COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray

fun
Content

➜ Intro & motivation, getting started

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

➜ Proof & Specification Techniques
• Isar
• Inductively defined sets, rule induction
• Datatypes, recursion, induction
• Calculational reasoning, mathematics style proofs
• Hoare logic, proofs about programs

Rough timeline

[1]

[2,3,4]a

[5,6b,7]

[8,9,10c]

[11,12d]

[13e,15]

[16,17f,18,19]

[20]

[21g,22,23]

a a1 out; b a1 due; c a2 out; d a2 due; e session break; f a3 out; g a3 due
The Choice

- Limited expressiveness, automatic termination
  - primrec

- High expressiveness, termination proof may fail
  - fun

- High expressiveness, tweakable, termination proof manual
  - function
fun — examples

fun sep :: "'a ⇒ 'a list ⇒ 'a list"
where
  "sep a (x # y # zs) = x # a # sep a (y # zs)" |
  "sep a xs = xs"

fun ack :: "nat ⇒ nat ⇒ nat"
where
  "ack 0 n = Suc n" |
  "ack (Suc m) 0 = ack m 1" |
  "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"
The definition:

- pattern matching in all parameters
- arbitrary, linear constructor patterns
- reads equations sequentially like in Haskell (top to bottom)
- proves termination automatically in many cases (tries lexicographic order)

Generates own induction principle

May fail to prove termination:

- use function (sequential) instead
- allows you to prove termination manually
fun — induction principle

→ Each **fun** definition induces an induction principle

→ For each equation:

  show P holds for lhs, provided P holds for each recursive call on rhs

→ Example **sep.induct**:

  \[
  \[
  \forall a. P a \;
  \forall a w. P a [w] \;
  \forall a x y zs. P a (y\#zs) \implies P a (x\#y\#zs); \\
  \] \implies P a xs
  \]
Termination

Isabelle tries to prove termination automatically

- For most functions this works with a lexicographic termination relation.
- Sometimes not ⇒ error message with unsolved subgoal
- You can prove automation separately.

function (sequential) quicksort where
quicksort [] = [] |
quicksort (x#xs) = quicksort [y ← xs.y ≤ x]@[x]@ quicksort [y ← xs.x < y]
by pat_completeness auto

termination
by (relation “measure length”) (auto simp: less_Suc_eq_le)

function is the fully tweakable, manual version of fun
DEMO
How does fun/function work?

Recall **primrec**:

- defined one recursion operator per datatype
- inductive definition of its graph \((x, f\ x) \in G\)
- prove totality: \(\forall x. \exists y. (x, y) \in G\)
- prove uniqueness: \((x, y) \in G \Rightarrow (x, z) \in G \Rightarrow y = z\)
- recursion operator: \(\text{rec } x = \text{THE } y. (x, y) \in \text{rec}\)
How does fun/function work?

Similar strategy for **fun**:

- a new inductive definition for each **fun** $f$

- extract *recursion scheme* for equations in $f$

- define graph $f_{rel}$ inductively, encoding recursion scheme

- prove totality (= termination)

- prove uniqueness (automatic)

- derive original equations from $f_{rel}$

- export induction scheme from $f_{rel}$
How does fun/function work?

Can separate and defer termination proof:

- skip proof of totality

- instead derive equations of the form: \( x \in f_{\text{dom}} \Rightarrow f \ x = \ldots \)
- similarly, conditional induction principle

- \( f_{\text{dom}} = acc \ f_{\text{rel}} \)
- \( acc = \text{accessible part of } f_{\text{rel}} \)
- the part that can be reached in finitely many steps

- termination = \( \forall x. x \in f_{\text{dom}} \)
- still have conditional equations for partial functions
Proving Termination

Command `termination fun_name` sets up termination goal
\[ \forall x. \ x \in \text{fun_name}_\text{dom} \]

Three main proof methods:

- lexicographic_order (default tried by `fun`)
- size_change (different automated technique)
- relation R (manual proof via well-founded relation)
Definition

\( <_r \) is well founded if well founded induction holds

\[ \text{wf} \ r \equiv \forall P. \ (\forall x. \ (\forall y <_r x. P y) \rightarrow P x) \rightarrow (\forall x. P x) \]

Well founded induction rule:

\[
\frac{\text{wf} \ r \ \land x. \ (\forall y <_r x. P y) \rightarrow P x}{P a}
\]

Alternative definition (equivalent):

there are no infinite descending chains, or (equivalent):

every nonempty set has a minimal element \( \text{wrt} \ <_r \)

\[
\begin{align*}
\text{min} \ r \ Q \ x & \equiv \forall y \in Q. \ y \not<_r x \\
\text{wf} \ r & \equiv (\forall Q \neq \{\}. \exists m \in Q. \ \text{min} \ r \ Q \ m)
\end{align*}
\]
Well Founded Orders: Examples

→ < on \( \mathbb{N} \) is well founded
   well founded induction = complete induction

→ > and \( \leq \) on \( \mathbb{N} \) are **not** well founded

→ \( x <_r y = x \text{ dvd } y \land x \neq 1 \) on \( \mathbb{N} \) is well founded
   the minimal elements are the prime numbers

→ \((a, b) <_r (x, y) = a <_1 x \lor a = x \land b <_2 y \) is well founded
   if \( <_1 \) and \( <_2 \) are

→ \( A <_r B = A \subset B \land \text{finite } B \) is well founded

→ \( \subseteq \) and \( \subset \) in general are **not** well founded

More about well founded relations: *Term Rewriting and All That*
So far for termination. What about the recursion scheme? Not fixed anymore as in primrec.

Examples:

→ `fun fib where`
   fib 0 = 1 |
   fib (Suc 0) = 1 |
   fib (Suc (Suc n)) = fib n + fib (Suc n)

Recursion: Suc (Suc n) \sim n, Suc (Suc n) \sim Suc n

→ `fun f where`
   f x = (if x = 0 then 0 else f (x - 1) * 2)

Recursion: x \neq 0 \implies x \sim x - 1
Higher Oder:

→ **datatype** 'a tree = Leaf 'a | Branch 'a tree list

    **fun** treemap :: ('a ⇒ 'a) ⇒ 'a tree ⇒ 'a tree **where**
    treemap fn (Leaf n) = Leaf (fn n) |
    treemap fn (Branch l) = Branch (map (treemap fn) l)

**Recursion**: \( x \in \text{set } l \implies (fn, \text{Branch } l) \sim (fn, x) \)

**How to extract the context information for the call?**
Extracting the Recursion Scheme

Extracting context for equations

⇒

Congruence Rules!

Recall rule \textbf{if\_cong}:

\[
\begin{align*}
\begin{array}{l}
\text{if } b = c; c \Rightarrow x = u; \neg c \Rightarrow y = v \end{array}
\end{align*}
\]

\[(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)\]

\textbf{Read}: for transforming } x, \text{ use } b \text{ as context information, for } y \text{ use } \neg b.

\textbf{In fun\_def}: for recursion in } x, \text{ use } b \text{ as context, for } y \text{ use } \neg b.
Congruence Rules for fun_defs

The same works for function definitions.

\textbf{declare} my\_rule[fundef\_cong]

(if\_cong already added by default)

Another example (higher-order):

\[ |xs = ys; \land x. x \in \text{set } ys \Rightarrow f x = g x | \Rightarrow \text{map } f \ xs = \text{map } g \ ys \]

\textbf{Read:} for recursive calls in $f$, $f$ is called with elements of $xs$
DEMO
Further Reading

Alexander Krauss,

*Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.*

http://www4.in.tum.de/~krauss/diss/krauss_phd.pdf
We have seen today ...

- General recursion with **fun/function**
- Induction over recursive functions
- How **fun** works
- Termination, partial functions, congruence rules