

Interactive Formal Verification

Review (1-7)

Tjark Weber
Computer Laboratory
University of Cambridge

Isabelle Theories

```
theory T imports Main A B
```

```
begin
```

```
end
```

Isabelle Theories



Name of the theory

```
theory T imports Main A B
```

```
begin
```

```
end
```

Isabelle Theories

Name of the theory

```
theory T imports Main A B
```

```
begin
```

Names of existing theories

```
end
```

Isabelle Theories

Name of the theory

```
theory T imports Main A B
```

```
begin
```

Main: contains all of Isabelle/HOL

Names of existing theories

```
end
```

Defining Types

- `typedefcl ('a, 'b) t`

Defining Types

Introduces an unspecified type

- `typedef` ('a, 'b) t

Defining Types

Introduces an unspecified type

- `typedef` ('a, 'b) t

Optional: type arguments

Defining Types

Introduces an unspecified type

- `typedecl ('a, 'b) t`

Optional: type arguments

- `type_synonym`

```
'a multiset = "'a => nat"
```

Defining Types

Introduces an unspecified type

- `typedecl ('a, 'b) t`

Optional: type arguments

- `type_synonym`

Introduces a new name for an existing type

```
'a multiset = "'a => nat"
```

Defining Types

Introduces an unspecified type

- `typedecl ('a, 'b) t`

Optional: type arguments

- `type_synonym`

Introduces a new name for an existing type

```
'a multiset = "'a => nat"
```

- `datatype 'a list =`

```
Nil | Cons 'a "'a list"
```

Defining Types

Introduces an unspecified type

- `typedecl ('a, 'b) t`

Optional: type arguments

- `type_synonym`

Introduces a new name for an existing type

```
'a multiset = "'a => nat"
```

Defines an inductive datatype

- `datatype 'a list =`

```
Nil | Cons 'a "'a list"
```

Defining Types

Introduces an unspecified type

- `typedecl ('a, 'b) t`

Optional: type arguments

- `type_synonym`

Introduces a new name for an existing type

```
'a multiset = "'a => nat"
```

Defines an inductive datatype

- `datatype 'a list =`

```
Nil | Cons 'a "'a list"
```

Constructor names and argument types

Defining Constants

- `definition` `even` `::` `"nat => bool"`
`where` `"even n = ($\exists k. n = 2*k$)"`

Defining Constants

For non-recursive definitions

- `definition` `even` `::` `"nat => bool"`
`where` `"even n = ($\exists k. n = 2*k$)"`

Defining Constants

For non-recursive definitions

Optional: the constant's type

- `definition` `even` `::` `"nat => bool"`
`where` `"even n = ($\exists k. n = 2*k$)"`

Defining Constants

For non-recursive definitions

Optional: the constant's type

- `definition` `even` `::` `"nat => bool"`
`where` `"even n = ($\exists k. n = 2*k$)"`

Provides a lemma: `even_def`

Defining Constants

For non-recursive definitions

Optional: the constant's type

- `definition` `even` :: `"nat => bool"`
where `"even n = ($\exists k. n = 2*k$)"`

Provides a lemma: `even_def`

- `fun` `even'` where
 `"even' 0 = True"`
 | `"even' (Suc 0) = False"`
 | `"even' n = even' (n-2)"`

Defining Constants

For non-recursive definitions

Optional: the constant's type

- `definition` `even` :: `"nat => bool"`
`where` `"even n = ($\exists k. n = 2*k$)"`

For recursive functions

Provides a lemma: `even_def`

- `fun` `even'` `where`
 `"even' 0 = True"`
| `"even' (Suc 0) = False"`
| `"even' n = even' (n-2)"`

Defining Constants

For non-recursive definitions

Optional: the constant's type

- `definition even :: "nat => bool"`
`where "even n = ($\exists k. n = 2*k$)"`

For recursive functions

Provides a lemma: `even_def`

- `fun even' where`
`"even' 0 = True"`
`| "even' (Suc 0) = False"`
`| "even' n = even' (n-2)"`

Provides `even'.simps` and `even'.induct`

Defining Constants

- `inductive_set tcl`
for `R :: "('a*'a) set"`
where
 `"(x,y):R ==> (x,y):tcl R"`
| `"(x,y):tcl R ==> (y,z):tcl R`
 `==> (x,z):tcl R"`

Defining Constants

For inductive sets

- `inductive_set tcl`
`for R :: "('a*'a) set"`
`where`
 `"(x,y):R ==> (x,y):tcl R"`
| `"(x,y):tcl R ==> (y,z):tcl R`
 `==> (x,z):tcl R"`

Defining Constants

For inductive sets

Optional: the constant's type

- `inductive_set tcl`
`for R :: "('a*'a) set"`
`where`
 `"(x,y):R ==> (x,y):tcl R"`
| `"(x,y):tcl R ==> (y,z):tcl R`
 `==> (x,z):tcl R"`

Defining Constants

For inductive sets

Optional: the constant's type

- `inductive_set tcl`

```
for R :: "( $\alpha$ * $\beta$ ) set"
```

```
where
```

```
"(x,y):R ==> (x,y):tcl R"
```

```
| "(x,y):tcl R ==> (y,z):tcl R  
   ==> (x,z):tcl R"
```

Parameters (types are optional again)

Defining Constants

For inductive sets

Optional: the constant's type

- `inductive_set tcl`

`for R :: "(α * β) set"`

`where`

`"(x,y):R ==> (x,y):tcl R"`

`| "(x,y):tcl R ==> (y,z):tcl R
==> (x,z):tcl R"`

Parameters (types are optional again)

Provides `tcl.cases`, `tcl.induct`,
`tcl.intros` and `tcl.simps`

Theorems and Proofs

- `lemma add_com [simp]: "x+y = y+x"`
- `apply` method
- `done`
- `by` method
- `oops`
- `sorry`

Theorems and Proofs

Starts a proof

- `lemma` `add_com [simp]: "x+y = y+x"`
- `apply` method
- `done`
- `by` method
- `oops`
- `sorry`

Theorems and Proofs

Starts a proof

Optional: a name and attributes

- `lemma` `add_com [simp]: "x+y = y+x"`
- `apply` method
- `done`
- `by` method
- `oops`
- `sorry`

Theorems and Proofs

Starts a proof

Optional: a name and attributes

- `lemma` `add_com [simp]: "x+y = y+x"`
- `apply` method

Modifies some subgoal(s)
- `done`
- `by` method
- `oops`
- `sorry`

Theorems and Proofs

Starts a proof

Optional: a name and attributes

- `lemma` `add_com [simp]: "x+y = y+x"`
- `apply` method

Modifies some subgoal(s)
- `done`

Finishes a proof
- `by` method
- `oops`
- `sorry`

Theorems and Proofs

Starts a proof

Optional: a name and attributes

- `lemma` `add_com [simp]: "x+y = y+x"`
- `apply` method

Modifies some subgoal(s)
- `done`

Finishes a proof
- `by` method

Finishes a proof in a single step
- `oops`
- `sorry`

Theorems and Proofs

Starts a proof

Optional: a name and attributes

- `lemma` `add_com [simp]: "x+y = y+x"`
- `apply` method

Modifies some subgoal(s)
- `done`

Finishes a proof
- `by` method

Finishes a proof in a single step
- `oops`

Aborts a proof attempt
- `sorry`

Theorems and Proofs

Starts a proof

Optional: a name and attributes

- `lemma` `add_com [simp]: "x+y = y+x"`

- `apply` method

Modifies some subgoal(s)

- `done`

Finishes a proof

- `by` method

Finishes a proof in a single step

- `oops`

Aborts a proof attempt

- `sorry`

Finishes a proof (cheating!)

Automated Proof Methods

- `(induct x y arbitrary: z rule: r.induct)`
- `(simp add: l1 del: l2)`
- `(auto simp add: l1 intro: l2)`
- `(blast intro: l1 elim: l2)`
- `arith`
- `(metis l1 l2 l3)`
- `sledgehammer`

Automated Proof Methods

Induction

- `(induct x y arbitrary: z rule: r.induct)`
- `(simp add: l1 del: l2)`
- `(auto simp add: l1 intro: l2)`
- `(blast intro: l1 elim: l2)`
- `arith`
- `(metis l1 l2 l3)`
- `sledgehammer`

Automated Proof Methods

Induction

- `(induct x y arbitrary: z rule: r.induct)`
- `(simp add: l1 del: l2)`
- `(auto simp add: l1 intro: l2)`
- `(blast intro: l1 elim: l2)`
- `arith`
- `(metis l1 l2 l3)`
- `sledgehammer`

Simplification

Automated Proof Methods

Induction

- `(induct x y arbitrary: z rule: r.induct)`

Simplification

- `(simp add: l1 del: l2)`

Simplification and some logic

- `(auto simp add: l1)`

- `(blast intro: l1 elim: l2)`

- `arith`

- `(metis l1 l2 l3)`

- `sledgehammer`

Automated Proof Methods

Induction

- `(induct x y arbitrary: z rule: r.induct)`

Simplification

- `(simp add: l1 del: l2)`

Simplification and some logic

- `(auto simp add: l1)`

Good for sets and quantifiers

- `(blast intro: l1 elim: l2)`

- `arith`

- `(metis l1 l2 l3)`

- `sledgehammer`

Automated Proof Methods

Induction

- `(induct x y arbitrary: z rule: r.induct)`

Simplification

- `(simp add: l1 del: l2)`

Simplification and some logic

- `(auto simp add: l1)`

Good for sets and quantifiers

- `(blast intro: l1 elim: l2)`

Good for arithmetic goals

- `arith`

- `(metis l1 l2 l3)`

- `sledgehammer`

Automated Proof Methods

Induction

• `(induct x y arbitrary: z rule: r.induct)`

Simplification

• `(simp add: l1 del: l2)`

Simplification and some logic

• `(auto simp add: l1)`

Good for sets and quantifiers

• `(blast intro: l1 elim: l2)`

Good for arithmetic goals

• `arith`

• `(metis l1 l2 l3)`

Powerful first-order prover

• `sledgehammer`

Automated Proof Methods

Induction

• `(induct x y arbitrary: z rule: r.induct)`

Simplification

• `(simp add: l1 del: l2)`

Simplification and some logic

• `(auto simp add: l1)`

Good for sets and quantifiers

• `(blast intro: l1 elim: l2)`

Good for arithmetic goals

• `arith`

• `(metis l1 l2 l3)`

Powerful first-order prover

• `sledgehammer`

Finds lemmas for metis

Basic Methods for Rules

thm: "[| P1; ...; Pn |] ==> Q"

- (rule thm)
- (erule thm)
- (drule thm)
- (frule thm)

- (rule_tac x="..." and y="..." in thm)

Basic Methods for Rules

thm: "[| P1; ...; Pn |] ==> Q"

Unifies Q with the conclusion

- (rule thm)
- (erule thm)
- (drule thm)
- (frule thm)

- (rule_tac x="..." and y="..." in thm)

Basic Methods for Rules

thm: "[| P1; ...; Pn |] ==> Q"

Unifies Q with the conclusion

- (rule thm)

Unifies Q; unifies P1 with some assumption

- (erule thm)

- (drule thm)

- (frule thm)

- (rule_tac x="..." and y="..." in thm)

Basic Methods for Rules

thm: "[| P1; ...; Pn |] ==> Q"

- (rule thm)

Unifies Q with the conclusion

- (erule thm)

Unifies Q; unifies P1 with some assumption

- (drule thm)

Unifies P1 with some assumption

- (frule thm)

- (rule_tac x="..." and y="..." in thm)

Basic Methods for Rules

thm: "[| P1; ...; Pn |] ==> Q"

- (rule thm)

Unifies Q with the conclusion

- (erule thm)

Unifies Q; unifies P1 with some assumption

- (drule thm)

Unifies P1 with some assumption

- (frule thm)

Like drule, but does not delete the assumption

- (rule_tac x="..." and y="..." in thm)

Basic Methods for Rules

thm: "[| P1; ...; Pn |] ==> Q"

- (rule thm)

Unifies Q with the conclusion

- (erule thm)

Unifies Q; unifies P1 with some assumption

- (drule thm)

Unifies P1 with some assumption

- (frule thm)

Like drule, but does not delete the assumption

- (rule_tac x="..." and y="..." in thm)

Manual instantiation of variables

Insiders' Tips

- `term " . . . "`
- `thm name`
- Find theorems
- Isabelle > Settings > Display ...
- Isabelle > Show me ...

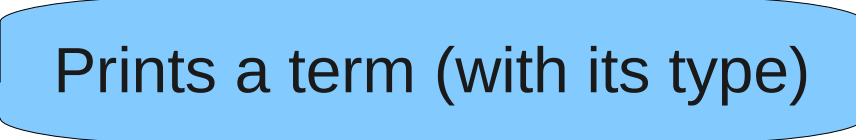
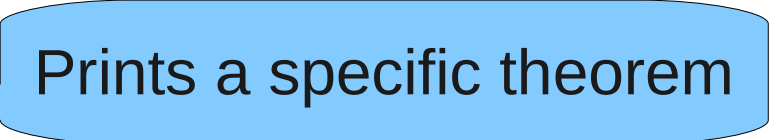
Insiders' Tips

- `term " . . . "`

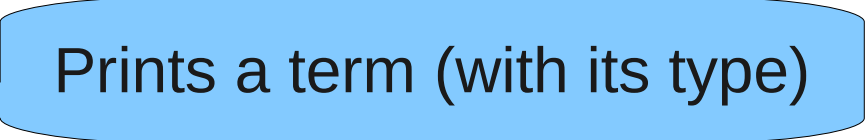

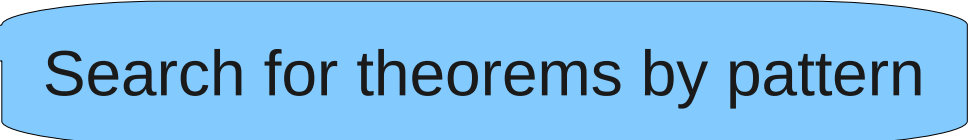
Prints a term (with its type)

- `thm name`
- Find theorems
- Isabelle > Settings > Display ...
- Isabelle > Show me ...

Insiders' Tips

- `term " . . . "` 
- `thm name` 
- Find theorems
- Isabelle > Settings > Display ...
- Isabelle > Show me ...

Insiders' Tips

- `term " . . . "` 
- `thm name` 
- Find theorems 
- Isabelle > Settings > Display ...
- Isabelle > Show me ...

Insiders' Tips

- `term " . . . "` Prints a term (with its type)
- `thm name` Prints a specific theorem
- Find theorems Search for theorems by pattern
- Isabelle > Settings > Display ...
- Isabelle > Show me ... Show types, sorts etc.

Insiders' Tips

- `term " . . . "` Prints a term (with its type)
 - `thm name` Prints a specific theorem
 - Find theorems Search for theorems by pattern
 - Isabelle > Settings > Display ...
 - Isabelle > Show me ... Show types, sorts etc.
- Show all commands, all methods etc.