



COMP4161: Advanced Topics in Software Verification

**fun**

Gerwin Klein, Johannes Åman Pohjola, Christine Rizkallah, Miki Tanaka  
T3/2020

# Content

## → Foundations & Principles

- Intro, Lambda calculus, natural deduction [1,2]
- Higher Order Logic, Isar (part 1) [2,3<sup>a</sup>]
- Term rewriting [3,4]

## → Proof & Specification Techniques

- Inductively defined sets, rule induction, datatype induction, primitive recursion [4,5]
- General recursive functions, termination proofs [7<sup>b</sup>]
- Proof automation, Hoare logic, proofs about programs, invariants [8]
- C verification [9,10]
- Practice, questions, exam prep [10<sup>c</sup>]

---

<sup>a</sup>a1 due; <sup>b</sup>a2 due; <sup>c</sup>a3 due

# General Recursion

The Choice

# 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 — 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)

# 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**



# 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

# fun — induction principle

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

# 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

# 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.

# Termination

**Isabelle tries to prove termination automatically**

- For most functions this works with a lexicographic termination relation.
- Sometimes not

# 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

# 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 ← xs.y ≤ x]@[x]@ quicksort [y ← 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**  $D$

# How does fun/function work?

Recall **primrec**:

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

# 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:  $\forall x. \exists y. (x, y) \in D\_rel$

# 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:  $(x, y) \in D\_rel \Rightarrow (x, z) \in D\_rel \Rightarrow y = z$

# 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*.

# 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

# How does fun/function work?

**function** can separate and defer termination proof:

- skip proof of totality
- instead derive equations of the form:  $x \in f\_dom \Rightarrow f\ x = \dots$
- similarly, conditional induction principle

# How does fun/function work?

**function** can separate and defer termination proof:

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

# How does fun/function work?

**function** can separate and defer termination proof:

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

**termination fun\_name** sets up termination goal  $\forall x. x \in \text{fun\_name\_dom}$

Three main proof methods:

# Proving Termination

**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**)

# Proving Termination

**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** (automated translation to simpler size-change graph<sup>1</sup>)

---

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

# Proving Termination

**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** (automated translation to simpler size-change graph<sup>1</sup>)
- **relation R** (manual proof via well-founded relation)

---

<sup>1</sup>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

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

# 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:

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

# 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

# Well Founded Orders: Examples

- $<$  on  $\mathbb{N}$  is well founded  
well founded induction = complete induction
- $>$  and  $\leq$  on  $\mathbb{N}$  are **not** well founded

# 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

# 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

# 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



# 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?

# 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)

# 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:  $\text{Suc (Suc } n) \rightsquigarrow n$ ,  $\text{Suc (Suc } n) \rightsquigarrow \text{Suc } n$

# 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:  $\text{Suc} (\text{Suc } n) \rightsquigarrow n$ ,  $\text{Suc} (\text{Suc } n) \rightsquigarrow \text{Suc } n$

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

# 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:  $\text{Suc} (\text{Suc } n) \rightsquigarrow n$ ,  $\text{Suc} (\text{Suc } n) \rightsquigarrow \text{Suc } n$

→ **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)

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



# 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 does Isabelle extract context information for the call?

# Extracting the Recursion Scheme

Extracting context for equations

# Extracting the Recursion Scheme

Extracting context for equations

$\Rightarrow$

Congruence Rules!

# Extracting the Recursion Scheme

Extracting context for equations

$\Rightarrow$

Congruence Rules!

Recall rule **if\_cong**:

$$\begin{aligned} & [ [ b = c; c \implies x = u; \neg c \implies y = v ] ] \implies \\ & (\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v) \end{aligned}$$

**Read:** for transforming  $x$ , use  $b$  as context information, for  $y$  use  $\neg b$ .

# Extracting the Recursion Scheme

Extracting context for equations

$\Rightarrow$

Congruence Rules!

Recall rule **if\_cong**:

$$\begin{aligned} & [ [ b = c; c \implies x = u; \neg c \implies y = v ] ] \implies \\ & (\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v) \end{aligned}$$

**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]

# 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):

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

# 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  $xs$



**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 today ...

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