COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski

fun

Slide 1

Content

[a] a1 due; [b] a2 due; [c] session break; [d] a3 due

Slide 2

General Recursion

The Choice

- ➜ Limited expressiveness, automatic termination
  - primrec
- ➜ High expressiveness, termination proof may fail
  - fun
- ➜ High expressiveness, tweakable, termination proof manual
  - function

Slide 3

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)"

Slide 4

fun

- ➜ The definiton:
  - 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

Slide 5

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**:

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

Slide 6

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 automation 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)

function is the fully tweakable, manual version of **fun**

Slide 7

DEMO

Slide 8

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: $rec\ x = THE\ y.\ (x, y) \in rec$

Slide 9

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

Slide 10

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

Slide 11

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)

Slide 12

Well Founded Orders

Definition

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

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:

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

Alternative definition (equivalent):

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

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

$\text{min } r\ Q\ x \quad \equiv \quad \forall y \in Q.\ y \not<_r x$

$\text{wf } r \quad = \quad (\forall Q \neq \{\}.\ \exists m \in Q.\ \text{min } r\ Q\ m)$

Slide 13

Well Founded Orders: Examples

- $<$ on $\mathbb{N}$ is well founded
  well founded induction = complete induction
- $>$ and $\leq$ on $\mathbb{N}$ are **not** well founded
- $x <_r y = x$ 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
- $A <_r B = A \subset B \wedge$ finite B is well founded
- $\subseteq$ and $\subset$ in general are **not** well founded

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

Slide 14

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) $\rightsquigarrow$ n, Suc (Suc n) $\rightsquigarrow$ Suc n

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

  Recursion: x $\neq$ 0 $\Longrightarrow$ x $\rightsquigarrow$ x - 1

Slide 15

Extracting the Recursion Scheme

Higher Oder:

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

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

  Recursion: x $\in$ set l $\Longrightarrow$ (fn, Branch l) $\rightsquigarrow$ (fn, x)

How to extract the context information for the call?

Slide 16

Extracting the Recursion Scheme

Extracting context for equations

$\Rightarrow$

Congruence Rules!

Recall rule **if_cong**:

$[\![\text{b = c; c} \Longrightarrow \text{x = u;} \neg \text{c} \Longrightarrow \text{y = v}]\!] \Longrightarrow$
(if b then x else y) = (if c then u 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 $\neg b$.

Slide 17

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):
$[\![\text{xs = ys;} \bigwedge \text{x. x} \in \text{set ys} \Longrightarrow \text{f x = g x}]\!] \Longrightarrow$ map f xs = map g ys

Read: for recursive calls in f, f is called with elements of xs

Slide 18

DEMO

Slide 19

Further Reading

Alexander Krauss,
Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.
PhD thesis, TU Munich, 2009.

`http://www4.in.tum.de/~krauss/diss/krauss_phd.pdf`

Slide 20

We have seen today ...

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

Slide 21