COMP4161: Advanced Topics in Software Verification

fun

Gerwin Klein, June Andronick, Ramana Kumar, Miki Tanaka

S2/2017

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]

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

© Data61, CSIRO: provided under Creative Commons Attribution License

General Recursion

The Choice

 © Data61, CSIRO: provided under Creative Commons Attribution License

General Recursion

The Choice

➜ Limited expressiveness, automatic termination

- primrec

 © Data61, CSIRO: provided under Creative Commons Attribution License

General Recursion

The Choice

➜ Limited expressiveness, automatic termination
  - primrec

➜ High expressiveness, termination proof may fail
  - fun

 © Data61, CSIRO: provided under Creative Commons Attribution License

General Recursion

The Choice

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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"
```

 © Data61, CSIRO: provided under Creative Commons Attribution License

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)"

 © Data61, CSIRO: provided under Creative Commons Attribution License

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)

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

© Data61, CSIRO: provided under Creative Commons Attribution License

fun — induction principle

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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**:

$$
\begin{array}{l}
[\![\ \bigwedge a.\ P\ a\ [\,];\\
\quad \bigwedge a\ w.\ P\ a\ [w]\\
\quad \bigwedge a\ x\ y\ zs.\ P\ a\ (y\#zs) \Longrightarrow P\ a\ (x\#y\#zs);\\
]\!] \Longrightarrow P\ a\ xs
\end{array}
$$

 © Data61, CSIRO: provided under Creative Commons Attribution License

Termination

Isabelle tries to prove termination automatically

➜ For most functions this works with a lexicographic termination relation.

 © Data61, CSIRO: provided under Creative Commons Attribution License

Termination

Isabelle tries to prove termination automatically

➜ For most functions this works with a lexicographic termination relation.

➜ Sometimes not

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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)

 © Data61, CSIRO: provided under Creative Commons Attribution License

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**

 © Data61, CSIRO: provided under Creative Commons Attribution License

Demo

How does fun/function work?

Recall **primrec**:

➜ defined one recursion operator per datatype D

 © Data61, CSIRO: provided under Creative Commons Attribution License

How does fun/function work?

Recall **primrec**:

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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.

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

How does fun/function work?

Similar strategy for **fun**:

➜ a new inductive definition for each **fun** f

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

© Data61, CSIRO: provided under Creative Commons Attribution License

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)

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

How does fun/function work?

Can separate and defer termination proof:

➜ skip proof of totality

© Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

Proving Termination

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

Three main proof methods:

 © Data61, CSIRO: provided under Creative Commons Attribution License

Proving Termination

Command **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**)

 © Data61, CSIRO: provided under Creative Commons Attribution License

Proving Termination

Command **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** (different automated technique)

 © Data61, CSIRO: provided under Creative Commons Attribution License

Proving Termination

Command **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** (different automated technique)
- **relation R** (manual proof via well-founded relation)

 © Data61, CSIRO: provided under Creative Commons Attribution License

Well Founded Orders

Definition

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

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

© Data61, CSIRO: provided under Creative Commons Attribution License

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:

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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:

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

Alternative definition (equivalent):

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

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

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

Well Founded Orders: Examples

- $<$ on $\mathbb{N}$ is well founded
  well founded induction = complete induction

 © Data61, CSIRO: provided under Creative Commons Attribution License

Well Founded Orders: Examples

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

© Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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*

© Data61, CSIRO: provided under Creative Commons Attribution License

Extracting the Recursion Scheme

So far for termination. What about the recursion scheme?

 © Data61, CSIRO: provided under Creative Commons Attribution License

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)

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

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)

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

Extracting the Recursion Scheme

Higher Oder:

➜ **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)

 © Data61, CSIRO: provided under Creative Commons Attribution License

Extracting the Recursion Scheme

Higher Oder:

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

Extracting the Recursion Scheme

Higher Oder:

➜ **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 to extract the context information for the call?

 © Data61, CSIRO: provided under Creative Commons Attribution License

Extracting the Recursion Scheme

Extracting context for equations

© Data61, CSIRO: provided under Creative Commons Attribution License

Extracting the Recursion Scheme

Extracting context for equations

$\Rightarrow$

Congruence Rules!

© Data61, CSIRO: provided under Creative Commons Attribution License

Extracting the Recursion Scheme

Extracting context for equations

$\Rightarrow$

Congruence Rules!

Recall rule **if_cong**:

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

Extracting the Recursion Scheme

Extracting context for equations

$\Rightarrow$

Congruence 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 $\neg b$.

 © Data61, CSIRO: provided under Creative Commons Attribution License

Extracting the Recursion Scheme

Extracting context for equations

$\Rightarrow$

Congruence 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 $\neg b$.

 © Data61, CSIRO: provided under Creative Commons Attribution License

Congruence Rules for fun_defs

The same works for function definitions.

declare my_rule[fundef_cong]

 © Data61, CSIRO: provided under Creative Commons Attribution License

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

© Data61, CSIRO: provided under Creative Commons Attribution License

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

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

 © Data61, CSIRO: provided under Creative Commons Attribution License

Demo

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`

 © Data61, CSIRO: provided under Creative Commons Attribution License

We have seen today ...

➜ General recursion with **fun**/**function**

 Data61, CSIRO: provided under Creative Commons Attribution License

We have seen today ...

- General recursion with **fun**/**function**
- Induction over recursive functions

© Data61, CSIRO: provided under Creative Commons Attribution License

We have seen today ...

- ➜ General recursion with **fun**/**function**
- ➜ Induction over recursive functions
- ➜ How **fun** works

© Data61, CSIRO: provided under Creative Commons Attribution License

We have seen today ...

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

 © Data61, CSIRO: provided under Creative Commons Attribution License