COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein
Formal Methods

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

Can we do this in Isabelle?

➤ Simplifier: too eager
➤ Manual: difficult in apply style
➤ Isar: with the methods we know, too verbose
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
- \( \ldots : \) predefined schematic term variable,
  refers to right hand side of last expression
- Automatic use of transitivity rules to connect steps
have "\( t_0 = t_1 \)" [proof]
also
have "\( \ldots = t_2 \)" [proof]
also
:  
also
have "\( \ldots = t_n \)" [proof]
finally
show P  
— 'finally' pipes fact "\( t_0 = t_n \)" into the proof

\[
\text{calculation register}
\]
\[
"t_0 = t_1"
\]

\[
"t_0 = t_2"
\]

\[
:  
\]

\[
"t_0 = t_{n-1}"
\]

\[
t_0 = t_n
\]
More about also

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

- Uses all rules declared as [trans].

- To view all combinations in Proof General:
  
  Isabelle/Isar → Show me → Transitivity rules
**Designing [trans] Rules**

calculation = ”$l_1 \odot r_1$”

have ”... $\odot r_2$” [proof]

also ⇐

**Anatomy of a [trans] rule:**

→ Usual form: plain transitivity $[l_1 \odot r_1; r_1 \odot r_2] \implies l_1 \odot r_2$

→ 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: $[a < b; b < a] \implies P$

→ monotonicity: $[a = f b; b < c; \land x y. x < y \implies f x < f y] \implies a < f c$
DEMO
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)
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
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
 lemma instead of definition: \textbf{[code]} attribute

\begin{verbatim}
lemma [code]: "(0 < Suc n) = True" by simp
\end{verbatim}

provide own code for types: \textbf{types_code}

\begin{verbatim}
types_code "\times" "("("_ */ _")")
\end{verbatim}

provide own code for consts: \textbf{consts_code}

\begin{verbatim}
consts_code "Pair" "("_/_")")
\end{verbatim}

complex code template: patterns + \textbf{attach}

\begin{verbatim}
consts_code "wfrec" "\backslash <module>\backslash wfrec?"
attach { * fun wfrec f x = f (wfrec f) x; *}
\end{verbatim}
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