of transistors) and the specification (in terms of arithmetic). This type of proof is trivial for reasoning tools based on BDDs or SAT solvers. Isabelle is not ideal for such proofs, and this one requires over four seconds of CPU time. In the simplifier call, the last theorem named is crucial, because it forces a case split on every existentially quantified wire.

An n -bit Ripple-Carry Adder



$$(2^n \times \text{cout}) + s = a + b + \text{cin}$$

- Cascading several full adders yields an n -bit adder.
- The implementation is expressed recursively.
- The specification is obvious mathematics.

Adder Specification

$$(2^n \times cout) + s = a + b + cin$$

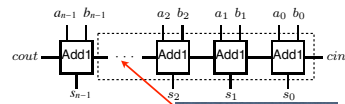
```

text(* Unsigned number denoted by bitstring f(n-1)...f(0) *)
fun bits_val where
  "bits_val f 0 = 0"
  | "bits_val f (Suc n) = 2^n * bit_val(f n) + bits_val f n"
text(* Specification of an n-bit adder *)
definition
  "AdderSpec n = (λ a b, c in, sum, cout).
  2^n * bit_val cout + bits_val sum n =
  bits_val a n + bits_val b n + bit_val cin)"
  
```

value of n-bit words

The function `bits_val` converts a binary numeral (supplied in the form of a boolean valued function, `f`) to a non-negative integer. The specification of the adder then follows the obvious arithmetic specification closely. When `n=0`, the specification merely requires `cin=cout`.

Adder Implementation



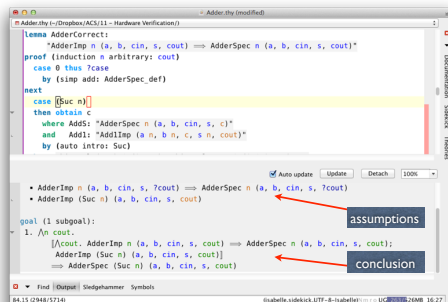
```

text(* Implementation of an n-bit ripple-carry adder*)
fun AdderImp where
  "AdderImp 0 a b, c in, sum, cout = (cout = cin)"
  | "AdderImp (Suc n) (a, b, c in, sum, cout) =
  (λ c in. AdderImp n (a, b, c in, sum, c) +
  AdderImp (a n, b n, c, sum n, cout))"
  
```

a zero-bit adder simply connects the carry lines!

An `(n+1)`-bit adder consists of a full adder connected to an `n`-bit adder. Note that `AdderImp n` specifies an `n`-bit adder, and in particular, a 0-bit adder is nothing but a wire connecting carry in to carry out.

Partial Correctness Proof



We are proving *partial correctness* only: that the implementation implies the specification. The term “partial correctness” here refers to a limitation of the approach, namely that an inconsistent implementation (one with short circuits) can imply any specification. Termination, obviously, plays no role in this circuit.

The base case is trivial. Our task in the induction step is shown on the slide. It is expressed in terms of predicates for the implementation and specification. The induction hypothesis asserts that the implementation implies the specification for `n`. We now assume the implementation for `n+1` and must prove the corresponding specification.

Using the Induction Hypothesis

```

Lemma AdderCorrect:
  "AdderImp n (a, b, cin, s, cout) ==> AdderSpec n (a, b, cin, s, cout)"
proof (induction n arbitrary: cout)
case 0 thus ?case
next
  case [Suc n]
  then obtain c
    where Add1: "Add1Imp (a n, b n, c, s n, cout)"
    and Add1Spec: "Add1Spec (a n, b n, c, s n, cout)"
  by (auto intro: Suc)
  
```

By assumption, we have `AdderImp (Suc n)` and therefore both `AdderImp n` and `Add1Imp`. The simplest use of “obtain” would derive those assumptions, but we can skip a step and go directly to `AdderSpec n` by referring to the induction hypothesis.

A Tiresome Calculation

```

then obtain c
  where Add1: "Add1Spec (a n, b n, c, s n, cout)"
  and Add1: "Add1Imp (a n, b n, c, s n, cout)"
  by (auto intro: Suc)
  have "bit_val (s n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
    (bit_val (a n) + (bit_val cout * 2)) * (2 ^ n)"
  by (simp add: algebra_simps)
  also have "... = (bit_val c + (bit_val (a n) + bit_val (b n))) *
    (2 ^ n)"
  using Add1 by (simp add: Add1Correct Add1Spec def)
  finally show "AdderSpec (Suc n) (a, b, cin, s, cout)" using Add1
  
```

This equation is suggested by earlier attempts to prove the induction step directly. The proof involves using the correctness of a full adder to replace `Add1Imp` by `Add1Spec`, then unfolding the latter to get the sum $c + a n + b n$. The precise form of the left-hand side has been chosen to match a term that will appear in the main proof. This kind of reasoning is tedious even with the help of Isar. Better support for arithmetic could make this proof almost automatic.

Partial Correctness is Proved!

```

text ("Partial correctness of ripple-carry adder for all n by induction *")
lemma AdderCorrect:
  "AdderImp n (a, b, cin, s, cout) ==> AdderSpec n (a, b, cin, s, cout)"
proof (induction n arbitrary: cout)
case 0 thus ?case
next
  case [Suc n]
  then obtain c
    where Add1: "Add1Spec (a n, b n, c, s n, cout)"
    and Add1: "Add1Imp (a n, b n, c, s n, cout)"
  by (auto intro: Suc)
  have "bit_val (s n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
    (bit_val (a n) + (bit_val cout * 2)) * (2 ^ n)"
  by (simp add: algebra_simps)
  also have "... = (bit_val c + (bit_val (a n) + bit_val (b n))) *
    (2 ^ n)"
  using Add1 by (simp add: Add1Correct Add1Spec def)
  finally show "AdderSpec (Suc n) (a, b, cin, s, cout)" using Add1
  qed
  
```

We end up with a fairly simple structure. Note that we could have used `Add1Correct` earlier in the proof, obtaining `Add1: "Add1Spec ..."` directly.

To repeat: we have proved that every possible configuration involving the connectors to our circuit satisfies the specification of an `n`-bit adder. Tools based on BDDs or SAT solvers can prove instances of this result for fixed values of `n`, but not in the general case.

Proving Equivalence

```

Adder.thy (modified)
Adder.thy (D:\Dropbox\ACS11 - Hardware Verification)
Lemma AdderSpec_Suc:
  "AdderSpec (Suc n) (a, b, cin, s, cout) =
   (lc. AdderSpec n (a, b, cin, s, c) & AddISpec (a, b, n, c, s, n, cout))"
apply (auto simp add: AdderSpec_def AddISpec_def ex_bool_eq bit_val_def)

goal [16 subgoals]:
1. [a m; b m; s m; ~ cout; cin; bits_val s n = Suc (2 ^ n + (bits_val a n + bits_val b n))]
   ==> False
2. [a m; b m; s m; ~ cout; ~ cin; bits_val s n = 2 ^ n + (bits_val a n + bits_val b n)]
   ==> False
3. [a m; b m; ~ s m; ~ cout; cin;
   bits_val s n = Suc (2 ^ n + bits_val a n + (2 ^ n + bits_val b n))]
   ==> False
4. [a m; b m; ~ s m; ~ cout; ~ cin;
   bits_val s n = 2 ^ n + bits_val a n + (2 ^ n + bits_val b n)]
   ==> False
5. [a m; ~ b m; s m; cout; cin;
   2 * 2 ^ n + bits_val s n = Suc (bits_val a n + bits_val b n)]
   ==> False
6. [a m; ~ b m; s m; cout; ~ cin; 2 * 2 ^ n + bits_val s n = bits_val a n + bits_val b n]

144.13.0542.17390
  
```

To prove that the specification implies the implementation would yield their exact equivalence. It would also guarantee the lack of short circuits in the implementation, as the specification is obviously correct.

The verification requires the lemma shown above, which resembles the recursive case of `AdderImp`. We might expect its proof to be straightforward. Unfortunately, the obvious proof attempt leaves us with 16 subgoals. A bit of thought informs us that these cases represent impossible combinations of bits. These arithmetic equations cannot hold. But how can we prove this theorem with reasonable effort?

A Crucial Lemma

```

Adder.thy (modified)
Adder.thy (D:\Dropbox\ACS11 - Hardware Verification)
Lemma bits_val_less: "bits_val f n < 2^n"
by (induction n, auto simp add: bit_val_def)

Lemma AdderSpec_Suc:
  "AdderSpec (Suc n) (a, b, cin, s, cout) =
   (lc. AdderSpec n (a, b, cin, s, c) & AddISpec (a, b, n, c, s, n, cout))"
using bits_val_less [of a n] bits_val_less [of b n] bits_val_less [of s n]
by (simp add: AdderSpec_def AddISpec_def ex_bool_eq bit_val_def)

Lemma AdderCorrect2:
  "AdderSpec n (a, b, cin, s, cout) ==> AdderImp n (a, b, cin, s, cout)"
using this:
  • bits_val a n < 2 ^ n
  • bits_val b n < 2 ^ n
  • bits_val s n < 2 ^ n
goal [1 subgoal]:
1. AdderSpec (Suc n) (a, b, cin, s, cout) ==> (lc. AdderSpec n (a, b, cin, s, c) & AddISpec (a, b, n, c, s, n, cout))

144.16.0545.17240
  
```

The crucial insight is that all of the impossible cases involve bit strings that have impossibly high values. It is trivial to prove the obvious upper bound on an n -bit string. Less obvious is that Isabelle's arithmetic decision procedures can dispose of the impossible cases with the help of that upper bound. We use a couple of tricks. One is that "using" can be inserted before the "apply" command, where it makes the given theorems available. The other trick is the keyword "of", which is described below.

The Opposite Implication

```

Adder.thy (modified)
Adder.thy (D:\Dropbox\ACS11 - Hardware Verification)
Lemma AdderCorrect2:
  "AdderSpec n (a, b, cin, s, cout) ==> AdderImp n (a, b, cin, s, cout)"
proof (induction n arbitrary: cout)
  case 0 thus ?case
  by (simp add: AdderSpec_def)
next
  case (Suc n)
  thus "AdderImp (Suc n) (a, b, cin, s, cout)" using Suc
  by (auto simp add: AdderSpec_Suc AdderCorrect)
qed

using this:
  • AdderSpec n (a, b, cin, s, ?cout) ==> AdderImp n (a, b, cin, s, ?cout)
  • AdderSpec (Suc n) (a, b, cin, s, cout)
  • AdderSpec n (a, b, cin, s, ?cout) ==> AdderImp n (a, b, cin, s, ?cout)
  • AdderSpec (Suc n) (a, b, cin, s, cout)

goal [1 subgoal]:
1. AdderImp (Suc n) (a, b, cin, s, cout)

155.17.0433.19430
  
```

With the help of `AdderSpec_Suc`, the opposite direction of the logical equivalence is a trivial induction.

Making Instances of Theorems

- `thm [of a b c]`
replaces variables by terms from left to right
- `thm [where x=a]`
replaces the variable x by the term a
- `thm [OF thm1 thm2 thm3]`
discharges premises from left to right
- `thm [simplified]`
applies the simplifier to `thm`
- `thm [attr1, attr2, attr3]`
applying multiple attributes

We proved `AdderSpec_Suc` with the help of “using”, which inserted a crucial lemma into the proof. We needed specific instances of the lemma because Isabelle’s arithmetic decision procedures cannot make use of the general formula. Fortunately, we needed only three instances and could express them using the keyword “of”. This type of keyword is called an *attribute*. Attributes modify theorems and sometimes declare them: we have already seen attributes like `[simp]` and `[intro]` many times.

The most useful attributes are shown on the slide. Replacing variables in a theorem by terms (which must be enclosed in quotation marks unless they are atomic) can also be done using “where”, which replaces a named variable. In the left to right list of terms or theorems, use an underscore (`_`) to leave the corresponding item unspecified. An example is `bits_val_less [of _ n]`, which denotes `bits_val ?f n < 2 ^ n`.

Joining theorems conclusion to premise can be done in two different ways. An alternative to `OF` is `THEN`: `thm1 [THEN thm2]` joins the conclusion of `thm1` to the premise of `thm2`. Thus it is equivalent to `thm2 [THEN thm1]`. The result of such combinations can often be `simplified`. Finally, we often want to apply several attributes one after another to a theorem.

See the *Tutorial*, section **5.15 Forward Proof: Transforming Theorems**.

Interactive Formal Verification

12: More Operational Semantics

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Overview

- A *small-step* semantics for the λ -calculus, along with a big-step semantics and steps towards proving their equivalence.
- More techniques involving Isar.
- To conclude, brief references to other Isabelle tools and capabilities.

Our λ -calculus

- The usual variables, abstractions and applications
- But also if-expressions and integer arithmetic
- *Call-by-value* means that the argument is evaluated before a function is called by β -reduction

Taken from Jeremy Siek's AFP entry
Declarative Semantics for Functional Languages

In the pure lambda calculus, integers and arithmetic can be defined through complicated encodings. Here we have a stripped-down functional programming language with recognisable behaviour.

A λ -Calculus Datatype

```

Lambda.thy (/Dropbox/Teach/Semantics_Lambda/)
typedef name ← variable names (unspecified)
datatype exp = EVar name | ENat nat | ELam name exp | EApp exp exp
            | EPrim "nat ⇒ nat ⇒ nat" exp exp | EIf exp exp exp

fun FV :: "exp ⇒ name set" where
  "FV (EVar x) = {x}"
| "FV (ENat n) = {}"
| "FV (ELam x e) = FV e - {x}"
| "FV (EApp e1 e2) = FV e1 ∪ FV e2"
| "FV (EPrim f e1 e2) = FV e1 ∪ FV e2"
| "FV (EIf e1 e2 e3) = FV e1 ∪ FV e2 ∪ FV e3"

fun BV :: "exp ⇒ name set" where
  "BV (EVar x) = {}"
| "BV (ENat n) = {}"
| "BV (ELam x e) = BV e ∪ {x}"
| "BV (EApp e1 e2) = BV e1 ∪ BV e2"
| "BV (EPrim f e1 e2) = BV e1 ∪ BV e2"
| "BV (EIf e1 e2 e3) = BV e1 ∪ BV e2 ∪ BV e3"
    
```

Annotations in the image:

- variable names (unspecified) points to the `name` type parameter in `typedef name`.
- λ -expressions, including arithmetic ops points to the `ELam` constructor in `datatype exp`.
- free and bound variables in a λ -expression points to the `BV` function.

Substitution and Values

```

SmallStepLam.thy (/Dropbox/Teach/Semantics_Lambda/)
text{"This substitution function is not capture avoiding, so $v$ must be closed.*}
fun subst :: "name ⇒ exp ⇒ exp ⇒ exp" where
  "subst x v (EVar y) = (if x = y then v else EVar y)"
| "subst x v (ENat n) = ENat n"
| "subst x v (ELam y e) = (if x = y then ELam y e else ELam y (subst x v e))"
| "subst x v (EApp e1 e2) = EApp (subst x v e1) (subst x v e2)"
| "subst x v (EPrim f e1 e2) = EPrim f (subst x v e1) (subst x v e2)"
| "subst x v (EIf e1 e2 e3) = EIf (subst x v e1) (subst x v e2) (subst x v e3)"

text{"Normal forms (under CBV) include lambda-abstractions*}
inductive normal :: "exp ⇒ bool" where
  nat: "normal (ENat n)"
| lam: "normal (ELam x e)"
declare normal.intros [iff]

text{"Values are normal forms having no free variables*}
abbreviation is_value :: "exp ⇒ bool" where
  "is_value v ≡ normal v ∧ FV v = {}"
    
```

Annotations in the image:

- λ -abstractions are not reduced points to the `lam` constructor in the `normal` inductive definition.
- values are certain λ -expressions points to the `is_value` abbreviation.

Small-Step Semantics of λ -Calculus

$(\lambda x.e)v \rightarrow e[v/x]$ $f(n_1, n_2) \rightarrow$ evaluate f on (n_1, n_2)

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{e_2 \rightarrow e'_2}{e_1 e_2 \rightarrow e_1 e'_2}$$

$$\frac{e_1 \rightarrow e'_1}{f(e_1, e_2) \rightarrow f(e'_1, e_2)} \quad \frac{e_2 \rightarrow e'_2}{f(e_1, e_2) \rightarrow f(e_1, e'_2)}$$

$$\text{If}(0, e_1, e_2) \rightarrow e_2 \quad \frac{n \neq 0}{\text{If}(n, e_1, e_2) \rightarrow e_1}$$

$$\frac{e \rightarrow e'}{\text{If}(e, e_1, e_2) \rightarrow \text{If}(e', e_1, e_2)}$$

Above, n refers to an integer constant as opposed to a general expression, while v is either an integer or a λ -abstraction (both of which are values in our language). In the top line, we see that the argument of a function is evaluated to a value before the function is called (by substitution). A built-in function f when applied to a pair of integers will reduce to another integer.

Small-Step Semantics in Isabelle

```

inductive reduce :: "exp ⇒ exp ⇒ bool" (infix "→" 55) where
  beta: "is_value v ⇒ EApp (ELam x e) v → (subst x v e)"
| app_left: "e1 → e1' ⇒ EApp e1 e2 → EApp e1' e2"
| app_right: "e2 → e2' ⇒ EApp e1 e2 → EApp e1 e2'"
| delta: "EPrim f (ENat n1) (ENat n2) → ENat (f n1 n2)"
| prim_left: "e1 → e1' ⇒ EPrim f e1 e2 → EPrim f e1' e2"
| prim_right: "e2 → e2' ⇒ EPrim f e1 e2 → EPrim f e1 e2'"
| if_zero: "EIf (ENat 0) e1 e2 → e2"
| if_nz: "n ≠ 0 ⇒ EIf (ENat n) e1 e2 → e1"
| if_e: "e → e' ⇒ EIf e e1 e2 → EIf e' e1 e2"
declare reduce.intros [intro]

inductive multi_step :: "exp ⇒ exp ⇒ bool" (infix "→*" 55) where
  ms_nil: "e →* e"
| ms_cons: "[e1 → e2; e2 →* e3] ⇒ e1 →* e3"
declare multi_step.intros [intro]
  
```

A small-step semantics reduces λ-expressions to other λ-expressions one step at a time (hence the name). Formally, we define a reduction relation between two expressions, each of type exp.

A Big-Step Semantics

$$\begin{aligned}
 &\rho \vdash n \Downarrow n \quad \rho \vdash x \Downarrow \rho(x) \quad \rho \vdash \lambda x.e \Downarrow (x, e, \rho) \\
 &\frac{\rho \vdash e_1 \Downarrow n_1 \quad \rho \vdash e_2 \Downarrow n_2}{\rho \vdash f(e_1, e_2) \Downarrow \text{evaluate } f \text{ on } (n_1, n_2)} \\
 &\frac{\rho \vdash e_1 \Downarrow (x, e, \rho') \quad \rho \vdash e_2 \Downarrow w \quad \rho'[w/x] \vdash e \Downarrow v}{\rho \vdash e_1 e_2 \Downarrow v} \\
 &\frac{\rho \vdash e \Downarrow 0 \quad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash \mathbf{If}(e, e_1, e_2) \Downarrow \rho \vdash v_2} \quad \frac{\rho \vdash e \Downarrow n \quad n \neq 0 \quad \rho \vdash e_1 \Downarrow v_1}{\rho \vdash \mathbf{If}(e, e_1, e_2) \Downarrow \rho \vdash v_1}
 \end{aligned}$$

Here environments ρ map variables to values, which are either integers or function closures; the latter are triples consisting of a bound variable, expression and an environment. In other words, environments and values are mutually recursive. A big-step semantics has fewer rules but lacks the ability to express fine points of execution. That doesn't cover problem here.

Big-Step Semantics in Isabelle

```

datatype bval
  = BNat nat
  | BClos name exp "(name × bval) list"

type_synonym benv = "(name × bval) list"

fun lookup :: "('a × 'b) list ⇒ 'a ⇒ 'b option" where
  "lookup [] x = None"
| "lookup ((y,v)#ls) x = (if (x = y) then Some v else lookup ls x)"

inductive eval :: "benv ⇒ exp ⇒ bval ⇒ bool" ("_ ⊢ _ ⊢ _" [50,50,50] 51) where
  eval_nat: "ρ ⊢ ENat n ⊢ BNat n"
| eval_var: "lookup ρ x = Some v ⇒ ρ ⊢ EVar x ⊢ v"
| eval_lam: "ρ ⊢ ELam x e ⊢ BClos x e ρ"
| eval_prim: "[ρ ⊢ e1 ⊢ BNat n1; ρ ⊢ e2 ⊢ BNat n2; n3 = f n1 n2] ⇒ ρ ⊢ EPrim f e1 e2 ⊢ BNat n3"
| eval_app: "[ρ ⊢ e1 ⊢ BClos x e ρ'; ρ ⊢ e2 ⊢ v; (x,v)#ρ' ⊢ e ⊢ v] ⇒ ρ ⊢ EApp e1 e2 ⊢ v"
| eval_if0: "[ρ ⊢ e ⊢ BNat 0; ρ ⊢ e2 ⊢ v2] ⇒ ρ ⊢ EIf e e1 e2 ⊢ v2"
| eval_if1: "[ρ ⊢ e ⊢ BNat n; n ≠ 0; ρ ⊢ e1 ⊢ v1] ⇒ ρ ⊢ EIf e e1 e2 ⊢ v1"
declare eval.intros [intro]
  
```

values are integers and closures

environments and lookup function

Tips for Automatic Reasoning

- Declare introduction rules using `intro` or even `intro!` if the form of the conclusion is unique.
- Use rule inversion (`inductive_cases` or `inductive_simps`), with care.

Rule inversion is so powerful in semantics proofs that it must be regarded as a necessity. But looping is a risk, especially if you declare `elim!` (`inductive_cases`) or `simp` (`inductive_simps`).

Rule Inversion for λ -Calculus

```

inductive_cases
  eval_nat_inv[elim]: "ρ ⊢ ENat n ↓ v"
and eval_var_inv[elim]: "ρ ⊢ EVar x ↓ v"
and eval_lam_inv[elim]: "ρ ⊢ ELam x e ↓ v"
and eval_app_inv[elim]: "ρ ⊢ EApp e1 e2 ↓ v"
and eval_prim_inv[elim]: "ρ ⊢ EPrim f e1 e2 ↓ v"
and eval_if_inv[elim]: "ρ ⊢ EIf e e1 e2 ↓ v"

then eval_if_inv

```

will cause a case split!

It isn't easy to see, but the if-rule causes a case split depending on whether the first around is zero or not. In all the other cases, only one rule matches and there is only one way the given value could have been computed.

Determinacy of Our Semantics

```

theorem
  assumes "ρ ⊢ e ↓ v" "ρ ⊢ e ↓ v'" shows "v = v'"
  using assms apply (induction arbitrary: v')

```

proof (prove)

goal (7 subgoals):

1. $\Delta \rho \ n \ v'. \ \rho \vdash \text{ENat } n \ \Downarrow \ v' \implies \text{BNat } n = v'$
2. $\Delta \rho \ x \ v \ v'. \ [\text{lookup } \rho \ x = \text{Some } v_2; \ \rho \vdash \text{EVar } x \ \Downarrow \ v'] \implies v = v'$
3. $\Delta \rho \ x \ e \ v'. \ \rho \vdash \text{ELam } x \ e \ \Downarrow \ v' \implies \text{BClos } x \ e \ \rho = v'$
4. $\Delta \rho \ e_1 \ n_1 \ e_2 \ n_2 \ n_3 \ f \ v'. \$
 $[\rho \vdash e_1 \ \Downarrow \ \text{BNat } n_1; \ \Delta v'. \ \rho \vdash e_1 \ \Downarrow \ v' \implies \text{BNat } n_1 = v'; \ \rho \vdash e_2 \ \Downarrow \ \text{BNat } n_2;$
 $\Delta v'. \ \rho \vdash e_2 \ \Downarrow \ v' \implies \text{BNat } n_2 = v'; \ n_3 = f \ n_1 \ n_2; \ \rho \vdash \text{EPrim } f \ e_1 \ e_2 \ \Downarrow \ v']$
 $\implies \text{BNat } n_3 = v'$
5. $\Delta \rho \ e_1 \ x \ e' \ e_2 \ w \ v \ v'. \$
 $[\rho \vdash e_1 \ \Downarrow \ \text{BClos } x \ e \ \rho'; \ \Delta v'. \ \rho \vdash e_1 \ \Downarrow \ v' \implies \text{BClos } x \ e \ \rho' = v'; \ \rho \vdash e_2 \ \Downarrow \ w;$
 $\Delta v'. \ \rho \vdash e_2 \ \Downarrow \ v' \implies w = v'; \ (x, w) \# \rho' \vdash e \ \Downarrow \ v_2;$
 $\Delta v'. \ (x, v) \# \rho' \vdash e \ \Downarrow \ v' \implies v = v'; \ \rho \vdash \text{EApp } e_1 \ e_2 \ \Downarrow \ v']$
 $\implies v = v'$
6. $\Delta \rho \ e \ e_2 \ v_2 \ e_1 \ v'. \$
 $[\rho \vdash e \ \Downarrow \ \text{BNat } 0; \ \Delta v'. \ \rho \vdash e \ \Downarrow \ v' \implies \text{BNat } 0 = v'; \ \rho \vdash e_2 \ \Downarrow \ v_2;$

As we have seen before, these complicated-looking subgoals are easy to prove with the help of rule inversion. But they are not completely trivial and the full proof looks like this:

```

theorem big_step_fun:
  assumes "ρ ⊢ e ↓ v" "ρ ⊢ e ↓ v'" shows "v = v'"
  using assms by (induction arbitrary: v') (blast | fastforce)+

```

Induction is performed followed by either `blast` or `fastforce`, repeatedly.

Towards Semantic Equivalence

```

subsection 'Substitution wrt environments, and its properties'

type_synonym env = "(name × exp) list"

fun esubst :: "env ⇒ exp ⇒ exp" where
  "esubst ρ (ENat n) = ENat n"
| "esubst ρ (EVar x) = (case lookup ρ x of None ⇒ EVar x | Some v ⇒ v)"
| "esubst ρ (ELam x e) = ELam x (esubst ((x, EVar x)#ρ) e)"
| "esubst ρ (EApp e1 e2) = EApp (esubst ρ e1) (esubst ρ e2)"
| "esubst ρ (EPrim f e1 e2) = EPrim f (esubst ρ e1) (esubst ρ e2)"
| "esubst ρ (EIF e1 e2 e3) = EIF (esubst ρ e1) (esubst ρ e2) (esubst ρ e3)"

definition domain :: "env ⇒ name set" where
  "domain ρ = {x. ∃v. lookup ρ x = Some v ∧ v ≠ EVar x}"

definition closed_env :: "env ⇒ bool" where
  "closed_env ρ = (∀x v. x ∈ domain ρ ⇒ lookup ρ x = Some v ⇒ FV v = {})"

definition equiv_env :: "env ⇒ env ⇒ bool" where
  "equiv_env ρ ρ' ≡ domain ρ = domain ρ' ∧ (∀x ∈ domain ρ. lookup ρ x = lookup ρ' x)"
  
```

Our previous substitution function replaced a single variable, but now we need a substitution function driven by an environment. Later we shall prove that the two notions of substitution are equivalent. Note that the domain of an environment is the set of variables that are not mapped to themselves. A closed environment contains only closed terms (those with no free variables). Two environments are equivalent if their domains are the same and they agree on all variables.

Big vs Small-Step Values

```

text 'Relating big-step and small-step values and environments'
inductive bs_val :: "bval ⇒ exp ⇒ bool" and bs_env :: "benv ⇒ env ⇒ bool" where
  bs_nat: "bs_val (ENat n) (ENat n)"
| bs_clos: "[bs_env ρ ρ'; FV (ELam x (esubst ((x, EVar x)#ρ') e)) = {}] ⇒ bs_val (BClos x e ρ') (ELam x (esubst ((x, EVar x)#ρ') e))"
| bs_nil: "bs_env [] []"
| bs_cons: "[bs_val v v; bs_env ρ ρ'] ⇒ bs_env ((x, w)#ρ) ((x, v)#ρ)"
declare bs_val_bs_env.intros [intros!]

inductive_cases
  bsenv_nil_inv[elim!]: "bs_env [] ρ"
and bs_env_inv[elim!]: "bs_env (u # ρ) ρ'"
and bs_clos_inv[elim!]: "bs_val (BClos x e ρ') v1"
and bs_nat_inv[elim!]: "bs_val (ENat n) v"

lemma bs_val_is_val: "bs_val w v ⇒ is_value v"
  by (cases w) auto

lemma lookup_bs_env:
  "[bs_env ρ ρ'; lookup ρ x = Some w] ⇒ ∃v. lookup ρ' x = Some v ∧ bs_val w v"
  by (induction ρ arbitrary: ρ' x w) auto
  
```

Because big-step values and environments are mutually recursive, we declare them simultaneously. This declares a single internal relation, `bs_val1_bs_env` in addition to the two relations declared explicitly. These relations connect the values (`bval` versus `exp`) and environments (`benv` versus `env`) of the two semantic definitions.

The construction of a correct closure (second line) is rather technical, while the construction of a new environment requires the insertion of compatible values. (Note that `ρ` and `ρ'` have different types!)

Proving Equivalence (!!!)

```

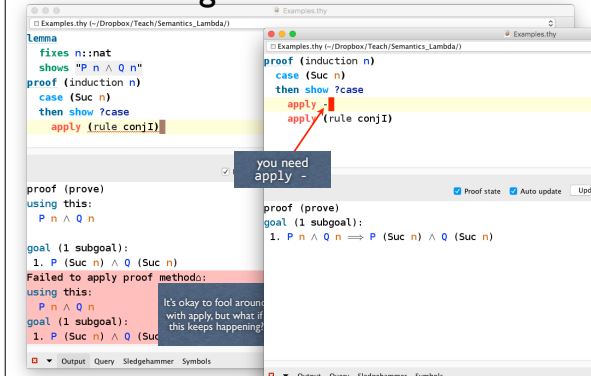
theorem big_small_step:
  assumes "ρ = {} w" "bs_env ρ ρ'" "FV e ⊆ set (map fst ρ)"
  shows "∃v. esubst ρ' e ⇒ v ∧ is_value v ∧ bs_val w v"
  using assms
  proof (induction arbitrary: ρ' rule: eval.induct)
  case (eval_app ρ e1 x e ρ' e2 w v ρ')
  then obtain v1 v2
    where e1_v1: "esubst ρ' e1 ⇒ v1" and clos_v1: "bs_val (BClos x e ρ') v1"
    and e2_v2: "esubst ρ' e2 ⇒ v2" and "is_value v2" and "bs_val w v2"
  by simp blast
  then have I2: "EApp (esubst ρ' e1) (esubst ρ' e2) ⇒ EApp v1 (esubst ρ' e2)"
    by (simp_all add: app_red_cong1 e1_v1 app_red_cong2 e2_v2)
  obtain ρ2 where rpp_r2: "bs_env ρ' ρ2" and fv_v1: "FV v1 = {}"
  and v1_lam: "v1 = ELam x (esubst ((x, EVar x)#ρ2) e)" and cr2: "closed_env ρ2"
  using clos_v1 bs_env_closed by auto
  then have fvs: "FV (esubst ((x, EVar x)#ρ2) e) = FV e - domain ((x, EVar x)#ρ2)"
  using esubst_fv[of "(x, EVar x)#ρ2"] by (fastforce simp: closed_env_def)
  let ?r2 = "{(x, v) # ρ'}"
  let ?r = "{(x, v2) # ρ2}"
  have r2_r: "set (map fst ?r2) = domain ?r" and r_r: "bs_env ?r2 ?r"
  
```

Here we see only part of one case of the induction: for function applications. The full proof is complicated, and 80 lines long. (The original on the AFP was considerably longer than that.)

The statement assumes that expression `e` evaluates (with the big step semantics) to some value `w` under an environment `ρ` that assigns values to all the free variables of `e`. (Note that the function domain is for small-step environments.) The theorem concludes that the evaluates (with a small step semantics) to some value `v` that is equivalent to `w`.

Because of the existential claim in the theorem, the need to check free-variable conditions and to relate the intermediate values between the two semantic definitions, we are forced to elaborately consider each step even in the evaluation of a single function application. Quite a few additional lemmas are needed to get to this point. They aren't shown here but can be downloaded if you are interested.

Finding Structured Proofs



A common way to arrive at structured proofs is to look for a short sequence of `apply`-steps that solve the goal at hand. If successful, you can even leave this sequence (terminated by “done”) as part of the proof, though it is better style to shorten it to a use of “by”. Sometimes however almost everything you try produces an error message. The problem may be that you are piping facts into your proof using `then/hence/thus/using`. Some proof methods (in particular, “rule”) and its variants expect these facts to match a premise of the theorem you give to “rule”. The simplest way to deal with this situation is to type `apply -`, which simply inserts those facts as new assumptions. It would be very ugly to leave `-` as a step in your final proof, but it is useful when exploring.

Other Facets of Isabelle

- **Document preparation:** you can generate LATEX documents from your theories.
- **Axiomatic type classes:** a general approach to polymorphism and overloading when there are shared laws.
- **Code generation:** you can generate executable code from the formal functional programs you have verified.
- **Locales:** encapsulated contexts, ideal for formalising abstract mathematics.

See the *Tutorial*, section 4.2, for an introduction to document preparation.

Locales are documented in the “Tutorial to Locales and Locale Interpretation” by Clemens Ballarin, which can be downloaded from Isabelle’s documentation page.

Axiomatic Type Classes

- Controlled overloading of operators, including $+ - \times / \wedge \leq$ and even `gcd`
- Can define concept hierarchies abstractly:
 - Prove theorems about an operator from its axioms
 - Prove that a type belongs to a class, making those theorems available
- Crucial to Isabelle’s formalisation of arithmetic

Axiomatic type classes are inspired by the type class concept in the programming language Haskell, which is based on the following seminal paper:

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16th Annual Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

A very early version was available in Isabelle by 1993:

Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

More recent papers include the following:

Markus Wenzel. Type Classes and Overloading in Higher-Order Logic. In Elsa L. Gunter and Amy P. Felty, *Theorem Proving in Higher Order Logics*. Springer Lecture Notes In Computer Science 1275 (1997), 307 - 322.

Code Generation

- Isabelle/HOL formulas can be translated to equivalent ML and Haskell code.
- Inefficient and non-executable parts of definitions can be replaced by equivalent, efficient terms.
- Algorithms can be verified and then executed.
- The method `eval` provides *reflection*: it proves equations by execution.

See “Code generation from Isabelle/HOL theories”, by Florian Haftmann; it can be downloaded from Isabelle’s documentation page.

The End

You know my methods. Apply them!
Sherlock Holmes