Welcome Again!

- Slides and File are as usual at:
  `http://isabelle.in.tum.de/nominal/activities/cas09/`
- Did all installation problems with Isabelle resolve?
- Any questions about the last tutorial?

Automatic Proofs

- Remember that I said: Do not expect that Isabelle solves automatically **show** "P=NP".
- Remember also:

```
lemma even_twice:
  shows "even (n + n)"
by (induct n) (auto)

lemma even_add:
  assumes a: "even n"
  and     b: "even m"
  shows "even (n + m)"
using a b by (induct) (auto)
```

A More Complicated Proof

```
lemma even_mult:
  assumes a: "even n"
  shows "even (n * m)"
using a proof (induct)
  case eZ
  show "even (0 * m)" by auto
next
  case (eSS n)
  have ih: "even (n * m)" by fact
  have "(Suc (Suc n) * m) = (m + m) + (n * m)" by simp
  moreover
  have "even (m + m)" using even_twice by simp
  ultimately
  show "even (Suc (Suc n) * m)" using ih even_add by (simp only:)
qed
```

- This proof cannot be found by the internal tools.

A More Complicated Proof

```
lemma even_mult:
  assumes a: "even n"
  shows "even (n * m)"
using a proof (induct)
```

Sledgehammer:

Can be used at any point in the development.

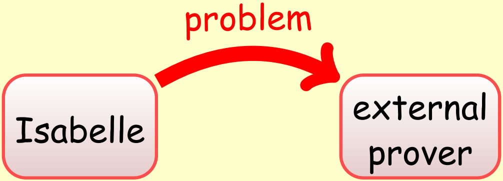
Isabelle

A More Complicated Proof

```
lemma even_mult:
  assumes a: "even n"
  shows "even (n * m)"
using a proof (induct)
  case e7
```

Sledgehammer:

Can be used at any point in the development.

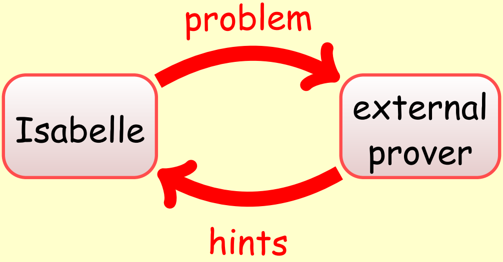


A More Complicated Proof

```
lemma even_mult:
  assumes a: "even n"
  shows "even (n * m)"
using a proof (induct)
```

Sledgehammer:

Can be used at any point in the development.



With Sledgehammer

- It can be started with ctrl-c/ctrl-a/ctrl-s.

```
lemma even_mult_auto:
  assumes a: "even n"
  shows "even (n * m)"
using a
apply(induct)
apply(metis eZ mult_is_0)
apply(metis even_add even_twice mult_Suc_right
                   nat_add_assoc nat_mult_commute)
done
```

With Sledgehammer

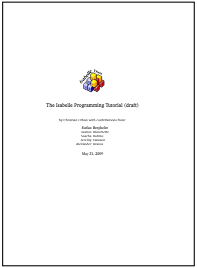
- It can be started with ctrl-c/ctrl-a/ctrl-s.

```
lemma even_mult_auto:
  assumes a: "even n"
  shows "even (n * m)"
using a
apply(induct)
apply(metis eZ mult_is_0)
apply(metis even_add even_twice mult_Suc_right
              nat_add_assoc nat_mult_commute)
done
```

- The disadvantage of such proofs is that you have no idea why they are true.

Decision Procedures

- You can write your own proof procedures either within Isabelle or feed back certificates like Sledgehammer.
- We have a tutorial explaining the Isabelle interfaces, but this is well beyond this tutorial.


Isabelle Docu

The Isabelle Programming Tutorial (draft)

by Christian Urban with contributions from:

Stefan Berghofer
Jasmin Blanchette
Sascha Böhme
Jeremy Dawson
Alexander Krauss

May 31, 2009

http://isabelle.in.tum.de/nominal/activities/idp/

Functions

- Let us return to function definitions: for example the Fibonnacci function

```
fun
  fib :: "nat ⇒ nat"
where
  "fib 0 = 0"
| "fib (Suc 0) = 1"
| "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Functions

- Let us return to function definitions: for example the Fibonnacci function

```
fun
  fib :: "nat ⇒ nat"
where
  "fib 0 = 0"
| "fib (Suc 0) = 1"
| "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

- We have to make sure every function terminates (this is proved automatically for the Fibonacci function).

$$f(x) = f(x) + 1$$
$$0 = 1$$

Functions

- The Ackermann function is also automatically proved to be terminating:

```
fun
  ack :: "nat ⇒ nat ⇒ nat"
where
  "ack 0 m = Suc m"
| "ack (Suc n) 0 = ack n (Suc 0)"
| "ack (Suc n) (Suc m) = ack n (ack (Suc n) m)"
```

Functions

- The Ackermann function is also automatically proved to be terminating:

```
fun
  ack :: "nat ⇒ nat ⇒ nat"
where
  "ack 0 m = Suc m"
| "ack (Suc n) 0 = ack n (Suc 0)"
| "ack (Suc n) (Suc m) = ack n (ack (Suc n) m)"
```

- For others you might have to show explicitly that they are terminating (for example by a decreasing measure).

- For example a generalised version of the Fibonacci function to integers cannot be automatically shown terminating.

```
function
  fib' :: "int ⇒ int"
where
  "n< -1 ⟹ fib' n = fib' (n + 2) - fib' (n + 1)"
| "fib' -1 = (1::int)"
| "fib' 0 = (0::int)"
| "fib' 1 = (1::int)"
| "n > 1 ⟹ fib' n = fib' (n - 1) + fib' (n - 2)"
by (atomize_elim, presburger) (auto)

termination
  by (relation "measure (λx. nat (|x|))")
     (simp_all add: zabs_def)
```

Datatypes

- You can introduce new datatypes. For example "my"-lists:

```
datatype 'a mylist =
  MyNil                    ("[]")
| MyCons "'a" "'a mylist" ("_ ::: _" 65)
```

Datatypes

- You can introduce new datatypes. For example "my"-lists:

```
datatype 'a mylist =
  MyNil                    ("[]")
| MyCons "'a" "'a mylist" ("_ ::: _" 65)

fun myappend :: "'a mylist ⇒ 'a mylist ⇒ 'a mylist" ("_ @@ _" 65)
where
  "[] @@ xs = xs"
| "(y:::ys) @@ xs = y:::(ys @@ xs)"

fun myrev :: "'a mylist ⇒ 'a mylist"
where
  "myrev [] = []"
| "myrev (x:::xs) = (myrev xs) @@ (x:::[])"
```

Your Turn

```
lemma myrev_append:
  shows "myrev (xs @@ ys) = (myrev ys) @@ (myrev xs)"
proof (induct xs)
  case MyNil
  show "myrev ([] @@ ys) = myrev ys @@ myrev []" sorry

next
  case (MyCons x xs)
  have ih: "myrev (xs @@ ys) = myrev ys @@ myrev xs" by fact

  show "myrev ((x:::xs) @@ ys) = myrev ys @@ myrev (x:::xs)"
    sorry
qed
```

A WHILE Language

- The memory is a function from nat to nat.

```
types memory = "nat ⇒ nat"
```

A WHILE Language

- The memory is a function from nat to nat.

```
types memory = "nat ⇒ nat"
```

- Arithmetical expressions are defined as:

```
datatype aexp =
  C nat
| X nat
| Op1 "nat ⇒ nat" aexp
| Op2 "nat ⇒ nat ⇒ nat" aexp aexp
```

- Arithmetical expressions are defined as:

```
datatype bexp =
  TRUE | FALSE
| ROp  "nat ⇒ nat ⇒ bool" aexp aexp
| NOT bexp | AND bexp bexp | OR  bexp bexp
```

Commands

- Commands are defined also as datatype:

```
datatype cmd =
  SKIP
| ASSIGN nat aexp        ("_ ::= _ " 60)
| SEQ    cmd cmd         ("_; _" [60, 60] 10)
| COND   bexp cmd cmd  ("IF _ THEN _ ELSE _" 60)
| WHILE  bexp cmd        ("WHILE _ DO _" 60)
```

- We use ::=, because := is already used for function update.

Commands

- Commands are defined also as datatype:

```
datatype cmd =
  SKIP
| ASSIGN nat aexp        ("_ ::= _ " 60)
| SEQ    cmd cmd         ("_; _" [60, 60] 10)
| COND   bexp cmd cmd    ("IF _ THEN _ ELSE _" 60)
| WHILE  bexp cmd        ("WHILE _ DO _" 60)
```

- We use ::=, because := is already used for function update.
- We have to define a semantics for the WHILE programs...

An Abstract Machine

- The instruction set

```
datatype instr =
   JPFZ  "nat"                    jump forward n steps, if stack is 0
 | JPB   "nat"                    jump backward n steps
 | FETCH "nat"                    move memory to top of stack
 | STORE "nat"                    pop top from stack to memory
 | PUSH  "nat"                    push to stack
 | OPU   "nat ⇒ nat"              pop one from stack and apply f
 | OPB   "nat ⇒ nat ⇒ nat"        pop two from stack and apply f
```

- A machine program is a list of instructions.
- Representation of booleans is 0 and 1

The Compiler Functions

```
fun compa
where
  "compa (C n) = [PUSH n]"
| "compa (X l) = [FETCH l]"
| "compa (Op1 f e) = (compa e) @ [OPU f]"
| "compa (Op2 f e₁ e₂) = (compa e₁) @ (compa e₂) @ [OPB f]"

fun compb
where
  "compb (TRUE) = [PUSH 1]"
| "compb (FALSE) = [PUSH 0]"
| "compb (ROp f e₁ e₂) = (compa e₁) @ (compa e₂)
                              @ [OPB (λx y. WRAP (f x y))]"
| "compb (NOT e) = (compb e) @ [OPU MNot]"
| "compb (AND e₁ e₂) = (compb e₁) @ (compb e₂) @ [OPB MAnd]"
| "compb (OR e₁ e₂) = (compb e₁) @ (compb e₂) @ [OPB MOr]"
```

The Compiler Functions

```
fun
  compc :: "cmd ⇒ instr list"
where
  "compc SKIP = []"
| "compc (x::=a) = (compa a) @ [STORE x]"
| "compc (c₁;c₂) = compc c₁ @ compc c₂"
| "compc (IF b THEN c₁ ELSE c₂) =
    (compb b) @ [JPFZ (length(compc c₁) + 2)] @ compc c₁ @
    [PUSH 0, JPFZ (length(compc c₂))] @ compc c₂"
| "compc (WHILE b DO c) =
    (compb b) @
    [JPFZ (length(compc c) + 1)] @ compc c @
    [JPB (length(compc c) + length(compb b)+1)]"
```

The Compiler Functions

```
fun
  compc :: "cmd ⇒ instr list"
where
  "compc SKIP = []"
| "compc (x::=a) = (compa a) @ [STORE x]"
| "compc (c1;c2) = compc c1 @ compc c2"
| "compc (IF b THEN c1 ELSE c2) =
    (compb b) @ [JPFZ (length(compc c1) + 2)] @ compc c1 @
    [PUSH 0, JPFZ (length(compc c2))] @ compc c2"
| "compc (WHILE b DO c) =
    (compb b) @
    [JPFZ (length(compc c) + 1)] @ compc c @
    [JPB (length(compc c) + length(compb b)+1)]"
```

- We now have to specify how the machine behaves.

Compiler Lemmas

- We like to prove:

```
lemma compa:
  assumes a: "(e, m) ⟶a n"
  shows "(compa e,[],[],m) ⟶m* ([],rev (compa e),[n], m)"

lemma compb:
  assumes a: "(e, m) ⟶b b"
  shows "(compb e,[],[],m) ⟶m* ([],rev (compb e),[WRAP b], m)"

lemma compc:
  assumes a: "(c, m) ⟶c m'"
  shows "(compc c,[],[],m) ⟶m* ([],rev (compc c),[], m')"
```

Compiler Lemmas

- They can be found automatically:

```
lemma compa_aux_cheating:
  assumes a: "(e,m) ⟶a n"
  shows "(compa e@p,q,s,m) ⟶m* (p,rev (compa e)@q,n#s,m)"
using a
by (induct arbitrary: p q s)
   (force intro: steps_trans simp add: steps_simp exec_simp)+
```

Compiler Lemmas

- They can be found automatically:

```
lemma compa_aux_cheating:
  assumes a: "(e,m) ⟶a n"
  shows "(compa e@p,q,s,m) ⟶m* (p,rev (compa e)@q,n#s,m)"
using a
by (induct arbitrary: p q s)
   (force intro: steps_trans simp add: steps_simp exec_simp)+
```

- **But that is cheating!!!** It is like playing chess with the help of Kasparov.

Isabelle Tutorial

Please also come tomorrow.

- 9:30 - 11:30, Tuesday, 2 June

- If Isabelle still does not run, maybe I can help.
- Please ask any question.