COMP4011/8011 Advanced Topics in Formal Methods and Programming Languages

– Software Verification with Isabelle/HOL –

Peter Höfner

September 17, 2024

Section 13

General Recursion

General Recursion

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)”

fun

- More permissive than **primrec**:
  - pattern matching in all parameters
  - nested, linear constructor patterns
  - reads equations sequentially like in Haskell (top to bottom)
  - proves termination automatically in many cases
    (tries lexicographic order)
- Generates more theorems than **primrec**
- May fail to prove termination:
  - use **function (sequential)** instead
  - allows you to prove termination manually

Demo

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**:

$$\llbracket\ \bigwedge a.\ P\ a\ [];\quad \bigwedge a\ w.\ P\ a\ [w];\quad \bigwedge a\ x\ y\ zs.\ P\ a\ (y\#zs) \Longrightarrow P\ a\ (x\#y\#zs);\ \rrbracket \Longrightarrow P\ a\ xs$$

Termination

Isabelle tries to prove termination automatically

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

function (sequential) quicksort **where**
"quicksort [] = []" |
"quicksort ($x\#xs$) = (quicksort [$y \leftarrow xs.\ y \leq x$])@[$x$]@(quicksort [$y \leftarrow xs.\ x < y$])"
by pat_completeness auto

termination
by (relation "measure length") (auto simp: less_Suc_eq_le)

Demo

How does fun/function work?

Recall **primrec**:

- defined one recursion operator per datatype D
- inductive definition of its graph $(x, f\ x) \in D\_rel$
- 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?

function can separate and defer termination proof:

- skip proof of totality
- instead derive equations of the form: $x \in f\_dom \Rightarrow f\ x = ...$
- 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

Demo

Proving Termination

termination fun_name sets up termination goal $\forall x.\ x \in \mathit{fun\_name\_dom}$

Three main proof methods:

- **lexicographic_order** (default tried by **fun**)
- **size_change** (automated translation to simpler size-change graph[1])
- **relation R** (manual proof via well-founded relation)

[1] C.S. Lee, N.D. Jones, A.M. Ben-Amram, *The Size-change Principle for Program Termination*, POPL 2001.

Well-Founded Orders

Definition

$<_r$ is well founded if well-founded induction holds

$\mathrm{wf}(<_r) \equiv \forall P.\ (\forall x.\ (\forall y <_r x. P\ y) \longrightarrow P\ x) \longrightarrow (\forall x.\ P\ x)$

Well founded induction rule:

$$\frac{\mathrm{wf}(<_r) \quad \bigwedge x.\ (\forall y <_r x.\ P\ y) \Longrightarrow P\ x}{P\ a}$$

Alternative definition (equivalent):

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

every nonempty set has a minimal element wrt $<_r$

$$\begin{array}{lcl} \min\ (<_r)\ Q\ x & \equiv & \forall y \in Q.\ y \not<_r x \\ \mathrm{wf}\ (<_r) & = & (\forall Q \neq \{\}.\ \exists m \in Q.\ \min\ r\ Q\ m) \end{array}$$

Well-Founded Orders: Examples

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

More about well founded relations: *Term Rewriting and All That*

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 $\Longrightarrow$ x $\leadsto$ x - 1

Extracting the Recursion Scheme

Higher Order:

- **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: $x \in \text{set } l \Longrightarrow (fn, \text{Branch } l) \leadsto (fn, x)$

How does Isabelle extract context information for the call?

Extracting the Recursion Scheme

Extracting context for equations

$\Rightarrow$

Congruence Rules!

Recall rule **if_cong**:

$$\llbracket\ b = c;\ c \Longrightarrow x = u;\ \neg\ c \Longrightarrow y = v\ \rrbracket \Longrightarrow$$
$$(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$$

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):
$\llbracket$ xs = ys; $\bigwedge$x. x $\in$ set ys $\Longrightarrow$ f x = g x $\rrbracket$ $\Longrightarrow$ map f xs = 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.
PhD thesis, TU Munich, 2009.

`https://www21.in.tum.de/~krauss/papers/krauss-thesis.pdf`

We have seen ...

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