



COMP4161: Advanced Topics in Software Verification

fun

Gerwin Klein, June Andronick, Ramana Kumar
S2/2016

data61.csiro.au



Content



- Intro & motivation, getting started [1]

- 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]

^aa1 due; ^ba2 due; ^ca3 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 — 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 defintion:
 - 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**:
$$\llbracket \bigwedge a. P\ a \rrbracket;$$
$$\bigwedge a\ w. P\ a\ [w]$$
$$\bigwedge a\ x\ y\ zs. P\ a\ (y\#\!zs) \implies P\ a\ (x\#\!y\#\!zs);$$
$$\rrbracket \implies P\ a\ xs$$

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 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**

A background pattern of white hexagons on a dark teal background, arranged in a staggered grid.

DATA
61



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?



Can separate and defer termination proof:

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

$\forall x. x \in \text{fun_name_dom}$

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

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

Well founded induction rule:

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

Alternative definition (equivalent):

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

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

$$\min r Q x \equiv \forall y \in Q. y \not<_r x$$

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

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
- $A <_r B = A \subset B \wedge \text{finite } B$ 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: $\text{Suc (Suc } n) \rightsquigarrow n$, $\text{Suc (Suc } n) \rightsquigarrow \text{Suc } n$

→ **fun f where** $f\ x = (\text{if } x = 0 \text{ then } 0 \text{ else } f\ (x - 1) * 2)$

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

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: $x \in \text{set } l \implies (\text{fn}, \text{Branch } l) \rightsquigarrow (\text{fn}, x)$

How to extract the context information for the call?

Extracting the Recursion Scheme



Extracting context for equations

\Rightarrow

Congruence Rules!

Recall rule **if_cong**:

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

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):

$$[| xs = ys; \bigwedge x. x \in \text{set } ys \implies f\ x = g\ x |] \implies \text{map } f\ xs = \text{map } g\ ys$$

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

A background pattern of white hexagons on a teal background, arranged in a staggered grid.

DATA
61



Demo

Further Reading



Alexander Krauss,

Automating Recursive Definitions and Termination Proofs in Higher-Order

PhD thesis, TU Munich, 2009.

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