What the course is about

Chapter 1

Introduction

Data Structures and Algorithms
for Functional Programming Languages
What the course is about

Data Structures and Algorithms for Functional Programming Languages

The code is not enough!

Formal Correctness and Complexity Proofs with the Proof Assistant *Isabelle*

---

Proof Assistants

- You give the structure of the proof
- The PA checks the correctness of each step

---

Proof Assistants

- You give the structure of the proof
- The PA checks the correctness of each step

Government health warnings:

*Time consuming*
Proof Assistants

- You give the structure of the proof
- The PA checks the correctness of each step

Government health warnings:

- Time consuming
- Potentially addictive

Two landmark verifications

C compiler
Two landmark verifications

C compiler
Competitive with gcc -01

Operating system
microkernel (L4)

Xavier Leroy
INRIA Paris
using Coq

Gerwin Klein (& Co)
NICTA Sydney
using Isabelle

Overview of course

- Week 1–5: Introduction to Isabelle

Overview of course

- Week 1–5: Introduction to Isabelle
- Rest of semester: Search trees, priority queues, etc and their (amortized) complexity

What we expect from you

Functional programming experience with an ML/Haskell-like language
What we expect from you

Functional programming experience with an ML/Haskell-like language

First course in data structures and algorithms

First course in discrete mathematics

You will not survive this course without doing the time-consuming homework

Part I
Isabelle

Chapter 2
Programming and Proving

Notation

Implication associates to the right:

\[ A \implies B \implies C \] means \[ A \implies (B \implies C) \]
Notation

Implication associates to the right:

\[ A \implies B \implies C \quad \text{means} \quad A \implies (B \implies C) \]

Similarly for other arrows: \( \Rightarrow, \rightarrow \)

\[
\frac{A_1 \ldots A_n}{\text{means} \quad A_1 \implies \ldots \implies A_n \implies B}
\]

HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has
- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!
HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has
- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:
- For the moment: only \( \text{term} = \text{term} \)

HOL = Higher-Order Logic
HOL = Functional Programming + Logic

HOL has
- datatypes
- recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too!

HOL Formulas:
- For the moment: only \( \text{term} = \text{term} \),
e.g. \( 1 + 2 = 4 \)

Types

Basic syntax:

\[
\tau ::= (\tau) \\
| \text{bool} | \text{nat} | \text{int} | \ldots \quad \text{base types} \\
| \ 'a \ | \ 'b \ | \ldots \quad \text{type variables}
\]

Types

Basic syntax:

\[
\tau ::= (\tau) \\
| \text{bool} | \text{nat} | \text{int} | \ldots \quad \text{base types} \\
| \ 'a \ | \ 'b \ | \ldots \quad \text{type variables} \\
| \tau \Rightarrow \tau \quad \text{functions}
\]
Types

Basic syntax:

\[ \tau ::= (\tau) \mid \text{bool} \mid \text{nat} \mid \text{int} \mid \ldots \] base types
\[ \mid 'a \mid 'b \mid \ldots \] type variables
\[ \mid \tau \Rightarrow \tau \] functions
\[ \mid \tau \times \tau \] pairs (ascii: *)

Terms

Basic syntax:

\[ t ::= (t) \mid a \] constant or variable (identifier)
Terms

Basic syntax:

\[
\begin{align*}
  t & ::= (t) \\
   & | a \quad \text{constant or variable (identifier)} \\
   & | t \ t \quad \text{function application} \\
   & | \lambda x. \ t \quad \text{function abstraction} \\
   & | \ldots \quad \text{lots of syntactic sugar}
\end{align*}
\]

Terms

Terms must be well-typed

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

Notation:

\[
t : : \tau \quad \text{means \ "t is a well-typed term of type \ \tau".}
\]
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 \( \tau \)”.

\[
\frac{t :: \tau_1 \Rightarrow \tau_2 \quad u :: \tau_1}{t \ u :: \tau_2}
\]

Type inference

Isabelle automatically computes the type of each variable in a term.

Type inference

Isabelle automatically computes the type of each variable in a term. This is called \textit{type inference}.

In the presence of \textit{overloaded} functions (functions with multiple types) this is not always possible.

Type inference

In the presence of \textit{overloaded} functions (functions with multiple types) this is not always possible.

User can help with \textit{type annotations} inside the term.

Example: \( f \ (x :: \text{nat}) \)
Currying

Thou shalt Curry your functions

- Curried: \( f :: \tau_1 \Rightarrow \tau_2 \Rightarrow \tau \)
- Tupled: \( f' :: \tau_1 \times \tau_2 \Rightarrow \tau \)

Predefined syntactic sugar

- Infix: \(+, - , \ast , \# , @, \ldots\)
- Mixfix: \( if \_ then \_ else \_ , case \_ of , \ldots\)

Prefix binds more strongly than infix:

\[ f \; x \; y \equiv (f \; x) \; y \not\equiv f \; (x \; y) \]
Predefined syntactic sugar

- **Infix:** +, −, *, #, ⊙, ...
- **Mixfix:** if then else, case of, ...

Prefix binds more strongly than infix:

\[
\text{! } f x + y \equiv (f x) + y \neq f (x + y) \text{ !}
\]

Enclose if and case in parentheses:

\[
\text{! } (\text{if } \text{ then } \text{ else }) \text{ !}
\]

Theory = Isabelle Module

**Syntax:**

```plaintext
theory MyTh
imports T1 ... Tn
begin
(definitions, theorems, proofs, ...)*
end
```

*MyTh*: name of theory. Must live in file `MyTh.thy`

*Ti*: names of imported theories. Import transitive.
Theory = Isabelle Module

Syntax:  
theory \texttt{MyTh}
imports \texttt{T_1} \ldots \texttt{T_n}
begin
  (definitions, theorems, proofs, ...)*
end

\texttt{MyTh}: name of theory. Must live in file \texttt{MyTh.thy}

\texttt{T_i}: names of imported theories. Import transitive.

Usually: \texttt{imports Main}

Concrete syntax

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

isabelle jedit

- Based on \textit{jEdit} editor
- Processes Isabelle text automatically when editing .thy files
Overview_Demo.thy

Type `bool`

datatype `bool = True | False`

Predefined functions:
`∧, ∨, →, ... :: bool ⇒ bool ⇒ bool`
Type `bool`

**datatype** `bool` = `True` | `False`

Predefined functions:

∧, ∨, →, ... :: `bool` ⇒ `bool` ⇒ `bool`

A formula is a term of type `bool`

A formula is a term of type `bool`

if-and-only-if: =

Type `nat`

**datatype** `nat` = 0 | Suc `nat`

Values of type `nat`: 0, Suc 0, Suc(Suc 0), ...
**Type nat**

**datatype** nat = 0 | Suc nat

Values of type nat: 0, Suc 0, Suc(Suc 0), ...

Predefined functions: +, *, ... :: 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**

**datatype** nat = 0 | Suc nat

Values of type nat: 0, Suc 0, Suc(Suc 0), ...

Predefined functions: +, *, ... :: 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**

**datatype** nat = 0 | Suc nat

Values of type nat: 0, Suc 0, Suc(Suc 0), ...

Predefined functions: +, *, ... :: 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**

**datatype** nat = 0 | Suc nat

Values of type nat: 0, Suc 0, Suc(Suc 0), ...

Predefined functions: +, *, ... :: 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
An informal proof

**Lemma** \( \text{add} \ m \ 0 = m \)

**Proof** by induction on \( m \).

- Case 0 (the base case):
  \( \text{add} \ 0 \ 0 = 0 \) holds by definition of \( \text{add} \).
An informal proof

Lemma \( \text{add} \ m \ 0 = m \)
Proof by induction on \( m \).
  
  • Case 0 (the base case):
    \( \text{add} \ 0 \ 0 = 0 \) holds by definition of \( \text{add} \).
  
  • Case \( \text{Suc} \ m \) (the induction step):
    We assume \( \text{add} \ m \ 0 = m \),
    the induction hypothesis (IH).
    We need to show \( \text{add} \ (\text{Suc} \ m) \ 0 = \text{Suc} \ m \).

An informal proof

Lemma \( \text{add} \ m \ 0 = m \)
Proof by induction on \( m \).
  
  • Case 0 (the base case):
    \( \text{add} \ 0 \ 0 = 0 \) holds by definition of \( \text{add} \).
  
  • Case \( \text{Suc} \ m \) (the induction step):
    We assume \( \text{add} \ m \ 0 = m \),
    the induction hypothesis (IH).
    We need to show \( \text{add} \ (\text{Suc} \ m) \ 0 = \text{Suc} \ m \).
    The proof is as follows:
    \[
    \text{add} \ (\text{Suc} \ m) \ 0 = \text{Suc} \ (\text{add} \ m \ 0) \quad \text{by def. of add}
    \]
    \[
    = \text{Suc} \ m \quad \text{by IH}
    \]

An informal proof

Lemma \( \text{add} \ m \ 0 = m \)
Proof by induction on \( m \).
  
  • Case 0 (the base case):
    \( \text{add} \ 0 \ 0 = 0 \) holds by definition of \( \text{add} \).
  
  • Case \( \text{Suc} \ m \) (the induction step):
    We assume \( \text{add} \ m \ 0 = m \),
    the induction hypothesis (IH).
    We need to show \( \text{add} \ (\text{Suc} \ m) \ 0 = \text{Suc} \ m \).
    The proof is as follows:
    \[
    \text{add} \ (\text{Suc} \ m) \ 0 = \text{Suc} \ (\text{add} \ m \ 0) \quad \text{by def. of add}
    \]
    \[
    = \text{Suc} \ m \quad \text{by IH}
    \]

Type `a list

Lists of elements of type `a
Type 'a list

Lists of elements of type 'a

```plaintext
datatype 'a list = Nil | Cons 'a ('a list)
```

Some lists: `Nil`, `Cons 1 Nil`, `Cons 1 (Cons 2 Nil)`, ...

Syntactic sugar:
- `[]` = `Nil`: empty list
- `x # xs` = `Cons x xs`: list with first element `x` ("head") and rest `xs` ("tail")
Type 'a list

Lists of elements of type 'a

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

Some lists: Nil, Cons 1 Nil, Cons 1 (Cons 2 Nil), ...

Syntactic sugar:
- [] = Nil: empty list
- x # xs = Cons x xs: list with first element x ("head") and rest xs ("tail")
- [x₁, ..., xₙ] = x₁ # ... # xₙ ≠ []

Structural Induction for lists

To prove that P(xs) for all lists xs, prove
- P([]) and
- for arbitrary but fixed x and xs, P(xs) implies P(x#xs).

\[
\begin{align*}
&P(\text{[]}) \land \forall x \, . \, P(xs) \Rightarrow P(x#xs) \\
&P(xs)
\end{align*}
\]
List_Demo.thy