Chapter 2

Functional Programming:Isabelle

Overview of Isabelle/HOL

System Architecture

Isabelle	generic theorem prover

System Architecture

Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover

System Architecture

ProofGeneral	(X)Emacs based interface
Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover

HOL

HOL = Higher-Order Logic

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators ($\wedge$, $\longrightarrow$, $\forall$, $\exists$, ...)

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators ($\wedge$, $\longrightarrow$, $\forall$, $\exists$, ...)

HOL is a programming language!

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators ($\wedge$, $\longrightarrow$, $\forall$, $\exists$, ...)

HOL is a programming language!

Higher-order = functions are values, too!

Formulae

Syntax (in decreasing priority):

$$
\begin{array}{lcllll}
form & ::= & (form) & | & term = term & | & \neg form \\
 & | & form \wedge form & | & form \vee form & | & form \longrightarrow form \\
 & | & \forall x.\ form & | & \exists x.\ form & &
\end{array}
$$

Formulae

Syntax (in decreasing priority):

$$\begin{array}{rcl} form & ::= & (form) \mid term = term \mid \neg form \\ & \mid & form \wedge form \mid form \vee form \mid form \longrightarrow form \\ & \mid & \forall x.\ form \mid \exists x.\ form \end{array}$$

Examples

- $\neg\ A \wedge B \vee C \equiv ((\neg\ A) \wedge B) \vee C$

Formulae

Syntax (in decreasing priority):

$$\begin{array}{lcllll} form & ::= & (form) & | & term = term & | & \neg form \\ & | & form \wedge form & | & form \vee form & | & form \longrightarrow form \\ & | & \forall x.\ form & | & \exists x.\ form & & \end{array}$$

Examples

- $\neg\ A \wedge B \vee C \equiv ((\neg\ A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$

Formulae

Syntax (in decreasing priority):

$$\begin{array}{lcllll} form & ::= & (form) & | & term = term & | & \neg form \\ & | & form \wedge form & | & form \vee form & | & form \longrightarrow form \\ & | & \forall x.\ form & | & \exists x.\ form & & \end{array}$$

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$
- $\forall x.\ P\ x \wedge Q\ x \equiv \forall x.\ (P\ x \wedge Q\ x)$

Formulae

Syntax (in decreasing priority):

$$\begin{array}{rcllll} form & ::= & (form) & | & term = term & | & \neg form \\ & | & form \wedge form & | & form \vee form & | & form \longrightarrow form \\ & | & \forall x.\ form & | & \exists x.\ form & & \end{array}$$

Examples

- $\neg\ A \wedge B \vee C \equiv ((\neg\ A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$
- $\forall x.\ P\ x \wedge Q\ x \equiv \forall x.\ (P\ x \wedge Q\ x)$

Scope of quantifiers: as far to the right as possible

Formulae

Abbreviation: $\forall x\ y.\ P\ x\ y \equiv \forall x.\ \forall y.\ P\ x\ y$

Formulae

Abbreviation: $\forall x\ y.\ P\ x\ y \equiv \forall x.\ \forall y.\ P\ x\ y \qquad (\forall, \exists, \lambda, \ldots)$

Formulae

Abbreviation: $\forall x\ y.\ P\ x\ y \equiv \forall x.\ \forall y.\ P\ x\ y \qquad (\forall, \exists, \lambda, \ldots)$

Parentheses:

- $\wedge$, $\vee$ and $\longrightarrow$ associate to the right:
  $A \wedge B \wedge C \equiv A \wedge (B \wedge C)$

Formulae

Abbreviation: $\forall x\ y.\ P\ x\ y \equiv \forall x.\ \forall y.\ P\ x\ y \qquad (\forall, \exists, \lambda, \ldots)$

Parentheses:

- $\wedge$, $\vee$ and $\longrightarrow$ associate to the right:
  $A \wedge B \wedge C \equiv A \wedge (B \wedge C)$
- $A \longrightarrow B \longrightarrow C \equiv A \longrightarrow (B \longrightarrow C) \not\equiv (A \longrightarrow B) \longrightarrow C$!

Warning

Quantifiers have low priority and need to be parenthesized:

! $P \wedge \forall x.\ Q\ x \rightsquigarrow P \wedge (\forall x.\ Q\ x)$!

Types and Terms

Types

Syntax:

$$\tau \quad ::= \quad (\tau)$$
$$\quad\quad | \quad \mathit{bool} \mid \mathit{nat} \mid \ldots \qquad \text{base types}$$

Types

Syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \ldots & \text{base types} \\
 & | & \mathit{'a} \mid \mathit{'b} \mid \ldots & \text{type variables}
\end{array}
$$

Types

Syntax:

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & \mathit{bool} \mid \mathit{nat} \mid \ldots & \text{base types} \\
 & | & \mathit{'a} \mid \mathit{'b} \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{total functions}
\end{array}
$$

Types

Syntax:

τ	$::=$	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid \ldots$	base types
	$\mid$	*'a* $\mid$ *'b* $\mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)

Types

Syntax:

τ	$::=$	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid \ldots$	base types
	$\mid$	*'a* $\mid$ *'b* $\mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	τ *list*	lists

Types

Syntax:

τ	$::=$	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid \ldots$	base types
	$\mid$	*'a* $\mid$ *'b* $\mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	τ *list*	lists
	$\mid$	$\ldots$	user-defined types

Types

Syntax:

τ	$::=$	(τ)	
	$\mid$	*bool* $\mid$ *nat* $\mid \ldots$	base types
	$\mid$	*'a* $\mid$ *'b* $\mid \ldots$	type variables
	$\mid$	$\tau \Rightarrow \tau$	total functions
	$\mid$	$\tau \times \tau$	pairs (ascii: $*$)
	$\mid$	τ *list*	lists
	$\mid$	$\ldots$	user-defined types

Parentheses: *T1* $\Rightarrow$ *T2* $\Rightarrow$ *T3* $\equiv$ *T1* $\Rightarrow$ *(T2* $\Rightarrow$ *T3)*

Terms: Basic syntax

Syntax:

$term$	$::=$	$(term)$	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$term\ term$	function application
	$\mid$	$\lambda x.\ term$	function “abstraction”

Terms: Basic syntax

Syntax:

$term$	$::=$	$(term)$	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$term\ term$	function application
	$\mid$	$\lambda x.\ term$	function “abstraction”
	$\mid$	$\ldots$	lots of syntactic sugar

Terms: Basic syntax

Syntax:

$term$	$::=$	$(term)$	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$term\ term$	function application
	$\mid$	$\lambda x.\ term$	function “abstraction”
	$\mid$	$\ldots$	lots of syntactic sugar

Examples: *f (g x) y* *h (λx. f (g x))*

Terms: Basic syntax

Syntax:

$term$	$::=$	$(term)$	
	$\mid$	a	constant or variable (identifier)
	$\mid$	$term\ term$	function application
	$\mid$	$\lambda x.\ term$	function "abstraction"
	$\mid$	$\ldots$	lots of syntactic sugar

Examples: *f (g x) y* *h (λx. f (g x))*

Parantheses: $f\ a_1\ a_2\ a_3 \equiv ((f\ a_1)\ a_2)\ a_3$

λ-calculus on one slide

Informal notation: $t[x]$

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a
- *Function abstraction:*
  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a
- *Function abstraction:*
  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.
- *Computation:*
  Replace formal by actual parameter ("β-reduction"):
  $(\lambda x.t[x])\ a \quad \longrightarrow_\beta \quad t[a]$

λ-calculus on one slide

Informal notation: $t[x]$

- *Function application:*
  $f\ a$ is the call of function f with argument a
- *Function abstraction:*
  $\lambda x.t[x]$ is the function with formal parameter x and body/result $t[x]$, i.e. $x \mapsto t[x]$.
- *Computation:*
  Replace formal by actual parameter ("β-reduction"):
  $(\lambda x.t[x])\ a \quad \longrightarrow_\beta \quad t[a]$

Example: $(\lambda\ x.\ x + 5)\ 3 \quad \longrightarrow_\beta \quad (3 + 5)$

$\longrightarrow_\beta$ *in Isabelle: Don't worry, be happy*

Isabelle performs β-reduction automatically

Isabelle considers $(\lambda x.t[x])a$ and $t[a]$ equivalent

Terms and Types

Terms must be well-typed

(the argument of every function call must be of the right type)

Terms and Types

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation: $t :: \tau$ means t is a well-typed term of type τ.

Type inference

Isabelle automatically computes ("*infers*") the type of each variable in a term.

Type inference

Isabelle automatically computes ("*infers*") the type of each variable in a term.

In the presence of *overloaded* functions (functions with multiple types) not always possible.

Type inference

Isabelle automatically computes ("*infers*") the type of each variable in a term.

In the presence of *overloaded* functions (functions with multiple types) not always possible.

User can help with type annotations inside the term.

Example: *f (x::nat)*

Currying

Thou shalt curry your functions

Currying

Thou shalt curry your functions

- Curried: *f* :: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: *f'* :: $\tau_1 \times \tau_2 \Rightarrow \tau$

Currying

Thou shalt curry your functions

- Curried: *f* :: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: *f'* :: $\tau_1 \times \tau_2 \Rightarrow \tau$

Advantage: *partial application* *f* a_1 with a_1 :: τ_1

Terms: Syntactic sugar

Some predefined syntactic sugar:

- *Infix:* +, -, *, #, @, . . .
- *Mixfix:* *if _ then _ else _*, *case _ of*, . . .

Terms: Syntactic sugar

Some predefined syntactic sugar:

- *Infix:* +, -, *, #, @, ...
- *Mixfix:* *if _ then _ else _*, *case _ of*, ...

Prefix binds more strongly than infix:

! $f\ x + y \equiv (f\ x) + y \not\equiv f\ (x + y)$!

Terms: Syntactic sugar

Some predefined syntactic sugar:

- *Infix:* +, -, *, #, @, ...
- *Mixfix:* *if _ then _ else _*, *case _ of*, ...

Prefix binds more strongly than infix:

! $f\ x + y \equiv (f\ x) + y \not\equiv f\ (x + y)$!

Enclose *if* and *case* in parentheses:

! *(if _ then _ else _)* !

Base types: bool, nat, list

Type bool

Formulae = terms of type *bool*

Type bool

Formulae = terms of type *bool*

True :: bool
False :: bool
$\wedge, \vee, \ldots$ *:: bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
$\vdots$

Type bool

Formulae = terms of type *bool*

True :: bool
False :: bool
$\wedge, \vee, \ldots$ *:: bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
$\vdots$

if-and-only-if: *=*

Type nat

$0 :: nat$
$Suc :: nat \Rightarrow nat$
$+, *, ... :: nat \Rightarrow nat \Rightarrow nat$
$\vdots$

Type nat

0 :: nat
Suc :: nat ⇒ nat
*+, *, ... :: nat ⇒ nat ⇒ nat*
⋮

! Numbers and arithmetic operations are overloaded:
0,1,2,... :: 'a, + :: 'a ⇒ 'a ⇒ 'a

You need type annotations: *1 :: nat, x + (y::nat)*

Type nat

0 :: nat
Suc :: nat ⇒ nat
*+, *, ... :: nat ⇒ nat ⇒ nat*
⋮

! Numbers and arithmetic operations are overloaded:
0,1,2,... :: 'a, + :: 'a ⇒ 'a ⇒ 'a

You need type annotations: *1 :: nat, x + (y::nat)*

. . . unless the context is unambiguous: *Suc z*

Type list

- *[]*: empty list
- *x # xs*: list with first element *x* ("*head*") and rest *xs* ("*tail*")
- Syntactic sugar: $[x_1,\ldots,x_n]$

Type list

- *[]*: empty list
- *x # xs*: list with first element *x* ("*head*") and rest *xs* ("*tail*")
- Syntactic sugar: $[x_1,\ldots,x_n]$

Large library:
hd, *tl*, *map*, *length*, *filter*, *set*, *nth*, *take*, *drop*, *distinct*, ...

Don't reinvent, reuse!
$\leadsto$ `HOL/List.thy`

Isabelle Theories

Theory = Module

Syntax: `theory` $MyTh$
`imports` $ImpTh_1 \ldots ImpTh_n$
`begin`
(declarations, definitions, theorems, proofs, ...)*
`end`

- $MyTh$: name of theory. Must live in file $MyTh$`.thy`
- $ImpTh_i$: name of *imported* theories. Import transitive.

Theory = Module

Syntax:

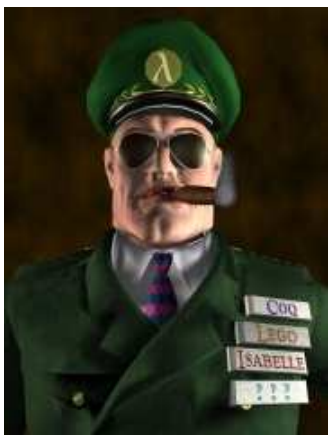
```
theory MyTh
imports ImpTh1 ... ImpThn
begin
(declarations, definitions, theorems, proofs, ...)*
end
```

- $MyTh$: name of theory. Must live in file $MyTh$`.thy`
- $ImpTh_i$: name of *imported* theories. Import transitive.

Usually:

```
theory MyTh
imports Main
⋮
```

Proof General



An Isabelle Interface

by David Aspinall

Proof General

Customized version of (x)emacs:

- all of emacs (info: `C-h i`)
- Isabelle aware (when editing `.thy` files)
- mathematical symbols (“x-symbols”)

X-Symbols

Input of funny symbols in Proof General

- via menu (“X-Symbol”)
- via ascii encoding (similar to LaTeX): `\<and>`, `\<or>`, ...
- via abbreviation: `/\`, `\/`, `-->`, ...

x-symbol	∀	∃	λ	¬	∧	∨	⟶	⇒
ascii (1)	`\<forall>`	`\<exists>`	`\<lambda>`	`\<not>`	`/\`	`\/`	`-->`	`=>`
ascii (2)	`ALL`	`EX`	`%`	`~`	`&`	|		

(1) is converted to x-symbol, (2) stays ascii.

Demo: terms and types

An introduction to recursion and induction

A recursive datatype: toy lists

datatype *'a list = Nil | Cons 'a ('a list)*

A recursive datatype: toy lists

datatype *'a list = Nil | Cons 'a ('a list)*

***Nil*:** empty list

***Cons x xs*:** head *x :: 'a*, tail *xs :: 'a list*

A recursive datatype: toy lists

datatype *'a list = Nil | Cons 'a ('a list)*

***Nil*:** empty list

***Cons x xs*:** head *x :: 'a*, tail *xs :: 'a list*

A toy list: *Cons False (Cons True Nil)*

A recursive datatype: toy lists

datatype *'a list = Nil | Cons 'a ('a list)*

***Nil*:** empty list

***Cons x xs*:** head *x :: 'a*, tail *xs :: 'a list*

A toy list: *Cons False (Cons True Nil)*

Predefined lists: *[False, True]*

Structural induction on lists

P xs holds for all lists *xs* if

Structural induction on lists

P xs holds for all lists *xs* if

- *P Nil*

Structural induction on lists

P xs holds for all lists *xs* if

- *P Nil*
- and for arbitrary *x* and *xs*, *P xs* implies *P (Cons x xs)*

A recursive function: append

Definition by *primitive recursion*:

primrec *app :: 'a list ⇒ 'a list ⇒ 'a list* **where**
app Nil ys = ? |
app (Cons x xs) ys = ??

A recursive function: append

Definition by *primitive recursion*:

primrec *app :: 'a list ⇒ 'a list ⇒ 'a list* **where**
app Nil ys = ? I
app (Cons x xs) ys = ??

1 rule per constructor
Recursive calls must drop the constructor $\Longrightarrow$ Termination

Concrete syntax

In `.thy` files:
Types and formulas need to be inclosed in "

Concrete syntax

In `.thy` files:
Types and formulas need to be inclosed in "

Except for single identifiers, e.g. *'a*

Concrete syntax

In `.thy` files:
Types and formulas need to be inclosed in "

Except for single identifiers, e.g. *'a*

" normally not shown on slides

Demo: append and reverse

Proofs

General schema:

lemma *name*`: "..."`
apply `(...)`
apply `(...)`
⋮
done

If the lemma is suitable as a simplification rule:

lemma *name*`[simp]: "..."`

Proof methods

- Structural induction
  - Format: *(induct x)*
    x must be a free variable in the first subgoal.
    The type of *x* must be a datatype.
  - Effect: generates 1 new subgoal per constructor
- Simplification and a bit of logic
  - Format: *auto*
  - Effect: tries to solve as many subgoals as possible using simplification and basic logical reasoning.

Top down proofs

Command

sorry

"completes" any proof.

Top down proofs

Command

sorry

"completes" any proof.

Allows top down development:

Assume lemma first, prove it later.

Some useful tools

Disproving tools

Automatic counterexample search by random testing:
quickcheck

Disproving tools

Automatic counterexample search by random testing:

quickcheck

Counterexample search via SAT solver:

nitpick

Finding theorems

1. Click on Find button
2. Input search pattern (e.g. *"_ & True"*)

Demo: Disproving and Finding

Isabelle's meta-logic

Basic constructs

Implication $\Longrightarrow$ (==>)

For separating premises and conclusion of theorems

Basic constructs

Implication $\Longrightarrow$ (**==>**)

For separating premises and conclusion of theorems

Equality $\equiv$ (**==**)

For definitions

Basic constructs

Implication $\Longrightarrow$ (**==>**)
For separating premises and conclusion of theorems

Equality $\equiv$ (**==**)
For definitions

Universal quantifier $\bigwedge$ (**! !**)
For binding local variables

Basic constructs

Implication $\Longrightarrow$ (**==>**)
For separating premises and conclusion of theorems

Equality $\equiv$ (**==**)
For definitions

Universal quantifier $\bigwedge$ (**! !**)
For binding local variables

Do not use *inside* HOL formulae

Notation

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

Notation

$$[\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

abbreviates

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$$

$;$ $\approx$ "and"

The proof state

$$1.\ \bigwedge x_1 \ldots x_p.\ [\![A_1; \ldots ; A_n]\!] \Longrightarrow B$$

$x_1 \ldots x_p$	Local constants
$A_1 \ldots A_n$	Local assumptions
B	Actual (sub)goal

Type and function definition in Isabelle/HOL

Type definition in Isabelle/HOL

Introducing new types

Keywords:

- **typedecl**: pure declaration
- **types**: abbreviation
- **datatype**: recursive datatype

typedecl

typedecl $name$

Introduces new “opaque” type $name$ without definition

typedecl

typedecl $name$

Introduces new "opaque" type $name$ without definition

Example:

typedecl *addr* — An abstract type of addresses

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types
name = string
('a,'b)foo = 'a list $\times$ *'b list*

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

types
name = string
('a,'b)foo = 'a list $\times$ *'b list*

Type abbreviations are expanded immediately after parsing
Not present in internal representation and Isabelle output

datatype

The example

datatype *'a list = Nil | Cons 'a ('a list)*

Properties:

- Types: *Nil :: 'a list*
  Cons :: 'a ⇒ 'a list ⇒ 'a list
- Distinctness: *Nil ≠ Cons x xs*
- Injectivity: *(Cons x xs = Cons y ys) = (x = y ∧ xs = ys)*

The general case

datatype $(\alpha_1, \ldots, \alpha_n)\tau$ = $C_1\ \tau_{1,1} \ldots \tau_{1,n_1}$
| $\ldots$
| $C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$
- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

The general case

datatype $(\alpha_1, \ldots, \alpha_n)\tau$ = $C_1\ \tau_{1,1} \ldots \tau_{1,n_1}$
| ...
| $C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$
- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied automatically
Induction must be applied explicitly

Function definition in Isabelle/HOL

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow 0 = 1$$

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow 0 = 1$$

! All functions in HOL must be total !

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **fun**
  Automatic termination proof

Function definition schemas in Isabelle/HOL

- Non-recursive with **definition**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **fun**
  Automatic termination proof
- Well-founded recursion with **function**
  User-supplied termination proof

definition

Definition (non-recursive) by example

definition *sq :: nat ⇒ nat* **where** *sq n = n * n*

Definitions: pitfalls

definition *prime :: nat ⇒ bool* **where**
prime p = (1 < p ∧ (m dvd p ⟶ m = 1 ∨ m = p))

Definitions: pitfalls

definition *prime :: nat ⇒ bool* **where**
prime p = (1 < p ∧ (m dvd p ⟶ m = 1 ∨ m = p))

Not a definition: free *m* not on left-hand side

Definitions: pitfalls

definition *prime :: nat ⇒ bool* **where**
prime p = (1 < p ∧ (m dvd p ⟶ m = 1 ∨ m = p))

Not a definition: free *m* not on left-hand side

! Every free variable on the rhs must occur on the lhs !

Definitions: pitfalls

definition *prime :: nat ⇒ bool* **where**
prime p = (1 < p ∧ (m dvd p ⟶ m = 1 ∨ m = p))

Not a definition: free *m* not on left-hand side

! Every free variable on the rhs must occur on the lhs !

prime p = (1 < p ∧ (∀ m. m dvd p ⟶ m = 1 ∨ m = p))

Using definitions

Definitions are not used automatically

Using definitions

Definitions are not used automatically

Unfolding the definition of *sq*:

apply*(unfold sq_def)*

primrec

The example

primrec *app :: 'a list ⇒ 'a list ⇒ 'a list* **where**

app Nil ys = ys

app (Cons x xs) ys = Cons x (app xs ys)

The general case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then $f :: \cdots \Rightarrow \tau \Rightarrow \cdots \Rightarrow \tau'$ can be defined by *primitive recursion*:

$$
\begin{array}{lcl}
f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_p & = & r_1 \mid \\
\vdots & & \\
f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_p & = & r_k
\end{array}
$$

The general case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then $f :: \cdots \Rightarrow \tau \Rightarrow \cdots \Rightarrow \tau'$ can be defined by *primitive recursion*:

$$
\begin{array}{lcl}
f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_p & = & r_1 \mid \\
\vdots & & \\
f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_p & = & r_k
\end{array}
$$

The recursive calls in r_i must be *structurally smaller*, i.e. of the form $f\ a_1 \ldots y_{i,j} \ldots a_p$

nat is a datatype

datatype *nat = 0 | Suc nat*

nat is a datatype

datatype *nat = 0 | Suc nat*

Functions on *nat* definable by primrec!

primrec *f :: nat ⇒ ...*
f 0 = ...
f(Suc n) = ... f n ...

More predefined types and functions

Type option

datatype *'a option = None | Some 'a*

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \approx \text{partial function:}$$

$$\textit{None} \approx \text{no result}$$

$$\textit{Some a} \approx \text{result } a$$

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \approx \text{partial function:}$$

$$\textit{None} \approx \text{no result}$$
$$\textit{Some a} \approx \text{result } a$$

Example:

primrec *lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option* **where**

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \approx \text{partial function:}$$

$$\textit{None} \approx \text{no result}$$
$$\textit{Some a} \approx \text{result } a$$

Example:

primrec *lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option* **where**
lookup k [] = None

Type option

datatype *'a option = None | Some 'a*

Important application:

$$\ldots \Rightarrow \textit{'a option} \approx \text{partial function:}$$

$$\textit{None} \approx \text{no result}$$
$$\textit{Some a} \approx \text{result } a$$

Example:

primrec *lookup :: 'k ⇒ ('k × 'v) list ⇒ 'v option* **where**
lookup k [] = None |
lookup k (x#xs) =
(if fst x = k then Some(snd x) else lookup k xs)

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

Wildcards:

(case xs of [] ⇒ [] | y#_ ⇒ [y])

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

Wildcards:

(case xs of [] ⇒ [] | y#_ ⇒ [y])

Nested patterns:

(case xs of [0] ⇒ 0 | [Suc n] ⇒ n | _ ⇒ 2)

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

Wildcards:

(case xs of [] ⇒ [] | y#_ ⇒ [y])

Nested patterns:

(case xs of [0] ⇒ 0 | [Suc n] ⇒ n | _ ⇒ 2)

Complicated patterns mean complicated proofs!

case

Datatype values can be taken apart with *case* expressions:

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

Wildcards:

(case xs of [] ⇒ [] | y#_ ⇒ [y])

Nested patterns:

(case xs of [0] ⇒ 0 | [Suc n] ⇒ n | _ ⇒ 2)

Complicated patterns mean complicated proofs!

Needs *( )* in context

Proof by case distinction

If $t :: \tau$ and τ is a datatype

apply(*case_tac* t)

Proof by case distinction

If $t :: \tau$ and τ is a datatype

apply(*case_tac* t)

creates k subgoals

$$t = C_i\ x_1 \ldots x_p \Longrightarrow \ldots$$

one for each constructor C_i of type τ.

Demo: trees

fun

From primitive recursion to arbitrary pattern matching

Example: Fibonacchi

fun *fib :: nat* $\Rightarrow$ *nat* **where**

fib 0 = 0 |
fib (Suc 0) = 1 |
fib (Suc(Suc n)) = fib (n+1) + fib n

Example: Separation

fun *sep :: 'a ⇒ 'a list ⇒ 'a list* **where**

sep a [] = [] |
sep a [x] = [x] |
sep a (x#y#zs) = x # a # sep a (y#zs)

Example: Ackermann

fun *ack :: nat ⇒ nat ⇒ nat* **where**

```
ack 0       n       = Suc n  |
ack (Suc m) 0       = ack m (Suc 0)  |
ack (Suc m) (Suc n) = ack m (ack (Suc m) n)
```

Key features of fun

- Arbitrary pattern matching

Key features of fun

- Arbitrary pattern matching
- Order of equations matters

Key features of fun

- Arbitrary pattern matching
- Order of equations matters
- Termination must be provable
  by lexicographic combination of size measures

Size

- *size(n::nat) = n*

Size

- *size(n::nat) = n*
- *size(xs) = length xs*

Size

- *size(n::nat) = n*
- *size(xs) = length xs*
- *size* counts number of (non-nullary) constructors

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Similar for tuples:

$$(5,6,3) > (4,12,5) > (4,11,9) > (4,11,8) > \cdots$$

Lexicographic ordering

Either the first component decreases, or it stays unchanged and the second component decreases:

$$(5,3) > (4,7) > (4,6) > (4,0) > (3,42) > \cdots$$

Similar for tuples:

$$(5,6,3) > (4,12,5) > (4,11,9) > (4,11,8) > \cdots$$

Theorem If each component ordering terminates, then their *lexicographic product* terminates, too.

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

because the arguments of each recursive call are lexicographically smaller than the arguments on the lhs.

Ackermann terminates

ack 0 n = Suc n

ack (Suc m) 0 = ack m (Suc 0)

ack (Suc m) (Suc n) = ack m (ack (Suc m) n)

because the arguments of each recursive call are lexicographically smaller than the arguments on the lhs.

Note: order of arguments not important for Isabelle!

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

> for each equation $f(e) = t$,
> prove $P(e)$ assuming $P(r)$ for all recursive calls $f(r)$ in t.

Computation Induction

If $f :: \tau \Rightarrow \tau'$ is defined by **fun**, a special induction schema is provided to prove $P(x)$ for all $x :: \tau$:

> for each equation $f(e) = t$,
> prove $P(e)$ assuming $P(r)$ for all recursive calls $f(r)$ in t.

Induction follows course of (terminating!) computation

Computation Induction: Example

fun *div2 :: nat ⇒ nat* **where**
div2 0 = 0 |
div2 (Suc 0) = 0 |
div2(Suc(Suc n)) = Suc(div2 n)

Computation Induction: Example

fun *div2 :: nat ⇒ nat* **where**
div2 0 = 0 |
div2 (Suc 0) = 0 |
div2(Suc(Suc n)) = Suc(div2 n)

⇝ induction rule `div2.induct`:

$$\frac{P(0) \quad P(Suc\ 0) \quad P(n) \Longrightarrow P(Suc(Suc\ n))}{P(m)}$$

Demo: fun