COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Simon Winwood, Toby Murray, June Andronick, Gerwin Klein

wf_rec
Content

➜ Intro & motivation, getting started with Isabelle
➜ Foundations & Principles
  • Lambda Calculus
  • Higher Order Logic, natural deduction
  • Term rewriting
➜ Proof & Specification Techniques
  • Inductively defined sets, rule induction
  • Datatypes, recursion, induction
  • More recursion, Calculational reasoning
  • Hoare logic, proofs about programs
  • Locales, Presentation
The Choice
General Recursion

The Choice

- Limited expressiveness, automatic termination
  - primrec
The Choice

→ Limited expressiveness, automatic termination
  - primrec

→ High expressiveness, termination proof may fail
  - fun
General Recursion

The Choice

→ Limited expressiveness, automatic termination
  ● primrec

→ High expressiveness, termination proof may fail
  ● fun

→ High expressiveness, tweakable, termination proof manual
  ● function
fun sep :: "'a ⇒ 'a list ⇒ 'a list"
where
  "sep a (x # y # zs) = x # a # sep a (y # zs)" | 
  "sep a xs = xs"
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

The definition:

- 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)
The definition:

- 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
The definition:

- 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 have fail to prove automation:

- use **function** (**sequential**) instead
- allows to prove termination manually
Each `fun` definition induces an induction principle
Each **fun** definition induces an induction principle

For each equation:

- show that the property holds for the lhs provided it holds for each recursive call on the rhs
Each **fun** definition induces an induction principle

For each equation:

show that the property holds for the lhs provided it holds for each recursive call on the rhs

**Example sep.induct:**

\[
\begin{align*}
\forall a. P a [] ; \\
\forall a w. P a [w] \\
\forall a x y zs. P a (y#zs) \implies P a (x#y#zs); \\
\implies P a xs
\end{align*}
\]
Isabelle tries to prove termination automatically

→ For most functions this works with a lexicographic termination relation.
Isabelle tries to prove termination automatically

→ For most functions this works with a lexicographic termination relation.
→ Sometimes not
Isabelle tries to prove termination automatically

- For most functions this works with a lexicographic termination relation.
- Sometimes not $\Rightarrow$ error message with unsolved subgoal
Isabelle tries to prove termination automatically

- For most functions this works with a lexicographic termination relation.
- Sometimes not ⇒ error message with unsolved subgoal
- You can prove automation separately.

```plaintext
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)
```
Termination

Isabelle tries to prove termination automatically

- For most functions this works with a lexicographic termination relation.
- Sometimes not ⇒ error message with unsolved subgoal
- You can prove automation separately.

```plaintext
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
```
DEMO
How does fun/function work?

We need: general recursion operator
How does fun/function work?

We need: general recursion operator

something like: \( \text{rec } F = F (\text{rec } F) \)
How does fun/function work?

We need: general recursion operator

something like: \( 	ext{rec } F = F \circ \text{rec } F \) 

(\( F \) stands for the recursion equations)

Example:
How does fun/function work?

**We need:** general recursion operator

**something like:** $rec \ F = F \ (rec \ F)$

($F$ stands for the recursion equations)

**Example:**

$\Rightarrow$ recursion equations: $f \ 0 = 0 \quad f \ (Suc \ n) = f \ n$
How does fun/function work?

**We need:** general recursion operator

**something like:**

\[ \text{rec } F = F \left( \text{rec } F \right) \]

\((F\) stands for the recursion equations\)

**Example:**

→ recursion equations: \( f \ 0 = 0 \quad f \ (\text{Suc } n) = f \ n \)

→ as one \( \lambda \)-term: \( f = \lambda n'. \ \text{case } n' \ \text{of} \ 0 \Rightarrow 0 \ | \ \text{Suc } n \Rightarrow f \ n \)
How does fun/function work?

We need: general recursion operator

something like: \[ \text{rec } F = F \left( \text{rec } F \right) \]

(F stands for the recursion equations)

Example:

⇒ recursion equations: \[ f \ 0 = 0 \quad f \ (\text{Suc } n) = f \ n \]
⇒ as one \( \lambda \)-term: \[ f = \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow f \ n \]
⇒ functor: \[ F = \lambda f. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow f \ n \]
How does fun/function work?

We need: general recursion operator

something like: \( \text{rec } F = F (\text{rec } F) \)

\( (F \text{ stands for the recursion equations}) \)

Example:

\( \rightarrow \) recursion equations: \( f 0 = 0 \quad f (\text{Suc } n) = f n \)

\( \rightarrow \) as one \( \lambda \)-term: \( f = \lambda n’. \text{case } n’ \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow f n \)

\( \rightarrow \) functor: \( F = \lambda f. \lambda n’. \text{case } n’ \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow f n \)

\( \rightarrow \) \( \text{rec} :: ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta)) \Rightarrow (\alpha \Rightarrow \beta) \) like above cannot exist in HOL (only total functions)

\( \rightarrow \) But 'guarded' form possible: \( \text{wfre}c :: (\alpha \times \alpha) \text{ set } \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta)) \Rightarrow (\alpha \Rightarrow \beta) \)

\( \rightarrow (\alpha \times \alpha) \text{ set a well founded order, decreasing with execution} \)
How does fun/function work?

Why $\text{rec } F = F \left( \text{rec } F \right)$?
How does fun/function work?

Why \( \text{rec } F = F (\text{rec } F) \)?

Because we want the recursion equations to hold.

Example:

\[
\begin{align*}
F & \equiv \lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow g\ n \\
\text{f} & \equiv \text{rec } F
\end{align*}
\]
How does fun/function work?

Why \( \text{rec } F = F (\text{rec } F) \)?

Because we want the recursion equations to hold.

Example:

\[
F \equiv \lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 | \text{Suc } n \Rightarrow g \ n
\]

\[
f \equiv \text{rec } F
\]

\[
f \ 0 = \text{rec } F \ 0
\]
How does fun/function work?

Why \( \text{rec } F = F (\text{rec } F) \)?

Because we want the recursion equations to hold.

Example:

\[
F \equiv \lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow g \ n
\]

\[
f \equiv \text{rec } F
\]

\[
f \ 0 = \text{rec } F \ 0
\]

\[
\ldots = F (\text{rec } F) \ 0
\]
How does fun/function work?

Why \( \text{rec } F = F (\text{rec } F) \)?

Because we want the recursion equations to hold.

Example:

\[
F \equiv \lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow g \ n
\]

\[
f \equiv \text{rec } F
\]

\[
f \ 0 \ = \ \text{rec } F \ 0
\]

\[
\ldots \ = \ F (\text{rec } F) \ 0
\]

\[
\ldots \ = \ (\lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow g \ n) \ (\text{rec } F) \ 0
\]
How does fun/function work?

Why \( \text{rec } F = F (\text{rec } F) \)?

Because we want the recursion equations to hold.

Example:

\[
F \equiv \lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow g \ n
\]

\[
f \equiv \text{rec } F
\]

\[
f \ 0 \ = \ \text{rec } F \ 0
\]

\[
\ldots \ = \ F \ (\text{rec } F) \ 0
\]

\[
\ldots \ = \ (\lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow g \ n) \ (\text{rec } F) \ 0
\]

\[
\ldots \ = \ (\text{case } 0 \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow \text{rec } F \ n)
\]
How does fun/function work?

Why \( \text{rec } F = F \ (\text{rec } F) \)?

**Because we want the recursion equations to hold.**

**Example:**

\[
F \equiv \lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow g \ n
\]

\[
f \equiv \text{rec } F
\]

\[
f0 = \text{rec } F0
\]

\[
= F \ (\text{rec } F) \ 0
\]

\[
= (\lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow g \ n) \ (\text{rec } F) \ 0
\]

\[
= (\text{case } 0 \text{ of } 0 \Rightarrow 0 \mid \text{Suc } n \Rightarrow \text{rec } F \ n)
\]

\[
= 0
\]
Definition

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

$\text{wf } r \equiv \forall P. (\forall x. (\forall y <_r x. P y) \rightarrow P x) \rightarrow (\forall x. P x)$
Well Founded Orders

Definition

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

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

Well founded induction rule:

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

$$P a$$
WellFounded Orders

Definition

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

\[ \text{wf } r \equiv \forall P. (\forall x. (\forall y <_r x. P y) \rightarrow P x) \rightarrow (\forall x. P x) \]

Well founded induction rule:

\[
\text{wf } r \quad \land \quad x. (\forall y <_r x. P y) \quad \Rightarrow \quad P x
\]

Alternative definition (equivalent):

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

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

\[ \min r \ Q \ x \quad \equiv \quad \forall y \in Q. \ y \not<_r x \]

\[ \text{wf } r \quad = \quad (\forall Q \neq \{\}. \ \exists m \in Q. \ \min r \ Q \ m) \]
Well Founded Orders: Examples

$\rightarrow$ < 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
- $x <_r y = x \text{ dvd } y \land x \neq 1$ 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

- $x <_r y = x \text{ dvd } y \land x \neq 1$ on $\mathbb{N}$ is well founded
  - the minimal elements are the prime numbers

- $(a, b) <_r (x, y) = a <_1 x \lor a = x \land b <_2 y$ is well founded
  - if $<_1$ and $<_2$ are
Well Founded Orders: Examples

$\rightarrow$ $<$ on $\mathbb{N}$ is well founded
  well founded induction = complete induction
$\rightarrow$ $> \text{ and } \leq \text{ on } \mathbb{N}$ are **not** well founded
$\rightarrow$ $x <_r y = x \text{ dvd } y \land x \neq 1$ on $\mathbb{N}$ is well founded
  the minimal elements are the prime numbers
$\rightarrow$ $(a, b) <_r (x, y) = a <_1 x \lor a = x \land b <_2 y$ is well founded
  if $<_1$ and $<_2$ are
$\rightarrow$ $A <_r B = A \subset B \land \text{ finite } B$ is well founded

12-d
Well Founded Orders: Examples

→ < on \( \mathbb{N} \) is well founded
  well founded induction = complete induction

→ > and \( \leq \) on \( \mathbb{N} \) are \textbf{not} well founded

→ \( x <_r y = x \text{ dvd } y \land x \neq 1 \) on \( \mathbb{N} \) is well founded
  the minimal elements are the prime numbers

→ \((a, b) <_r (x, y) = a <_1 x \lor a = x \land b <_2 y\) is well founded
  if \( <_1 \) and \( <_2 \) are

→ \( A <_r B = A \subset B \land \text{finite } B \) is well founded

→ \( \subseteq \) and \( \subset \) in general are \textbf{not} well founded

More about well founded relations: Term Rewriting and All That
The Recursion Operator

Back to recursion: $\text{rec } F = F (\text{rec } F)$ not possible

Idea:
The Recursion Operator

**Back to recursion:** $rec F = F (rec F)$ not possible

**Idea:** have $wfrec R F$ where $R$ is well founded
The Recursion Operator

Back to recursion: \( \text{rec } F = F \ (\text{rec } F) \) not possible

Idea: have \( \text{wfrec } R \ F \) where \( R \) is well founded

Cut:
- only do recursion if parameter decreases \( \text{wrt } R \)
- otherwise: abort
The Recursion Operator

Back to recursion: \( \text{rec } F = F \ (\text{rec } F) \) not possible

Idea: have \( \text{wfrec } R \ F \) where \( R \) is well founded

Cut:

- only do recursion if parameter decreases wrt \( R \)
- otherwise: abort
- arbitrary :: \( \alpha \)
  - cut :: \( \alpha \implies \beta \implies (\alpha \times \alpha) \text{ set } \implies \alpha \implies (\alpha \implies \beta) \)
  - cut \( G \ R \ x \equiv \lambda y. \text{ if } (y, x) \in R \text{ then } G \ y \text{ else arbitrary} \)
The Recursion Operator

Back to recursion: \( \text{rec } F = F \ (\text{rec } F) \) not possible

Idea: have \( \text{wfrec } R \ F \) where \( R \) is well founded

Cut:

- only do recursion if parameter decreases wrt \( R \)
- otherwise: abort
- arbitrary :: \( \alpha \)

\[
\text{cut} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha \times \alpha) \set \Rightarrow \alpha \Rightarrow (\alpha \Rightarrow \beta)
\]

\( \text{cut } G \ R \ x \equiv \lambda y. \text{if} \ (y, x) \in R \ \text{then} \ G \ y \ \text{else arbitrary} \)

\[
\text{wf } R \Rightarrow \text{wfrec } R \ F \ x = F \ (\text{cut} \ (\text{wfrec } R \ F) \ R \ x) \ x
\]
The Recursion Operator

Admissible recursion

→ recursive call for $x$ only depends on parameters $y <_R x$
→ describes exactly one function if $R$ is well founded
Admissible recursion

- recursive call for $x$ only depends on parameters $y <_R x$
- describes exactly one function if $R$ is well founded

\[
\text{adm}_w f \equiv \forall f, g, x. (\forall z. (z, x) \in R \rightarrow f z = g z) \rightarrow F f x = F g x
\]
Admissible recursion

→ recursive call for \( x \) only depends on parameters \( y <_R x \)
→ describes exactly one function if \( R \) is well founded

\[
\text{adm}_w \text{f} \ R \ F \equiv \forall f \ g \ x. \ (\forall z. \ (z, x) \in R \rightarrow f \ z = g \ z) \rightarrow F \ f \ x = F \ g \ x
\]

Definition of \( \text{wf}_\text{rec} \): again first by induction, then by epsilon

\[
\forall z. \ (z, x) \in R \rightarrow (z, g \ z) \in \text{wfrec}_\text{rel} \ R \ F
\]

\[
(x, F \ g \ x) \in \text{wfrec}_\text{rel} \ R \ F
\]
Admissible recursion

- recursive call for \( x \) only depends on parameters \( y \prec_R x \)
- describes exactly one function if \( R \) is well founded

\[
\text{adm}_{wf} \ R \ F \equiv \forall f \ g \ x. (\forall z. (z, x) \in R \rightarrow f \ z = g \ z) \rightarrow F \ f \ x = F \ g \ x
\]

**Definition of \text{wf}_\text{rec}:** again first by induction, then by epsilon

\[
\forall z. (z, x) \in R \rightarrow (z, g \ z) \in \text{wfrec}_\text{rel} \ R \ F
\]

\[
(x, F \ g \ x) \in \text{wfrec}_\text{rel} \ R \ F
\]

\[
\text{wfrec} \ R \ F \ x \equiv \text{THE} \ y. (x, y) \in \text{wfrec}_\text{rel} \ R \ (\lambda f \ x. F \ (\text{cut} \ f \ R \ x) \ x)
\]

More: John Harrison, *Inductive definitions: automation and application*
DEMO