COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Simon Winwood, Toby Murray, June Andronick, Gerwin Klein

\[ a = b = c = \ldots \]
Intro & motivation, getting started with Isabelle

Foundations & Principles
- Lambda Calculus
- Higher Order Logic, natural deduction
- Term rewriting

Proof & Specification Techniques
- Inductively defined sets, rule induction
- Datatypes, recursion, induction
- **Calculational reasoning**
- Hoare logic, proofs about programs
- Locales, Presentation
Last time ...

- fun, function
- Well founded recursion
DEMO
MORE FUN
CALCULATIONAL REASONING
The Goal

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]
\[ \ldots = 1 \]
The Goal

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]

\[ \ldots = 1 \cdot x \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]

\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]

\[ \ldots = 1 \]

Can we do this in Isabelle?
The Goal

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]

\[ \ldots = 1 \cdot x \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]

\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]

\[ \ldots = 1 \]

Can we do this in Isabelle?

→ Simplifier: too eager
The Goal

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]
\[ \ldots = 1 \]

Can we do this in Isabelle?

- Simplifier: too eager
- Manual: difficult in apply style
The Goal

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]

\[ \ldots = 1 \cdot x \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]

\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]

\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]

\[ \ldots = 1 \]

Can we do this in Isabelle?

➔ Simplifier: too eager

➔ Manual: difficult in apply style

➔ Isar: with the methods we know, too verbose
Chains of equations

The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of =
Chains of equations

The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of =

Each step usually nontrivial (requires own subproof)
Chains of equations

The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of =

Each step usually nontrivial (requires own subproof)

Solution in Isar:

➔ Keywords also and finally to delimit steps
Chains of equations

The Problem

\[ \begin{align*}
  a &= b \\
  \cdots &= c \\
  \cdots &= d
\end{align*} \]

shows \( a = d \) by transitivity of =

Each step usually nontrivial (requires own subproof)

Solution in Isar:

\[ \Rightarrow \ \text{Keywords \textbf{also} and \textbf{finally} to delimit steps} \]
\[ \Rightarrow \ \cdots : \text{predefined schematic term variable,} \]
\[ \text{refers to right hand side of last expression} \]
Chains of equations

The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of =

Each step usually nontrivial (requires own subproof)

Solution in Isar:

➔ Keywords \textbf{also} and \textbf{finally} to delimit steps
➔ \ldots: predefined schematic term variable, refers to right hand side of last expression
➔ Automatic use of transitivity rules to connect steps
also / finally

have "t_0 = t_1" [proof]
also
also/finally

have \( t_0 = t_1 \) [proof]

also

calculation register

\( t_0 = t_1 \)
also/finally

have \( t_0 = t_1 \) [proof]
also
have \( \ldots = t_2 \) [proof]
also / finally

**have** "\(t_0 = t_1\)" [proof]

**also**

**have** "\(\ldots = t_2\)" [proof]

**also**

**calculation register**

"\(t_0 = t_1\)"

"\(t_0 = t_2\)"
also

have "\( t_0 = t_1 \)" [proof]

also

have "\( \ldots = t_2 \)" [proof]

also

: 

also

also
also/finally

have "\( t_0 = t_1 \)" [proof]
also
have "\( \ldots = t_2 \)" [proof]
also
:  
also
have "\( \ldots = t_n \)" [proof]  

calculation register
"\( t_0 = t_1 \)"

"\( t_0 = t_2 \)"

:  

"\( t_0 = t_{n-1} \)"
also/finally

have "\(t_0 = t_1\)" [proof]
also
have "\(\ldots = t_2\)" [proof]
also
\[ \vdots \]
also
have "\(\ldots = t_n\)" [proof]
finally
calculation register
"\(t_0 = t_1\)"
"\(t_0 = t_2\)"
\[ \vdots \]
"\(t_0 = t_{n-1}\)"
\[ t_0 = t_n \]
also

have \( t_0 = t_1 \) [proof]
also
have \( \ldots = t_2 \) [proof]
also
:\nalso
have \( \ldots = t_n \) [proof]
finally

show P

— 'finally' pipes fact \( t_0 = t_n \) into the proof
More about also

- Works for all combinations of $\equiv$, $\leq$ and $<$. 
More about also

- Works for all combinations of $=$, $\leq$ and $<$.  
- Uses all rules declared as [trans].
More about also

- Works for all combinations of $=$, $\leq$ and $<$. 
- Uses all rules declared as $[\text{trans}]$. 
- To view all combinations in Proof General:
  
  Isabelle/Isar $\rightarrow$ Show me $\rightarrow$ Transitivity rules
have = ”$l_1 \odot r_1$” [proof]
also
have ”$\ldots \odot r_2$” [proof]
also
have = "l₁ ⊙ r₁" [proof]
also
have "... ⊙ r₂" [proof]
also

Anatomy of a [trans] rule:

→ Usual form: plain transitivity \[ [l₁ \circ r₁; r₁ \circ r₂] \rightarrow l₁ \circ r₂ \]
Designing [trans] Rules

have = ”$l_1 \odot r_1$” [proof]
also
have ”... $\odot r_2$” [proof]
also

Anatomy of a [trans] rule:

- Usual form: plain transitivity $[l_1 \odot r_1; r_1 \odot r_2] \rightarrow l_1 \odot r_2$
- More general form: $[[P l_1 r_1; Q r_1 r_2; A]] \rightarrow C l_1 r_2$

Examples:
Designing [trans] Rules

have = "l₁ ⊗ r₁" [proof]
also
have ". . . ⊗ r₂" [proof]
also

Anatomy of a [trans] rule:

→ Usual form: plain transitivity \([l₁ ⊗ r₁; r₁ ⊗ r₂] \rightarrow l₁ ⊗ r₂\)
→ More general form: \([P l₁ r₁; Q r₁ r₂; A] \rightarrow C l₁ r₂\)

Examples:

→ pure transitivity: \([a = b; b = c] \rightarrow a = c\)
Designing [trans] Rules

\[
\text{have } = "l_1 \odot r_1" \quad \text{[proof]}
\]

also

\[
\text{have } "\ldots \odot r_2" \quad \text{[proof]}
\]

also

Anatomy of a [trans] rule:

\[\to \quad \text{Usual form: plain transitivity } [l_1 \odot r_1; r_1 \odot r_2] \Rightarrow l_1 \odot r_2\]

\[\to \quad \text{More general form: } [P \ l_1 \ \ r_1; Q \ r_1 \ r_2; A] \Rightarrow C \ l_1 \ r_2\]

Examples:

\[\to \quad \text{pure transitivity: } [a = b; b = c] \Rightarrow a = c\]

\[\to \quad \text{mixed: } [a \leq b; b < c] \Rightarrow a < c\]
Designing [trans] Rules

have = "\(l_1 \odot r_1\)" [proof]
also
have "\(\ldots \odot r_2\)" [proof]
also

Anatomy of a [trans] rule:

→ Usual form: plain transitivity \([l_1 \odot r_1; r_1 \odot r_2] \implies l_1 \odot r_2\)
→ More general form: \([P l_1 r_1; Q r_1 r_2; A] \implies C l_1 r_2\)

Examples:

→ pure transitivity: \([a = b; b = c] \implies a = c\)
→ mixed: \([a \leq b; b < c] \implies a < c\)
→ substitution: \([P a; a = b] \implies P b\)
Designing [trans] Rules

\( \text{have} = "l_1 \odot r_1" \) [proof]
also
\( \text{have} \ldots \odot r_2" \) [proof]
also

Anatomy of a [trans] rule:

→ Usual form: plain transitivity \( [l_1 \odot r_1; r_1 \odot r_2] \implies l_1 \odot r_2 \)
→ More general form: \( [P l_1 r_1; Q r_1 r_2; A] \implies C l_1 r_2 \)

Examples:

→ pure transitivity: \( [a = b; b = c] \implies a = c \)
→ mixed: \( [a \leq b; b < c] \implies a < c \)
→ substitution: \( [P a; a = b] \implies P b \)
→ antisymmetry: \( [a < b; b < a] \implies P \)
Designing [trans] Rules

\[
\text{have } = "l_1 \odot r_1" \text{ [proof]}
\]
also
\[
\text{have } \ldots \odot r_2" \text{ [proof]}
\]
also

Anatomy of a [trans] rule:

- Usual form: plain transitivity \([l_1 \odot r_1; r_1 \odot r_2] \rightarrow l_1 \odot r_2\)
- More general form: \([P \ l_1 \ r_1; Q \ r_1 \ r_2; A] \rightarrow C \ l_1 \ r_2\)

Examples:

- pure transitivity: \([a = b; b = c] \rightarrow a = c\)
- mixed: \([a \leq b; b < c] \rightarrow a < c\)
- substitution: \([P \ a; a = b] \rightarrow P \ b\)
- antisymmetry: \([a < b; b < a] \rightarrow P\)
- monotonicity: \([a = f \ b; b < c; \bigwedge x \ y. \ x < y \rightarrow f \ x < f \ y] \rightarrow a < f \ c\)
DEMO
HOL as programming language

We have

- numbers, arithmetic
- recursive datatypes
- constant definitions, recursive functions
HOL as programming language

We have

- numbers, arithmetic
- recursive datatypes
- constant definitions, recursive functions
- = a functional programming language
- can be used to get fully verified programs

Executed using the simplifier.
HOL as programming language

We have

- numbers, arithmetic
- recursive datatypes
- constant definitions, recursive functions
- = a functional programming language
- can be used to get fully verified programs

Executed using the simplifier. But:

- slow, heavy-weight
- does not run stand-alone (without Isabelle)
Generating ML code

Generate stand-alone ML code for

- datatypes
- function definitions
- inductive definitions (sets)
Generating ML code

Generate stand-alone ML code for

- datatypes
- function definitions
- inductive definitions (sets)

Syntax (simplified):

```
code_module <structure-name> [file <name>]
contains
  <ML-name> = <term>
  ...
  <ML-name> = <term>
```

Generates ML structure, puts it in own file or includes in current context
Value and Quickcheck

Evaluate big terms quickly:

```
value "<term>"
```

→ generates ML code
→ runs ML
→ converts back into Isabelle term
Value and Quickcheck

Evaluate big terms quickly:

```
value "<term>"
```

- generates ML code
- runs ML
- converts back into Isabelle term

Try some values on current proof state:

```
quickcheck
```

- generates ML code
- runs ML on random values for numbers and datatypes
- increasing size of data set until limit reached
Customisation

→ lemma instead of definition: [code] attribute

**lemma** [code]: "(0 < Suc n) = True" by simp
→ lemma instead of definition: [code] attribute

    lemma [code]: "(0 < Suc n) = True" by simp

→ provide own code for types: types_code

    types_code "×" ("_/ _")
Customisation

➔ lemma instead of definition: [code] attribute

  lemma [code]: 
  
  lemma [code]: 
  lemma [code]: "(0 < Suc n) = True" by simp

➔ provide own code for types: types_code

  types_code "×" "("(_, _)/ _")"

➔ provide own code for consts: consts_code

  consts_code "Pair" "("(_, _)/ _")"
Customisation

➔ lemma instead of definition: [code] attribute
  lemma [code]: "(0 < Suc n) = True" by simp

➔ provide own code for types: types_code
  types_code "×" "(_ */ _)"

➔ provide own code for consts: consts_code
  consts_code "Pair" "(_/ _)"

➔ complex code template: patterns + attach
  consts_code "wfrec" "\ <module>wfrec?"
  attach { * fun wfrec f x = f (wfrec f) x; * }
Inductive definitions are Horn clauses:

\[(0, \text{Suc } n) \in L\]

\[(n, m) \in L \implies (\text{Suc } n, \text{Suc } m) \in L\]
Inductive definitions are Horn clauses:

\[(0, \text{Suc } n) \in L\]

\[(n,m) \in L \implies (\text{Suc } n, \text{Suc } m) \in L\]

**Can be evaluated like Prolog**
Inductive definitions are Horn clauses:

\[(0, \text{Suc } n) \in L\]
\[(n, m) \in L \implies (\text{Suc } n, \text{Suc } m) \in L\]

Can be evaluated like Prolog

```prolog
code_module T
contains x = \lambda x y. (x, y) \in L
y = (\_, 5) \in L
```

generates

- something of type bool for \(x\)
- a possibly infinite sequence for \(y\), enumerating all suitable \(\_\) in \((\_, 5) \in L\)
DEMO
We have seen today...

- More fun
- Calculations: also/finally
- [trans]-rules
- Code generation