Content

→ Intro & motivation, getting started

→ Foundations & Principles
  • Lambda Calculus, natural deduction \[1,2\]
  • Higher Order Logic \[3^a\]
  • Term rewriting \[4\]

→ Proof & Specification Techniques
  • Inductively defined sets, rule induction \[5\]
  • Datatypes, recursion, induction \[6,7\]
  • Hoare logic, proofs about programs, C verification \[8^b,9\]
  • (mid-semester break)
  • Writing Automated Proof Methods \[10\]
  • Isar, codegen, typeclasses, locales \[11^c,12\]

\(^{a}a_{1} \text{ due}; \ ^{b}a_{2} \text{ due}; \ ^{c}a_{3} \text{ due}\)
General Recursion

The Choice

- Limited expressiveness, automatic termination
  - primrec

- High expressiveness, termination proof may fail
  - fun

- High expressiveness, tweakable, termination proof manual
  - function
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`:
  
  \[
  \begin{align*}
  \forall & a. \; P \ a \ []; \\
  \quad & \forall a \ w. \; P \ a \ [w] \\
  \quad & \forall a \ x \ y \ zs. \; P \ a \ (y#zs) \implies P \ a \ (x#y#zs); \\
  \implies & P \ a \ xs \\
  \end{align*}
  \]
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]@[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* $D$
- inductive definition of its graph $\langle x, f \ x \rangle \in D\_rel$
- prove totality: $\forall x. \exists y. (x, y) \in D\_rel$
- prove uniqueness: $(x, y) \in D\_rel \Rightarrow (x, z) \in D\_rel \Rightarrow y = z$
- recursion operator for datatype $D\_rec$, defined via *THE*.
- primrec: apply datatype recursion operator
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: \( x \in f\_dom \Rightarrow f \ x = \ldots \)
- similarly, conditional induction principle
- \( f\_dom = acc \ f\_rel \)
- \( acc = \) accessible part of \( f\_rel \)
- the part that can be reached in finitely many steps
- termination = \( \forall x. \ x \in f\_dom \)
- still have conditional equations for partial functions
Proving Termination

Command **termination fun_name** sets up termination goal
\[ \forall x. \ x \in fun\_name\_dom \]

Three main proof methods:
- lexicographic_order (default tried by fun)
- size_change (different automated technique)
- relation R (manual proof via well-founded relation)
Well Founded Orders

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:

\[
\begin{align*}
\text{wf} \ r & \quad \land \ x. \ (\forall y <_r x. P y) \quad \rightarrow \quad P x \\
\hline
P \ a
\end{align*}
\]

Alternative definition (equivalent):

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

every nonempty set has a minimal element 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

$\rightarrow$ $<$ on $\mathbb{N}$ is well founded
  well founded induction $=$ complete induction
$\rightarrow$ $>$ and $\leq$ on $\mathbb{N}$ are not well founded
$\rightarrow$ $x <_r y = x \text{ dvd } y \land x \neq 1$ on $\mathbb{N}$ is well founded
  the minimal elements are the prime numbers
$\rightarrow$ $(a, b) <_r (x, y) = a <_1 x \lor a = x \land b <_2 y$ is well founded
  if $<_1$ and $<_2$ are
$\rightarrow$ $A <_r B = A \subset B \land \text{ finite } B$ is well founded
$\rightarrow$ $\subseteq$ and $\subset$ in general are not well founded

More about well founded relations: Term Rewriting and All That
Extracting the Recursion Scheme

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) \(\leadsto\) n, Suc (Suc n) \(\leadsto\) Suc n

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

  Recursion: \(x \neq 0 \implies x \leadsto x - 1\)
Higher Oder:

\[\text{datatype} \quad \text{'}a\text{ tree} = \text{Leaf 'a} | \text{Branch 'a tree list}\]

\[\text{fun treemap :: ('a} \Rightarrow \text{'}a\text{) \Rightarrow 'a tree \Rightarrow 'a tree where}\]

\[\text{treemap fn (Leaf n) = Leaf (fn n)} | \]

\[\text{treemap fn (Branch l) = Branch (map (treemap fn) l)}\]

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

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

Extracting context for equations

⇒

Congruence Rules!

Recall rule if_cong:

\[
\begin{align*}
| \quad b & = c; \ c \rightarrow x = u; \ \neg c \rightarrow y = v \quad | \quad \rightarrow \\
\text{(if } b \text{ then } x \text{ else } y) & = (\text{if } c \text{ then } u \text{ else } v)
\end{align*}
\]

Read: for transforming \( x \), use \( b \) as context information, for \( y \) use \( \neg b \).

In fun_def: for recursion in \( x \), use \( b \) as context, for \( y \) use \( \neg b \).
Congruence Rules for fun_defs

The same works for function definitions.

**declare** my_rule[fundef_cong]

(if_cong already added by default)

Another example (higher-order):

\[ [\mid xs = ys; \forall x. x \in \text{set } ys \implies f x = g x ] \implies \text{map } f \; xs = \text{map } g \; ys \]

**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