Title: FDS (28.04.2017)
Date: Fri Apr 28 08:29:57 CEST 2017
Duration: 92:30 min
Pages: 109

Functional Data Structures
with Isabelle/HOL

Tobias Nipkow
Fakultät für Informatik
Technische Universität München

2017-4-28
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

Terminology

Formal = machine-checked
Verification = formal correctness proof

Two landmark verifications

C compiler
Two landmark verifications

C compiler
Competitive with gcc -01

Xavier Leroy
INRIA Paris
using Coq

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
- 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

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

Which of the following formulas have the same meaning?

1. \( A \implies (B \implies C) \)
2. \((A \implies B) \implies C\)
3. \((A \land B) \implies C\)

Notation

Implication associates to the right:

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

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

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

Notation

Implication associates to the right:

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

1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction Heuristics
4. Simplification
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 $term = term$
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 $term = term$,
  e.g. $1 + 2 = 4$
- Later: $\land, \lor, \rightarrow, \forall, \ldots$

Overview of Isabelle/HOL
Types and terms
Interface
By example: types bool, nat and list
Numeric Types
Summary

Types

Basic syntax:

$$\tau ::= (\tau) \quad | \quad bool \quad | \quad nat \quad | \quad int \quad | \ldots$$

Base types

Types

Basic syntax:

$$\tau ::= (\tau) \quad | \quad bool \quad | \quad nat \quad | \quad int \quad | \ldots$$

Base types

$$\tau ::= 'a \quad | \quad 'b \quad | \ldots$$

Type variables
Types

Basic syntax:

\[ \tau ::= (\tau) \mid \text{base types} \]
\[ \mid \text{type variables} \]
\[ \mid \text{functions} \]
\[ \mid \tau \Rightarrow \tau \mid \tau \times \tau \]

Types

Basic syntax:

\[ \tau ::= (\tau) \mid \text{base types} \]
\[ \mid \text{type variables} \]
\[ \mid \text{functions} \]
\[ \mid \tau \Rightarrow \tau \mid \tau \times \tau \]

Types

Basic syntax:

\[ \tau ::= (\tau) \mid \text{base types} \]
\[ \mid \text{type variables} \]
\[ \mid \text{functions} \]
\[ \mid \tau \Rightarrow \tau \mid \tau \times \tau \]

Types

Basic syntax:

\[ \tau ::= (\tau) \mid \text{base types} \]
\[ \mid \text{type variables} \]
\[ \mid \text{functions} \]
\[ \mid \tau \Rightarrow \tau \mid \tau \times \tau \]

Basic syntax:

\[ \tau ::= (\tau) \mid \text{base types} \]
\[ \mid \text{type variables} \]
\[ \mid \text{functions} \]
\[ \mid \tau \Rightarrow \tau \mid \tau \times \tau \]

Basic syntax:

\[ \tau ::= (\tau) \mid \text{base types} \]
\[ \mid \text{type variables} \]
\[ \mid \text{functions} \]
\[ \mid \tau \Rightarrow \tau \mid \tau \times \tau \]

Basic syntax:

\[ \tau ::= (\tau) \mid \text{base types} \]
\[ \mid \text{type variables} \]
\[ \mid \text{functions} \]
\[ \mid \tau \Rightarrow \tau \mid \tau \times \tau \]
Types

Basic syntax:

\[ \begin{align*} 
\tau & ::= (\tau) \\
& | \text{bool} | \text{nat} | \text{int} | \ldots \\
& | 'a | 'b | \ldots \\
& | \tau \Rightarrow \tau \\
& | \tau \times \tau \\
& | \tau \text{ list} \\
& | \tau \text{ set} \\
& | \ldots \\
\end{align*} \]

base types

type variables

functions

pairs (ascii: \(*\))

lists

sets

user-defined types

Terms

Basic syntax:

\[ \begin{align*} 
t & ::= (t) \\
& | a \\
& | tt \\
& | \lambda x. t \\
& | \ldots \\
\end{align*} \]

constant or variable (identifier)

function application

function abstraction

lots of syntactic sugar

Terms

Basic syntax:

\[ \begin{align*} 
t & ::= (t) \\
& | a \\
& | tt \\
& | \lambda x. t \\
& | \ldots \\
\end{align*} \]

constant or variable (identifier)

function application

function abstraction

lots of syntactic sugar

\[ \lambda \text{-calculus} \]
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.

In the presence of overloaded functions (functions with multiple types) this is not always possible.
Type inference

Isabelle automatically computes the type of each variable in a term. This is called *type inference*.

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

User can help with *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*: \(+, -, *, #, @, \ldots\)

Predefined syntactic sugar

- *Infix*: \(+, -, *, #, @, \ldots\)
- *Mixfix*: \( \text{if then else case of} \)
Predefined syntactic sugar

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

Prefix binds more strongly than infix:

\[ f \ x + y \equiv (f \ x) + y \neq f (x + y) \]

Enclose if and case in parentheses:

\[ (if \ then \ else) \]
Theory = Isabelle Module

Syntax:    theory MyTh
    imports T_1 ... T_n
    begin
    (definitions, theorems, proofs, ...)*
    end

*MyTh*: name of theory. Must live in file *MyTh.thy*
*T_i*: names of imported theories. Import transitive.

Usually: imports Main

Concrete syntax

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

Syntax:    theory MyTh
    imports T_1 ... T_n
    begin
    (definitions, theorems, proofs, ...)*
    end
Theory = Isabelle Module

Syntax: theory MyTh
        imports T_1 ... T_n
        begin
        (definitions, theorems, proofs, ...)*
        end

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

T_i: names of imported theories. Import transitive.

Usually: imports Main

Concrete syntax

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

Except for single identifiers
"

normally not shown on slides

isabelle jedit

- Based on jEdit editor
- Processes Isabelle text automatically when editing .thy files

Overview_Demo.thy
Overview_Demo.thy

\textbf{Type bool}

\texttt{datatype bool = True | False}

\textbf{Type bool}

\texttt{datatype bool = True | False}

Predefined functions:
\(\land, \lor, \rightarrow, \ldots \:: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}\)

Predefined functions:
\(\land, \lor, \rightarrow, \ldots \:: \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}\)

\textit{A formula is a term of type bool}
**Type bool**

\[
datatype \text{ bool } = \text{ True } | \text{ False }
\]

Predefined functions:
\[ \land, \lor, \rightarrow, \ldots : \text{ bool } \Rightarrow \text{ bool } \Rightarrow \text{ bool } \]

A formula is a term of type bool

if-and-only-if: =

---

**Type nat**

\[
datatype \text{ nat } = 0 | \text{ Suc nat }
\]

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

---

**Type nat**

\[
datatype \text{ nat } = 0 | \text{ Suc nat }
\]

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

Predefined functions: +, *, ... : nat \Rightarrow nat \Rightarrow 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)`

---

```
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)`
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:
  $\text{add } (\text{Suc } m) \ 0 = \text{Suc } (\text{add } m \ 0)$ by def. of $\text{add}$.
Type `'a list

Lists of elements of type `'a

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

Some lists: `Nil,
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 \( P(x#xs) \).

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

Included in Main.

Don't reinvent, reuse!

1 Overview of Isabelle/HOL
   Types and terms
   Interface
   By example: types bool, nat and list
   Numeric Types
   Summary

Numeric types: nat, int, real

Need conversion functions (inclusions):

\[
\begin{align*}
\text{int} & : \text{nat} \Rightarrow \text{int} \\
\text{real} & : \text{nat} \Rightarrow \text{real} \\
\text{real_of_int} & : \text{int} \Rightarrow \text{real}
\end{align*}
\]
Numeric types: \textit{nat}, \textit{int}, \textit{real}

Need conversion functions (inclusions):

\begin{itemize}
  \item \texttt{int} :: \textit{nat} \Rightarrow \textit{int}
  \item \texttt{real} :: \textit{nat} \Rightarrow \textit{real}
  \item \texttt{real_of_int} :: \textit{int} \Rightarrow \textit{real}
\end{itemize}

If you need type \textit{real},
import theory \textit{Complex_Main} instead of \textit{Main}

Isabelle inserts conversion functions automatically

\begin{itemize}
  \item (with theory \textit{Complex_Main})
  \item If there are multiple correct completions,
  \item Isabelle chooses an \textit{arbitrary} one
\end{itemize}
**Numeric types:** nat, int, real

Isabelle inserts conversion functions automatically
(with theory Complex_Main)
If there are multiple correct completions,
Isabelle chooses an arbitrary one

**Examples**

\[(i::int) + (n::nat) \sim i + \text{int } n\]

\[(n::nat) + n) :: \text{real} \sim \text{real}(n+n), \text{real } n + \text{real } n\]
### Numeric types: \( nat, int, real \)

Coercion in the other direction:

\[ nat :: int \Rightarrow nat \]

### Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  \[ +, -, \cdot :: 'a \Rightarrow 'a \Rightarrow 'a \]
  \[ - :: 'a \Rightarrow 'a \]
- Division on \( nat \) and \( int \):
  \[ \text{div}, \text{mod} :: 'a \Rightarrow 'a \Rightarrow 'a \]
- Division on \( real \):
  \[ / :: 'a \Rightarrow 'a \Rightarrow 'a \]

### Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  \[ +, -, \cdot :: 'a \Rightarrow 'a \Rightarrow 'a \]
  \[ - :: 'a \Rightarrow 'a \]
- Division on \( nat \) and \( int \):
  \[ \text{div}, \text{mod} :: 'a \Rightarrow 'a \Rightarrow 'a \]
- Division on \( real \):
  \[ / :: 'a \Rightarrow 'a \Rightarrow 'a \]
- Exponentiation with \( nat \):
  \[ ^ :: 'a \Rightarrow nat \Rightarrow 'a \]
Overloaded arithmetic operations

- Basic arithmetic functions are overloaded:
  
  \[
  +, -, *, :: 'a \Rightarrow 'a \Rightarrow 'a
  
  - :: 'a \Rightarrow 'a
  \]

- Division on \textit{nat} and \textit{int}:
  
  \[
  \text{div, mod} :: 'a \Rightarrow 'a \Rightarrow 'a
  \]

- Division on \textit{real}:
  
  \[
  / :: 'a \Rightarrow 'a \Rightarrow 'a
  \]

- Exponentiation with \textit{nat}:
  
  \[
  ^ :: 'a \Rightarrow \text{nat} \Rightarrow 'a
  \]

- Exponentiation with \textit{real}:
  
  \[
  \text{powr} :: 'a \Rightarrow 'a \Rightarrow 'a
  \]

- Absolute value:
  
  \[
  \text{abs} :: 'a \Rightarrow 'a
  \]

---

Overview of Isabelle/HOL

Types and terms

Interface

By example: types \textit{bool}, \textit{nat} and \textit{list}

Numeric Types

Summary

- \textbf{datatype} defines (possibly) recursive data types.

- \textbf{fun} defines (possibly) recursive functions by pattern-matching over \textbf{datatype} constructors.
Proof methods

- \textit{induction} performs structural induction on some variable (if the type of the variable is a datatype).

- \textit{auto} solves as many subgoals as it can, mainly by simplification (symbolic evaluation):

\begin{quote}
\texttt{``=`` is used only from left to right!}
\end{quote}

Proofs

General schema:

\begin{verbatim}
lemma name: "..."
apply (...) apply (...)
:
done
\end{verbatim}
Proofs

General schema:

\textbf{lemma} \textit{name}: "..."
apply (...) apply (...):
done

If the lemma is suitable as a simplification rule:
\textbf{lemma} \textit{name}[simp]: "..."

Top down proofs

Command

\textbf{sorry}

"completes" any proof.

The proof state

1. \( \bigwedge x_1 \ldots x_n. \quad A \Rightarrow B \)

Multiple assumptions

\([ A_1; \ldots ; A_n ] \Rightarrow B \)

abbreviates

\( A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B \)
Multiple assumptions

\[ [ A_1; \ldots ; A_n ] \implies B \]

abbreviates

\[ A_1 \implies \ldots \implies A_n \implies B \]

; \approx \text{“and”}

1. Overview of Isabelle/HOL
2. Type and function definitions
3. Induction Heuristics
4. Simplification