UNSW SYDNEY

COMP4161

Advanced Topics in Software Verification

fun

Gerwin Klein, June Andronick, Miki Tanaka, Johannes Åman Pohjola

T3/2022

Content

➜ Foundations & Principles

- Intro, Lambda [illegible] atural deduction [1,2]
- Higher Order [illegible] part 1) [2,3[a]]
- Term rewriting [3,4]

➜ Proof & Specification Techniques

- Inductively defined sets, rule induction [4,5]
- Datatype induction, primitive recursion [5,7]
- General recursive functions, termination proofs [7[b]]
- Proof automation, Isar (part 2) [8]
- Hoare logic, proofs about programs, invariants [8,9]
- C verification [9,10]
- Practice, questions, exam prep [10[c]]

[a] a1 due; [b] a2 due; [c] a3 due

General Recursion

General Recursion

e Choice

- ➜ Limited expressiven tic 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 match[illegible]rameters
- nested, linear [illegible]patterns
- reads equation[illegible]ly like in Haskell (top to bottom)
- proves termination automatically in many cases (tries lexicographic order)

fun

➜ More permissive than **primrec**:

- pattern match[illegible]rameters
- nested, linear [illegible]patterns
- reads equation[illegible]ly 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 match[illegible]rameters
  - nested, linear [illegible]patterns
  - reads equation[illegible]ly 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 

induction principle

fun — induction principle

- ➜ Each **fun** definition induction principle
- ➜ For each equation:

  show P holds for lh P holds for each recursive call on rhs

fun — induction principle

- Each **fun** definition induction principle
- For each equation:

  show P holds for lh 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  with a lexicographic termination relation.

Termination

Isabelle tries to prove termination automatically

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

Termination

Isabelle tries to prove termination automatically

- ➜ For most functions with a lexicographic termination relation.
- ➜ Sometimes not ⇒ e with unsolved subgoal

Termination

Isabelle tries to prove termination automatically

- ➜ For most functions [illegible] with a lexicographic termination relation.
- ➜ Sometimes not ⇒ [illegible]e with unsolved subgoal
- ➜ You can prove term[illegible]arately.

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 recursic  per datatype D

How does fun/function work?

Recall **primrec**:

- ➜ defined one recursic  per datatype D
- ➜ inductive definition $(x, f\ x) \in D\_rel$

How does fun/function work?

Recall **primrec**:

- ➜ defined one recursic per datatype D
- ➜ inductive definition $(x, f\ x) \in D\_rel$
- ➜ prove totality: $\forall x.$ D_rel

How does fun/function work?

Recall **primrec**:

- ➜ defined one recursi[illegible]per datatype D
- ➜ inductive definition [illegible] $(x, f\ x) \in D\_rel$
- ➜ prove totality: $\forall x.$ [illegible] 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 recursic per datatype D
- ➜ inductive definition $(x, f\ x) \in D\_rel$
- ➜ prove totality: $\forall x.$: 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 recursi per datatype D
- ➜ inductive definition $(x, f\ x) \in D\_rel$
- ➜ prove totality: $\forall x.$ 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 def ach **fun** *f*
- ➜ extract *recursion sc* uations in *f*
- ➜ define graph *f_rel* i ncoding 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 totalit

How does fun/function work?

function can separate and defer termination proof:

- ➜ skip proof of totalit
- ➜ instead derive equa form: $x \in f\_dom \Rightarrow f\ x = \ldots$
- ➜ similarly, conditiona principle

How does fun/function work?

function can separate and defer termination proof:

- ➜ skip proof of totalit
- ➜ instead derive equa form: $x \in f\_dom \Rightarrow f\ x = \ldots$
- ➜ similarly, conditiona 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 totalit
- ➜ instead derive equa form: $x \in f\_dom \Rightarrow f\ x = \ldots$
- ➜ similarly, conditiona 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 meth

1

Proving Termination

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

Three main proof meth

➜ **lexicographic_orde** tied by **fun**)

1

Proving Termination

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

Three main proof meth

- ➜ **lexicographic_orde** tied by **fun**)
- ➜ **size_change** (automated translation to simpler size-change graph[1])

[1] 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 $\forall x.\ x \in \mathit{fun\_name\_dom}$

Three main proof meth

- ➜ **lexicographic_orde** tied 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 i[illegible]ded induction holds

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

Well Founded Orders

Definition

$<_r$ is well founded i[illegible]ded induction holds

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

Well founded induction rule:

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

Well Founded Orders

Definition

$<_r$ is well founded i[illegible]ded induction holds

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

Well founded induction rule:

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

Alternative definition (equivalent):

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

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

$\mathsf{min}\ (<_r)\ Q\ x \quad \equiv \quad \forall y \in Q.\ y \not<_r x$

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

Well Founded Orders: Examples

➜ $<$ on $\mathbb{N}$ is well fou
well founded induct lete induction

Well Founded Orders: Examples

- $<$ on $\mathbb{N}$ is well fou
  well founded induct lete induction
- $>$ and $\leq$ on $\mathbb{N}$ are unded

Well Founded Orders: Examples

- $<$ on $\mathbb{N}$ is well fou
  well founded induct lete induction
- $>$ and $\leq$ on $\mathbb{N}$ are nded
- $x <_r y = x$ dvd $y \wedge$ N is well founded
  the minimal elements are the prime numbers

Well Founded Orders: Examples

- $<$ on $\mathbb{N}$ is well fou
  well founded induct lete induction
- $>$ and $\leq$ on $\mathbb{N}$ are nded
- $x <_r y = x$ dvd $y \wedge$, $\mathbb{N}$ 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 fou
  well founded induct lete induction
- $>$ and $\leq$ on $\mathbb{N}$ are nded
- $x <_r y = x$ dvd $y \wedge$, $\mathbb{N}$ 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
- $A <_r B = A \subset B \wedge$ finite B is well founded

Well Founded Orders: Examples

- $<$ on $\mathbb{N}$ is well fou
  well founded induct lete induction
- $>$ and $\leq$ on $\mathbb{N}$ are unded
- $x <_r y = x$ dvd $y \wedge$ 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
- $A <_r B = A \subset B \wedge$ 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 termi[illegible]hat about the recursion scheme?

Extracting the Recursion Scheme

So far for termi[illegible]hat about the recursion scheme?
No[illegible]more 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 termi[illegible]hat about the recursion scheme?
No[illegible]more 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) ⇝ n, Suc (Suc n) ⇝ Suc n

Extracting the Recursion Scheme

So far for termi[illegible]hat about the recursion scheme?

No[illegible]more 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)

Extracting the Recursion Scheme

So far for termi[illegible]hat about the recursion scheme?
No[illegible]more 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 =  ranch 'a tree list

fun treemap :: ('a → tree ⇒ 'a tree **where**
treemap fn (Leaf n) = n) |
treemap fn (Branch l) = Branch (map (treemap fn) l)

Extracting the Recursion Scheme

Higher Order:

➜ **datatype** 'a tree =  ranch 'a tree list

fun treemap :: ('a → tree ⇒ 'a tree **where**
treemap fn (Leaf n) = n) |
treemap fn (Branch l) = Branch (map (treemap fn) l)

Recursion: x ∈ set l ⟹ (fn, Branch l) ⇝ (fn, x)

Extracting the Recursion Scheme

Higher Order:

➜ **datatype** 'a tree =  ranch 'a tree list

fun treemap :: ('a → tree ⇒ 'a tree **where**
treemap fn (Leaf n) = n) |
treemap fn (Branch l) = Branch (map (treemap fn) l)

Recursion: x ∈ set l ⟹ (fn, Branch l) ⇝ (fn, x)

How does Isabelle extract context information for the call?

Extracting the Recursion Scheme

Ex 

ntext for equations

Extracting the Recursion Scheme

Ex ntext for equations

$\Rightarrow$

uence Rules!

Extracting the Recursion Scheme

Ex  ntext for equations

$\Rightarrow$

uence Rules!

Recall rule **if_cong**:

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

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

Extracting the Recursion Scheme

Ex 

ntext for equations

$\Rightarrow$

uence Rules!

Recall rule **if_cong**:

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

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 s  for function definitions.

(_rule[fundef_cong]

Congruence Rules for fun_defs

The s  for function definitions.

_rule[fundef_cong]

(if_ dy added by default)

Another example (higher-order):

[| xs = ys; ⋀x. x ∈ set ys ⟹ f x = g x |] ⟹ map f xs = map g ys

Congruence Rules for fun_defs

The s  for function definitions.

_rule[fundef_cong]

(if_ dy added by default)

Another example (higher-order):

[| xs = ys; ⋀x. x ∈ set ys ⟹ f x = g x |] ⟹ 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 today ...

- ➜ General recursion w**ction**
- ➜ Induction over recu**ns**
- ➜ How **fun** works
- ➜ Termination, partial functions, congruence rules