COMP4161
Advanced Topics in Software Verification

fun

Thomas Sewell, Miki Tanaka, Rob Sison

T3/2024

Content

- Foundations & Principles
  - Intro, Lambda calculus, natural deduction [1,2]
  - Higher Order Logic, Isar (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]
  - Proof automation, Isar (part 2) [8[b]]
  - 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

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)"

fun

- Much 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 and datatype size)

- Generates more theorems than **primrec**

- May fail to prove termination:
  - use **function** instead
    - **function(sequential)** preserves sequential behaviour
  - allows you to prove termination manually

Demo

Why Termination?

Why does it matter that our recursive function definitions terminate?

- Because otherwise we might introduce unsoundness.
- We talked about this when we introduced **primrec**.

MINI-DEMO

Conservative Extensions

These are some **definitional mechanisms** of Isabelle/HOL:

- **definition**
- **primrec**
- **inductive**
- **datatype** (sort of)
- **fun**
- **function**

They all add a new constant (or constants) and their defining facts.

They all try to make a **conservative extension** of the logic:

- new symbols, thus new type-correct statements
- some of these new statements are provable
- previously type-correct statements **should not change meaning**

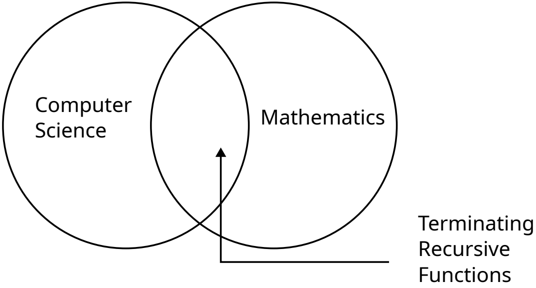
A Dramatic Aside

Ondřej Kunčar and Andrei Popescu
A Consistent Foundation for Isabelle/HOL
In ITP 2015, Nanjing

`https://andreipopescu.uk/pdf/ITP2015.pdf`

- discusses a (debatable) proof of `False` in Isabelle 2014.

Terminating Functions in the Intersection



If a recursive computational function *f* $:: \alpha \Rightarrow \beta$ terminates, then its type in the logic can be *f* $:: \alpha \Rightarrow \beta$.

Termination as Induction

Termination (of a recursive scheme) is $\approx$ induction.

- ➜ 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**:

$$\llbracket \bigwedge a.\ P\ a\ [];$$
$$\bigwedge a\ w.\ P\ a\ [w]$$
$$\bigwedge a\ x\ y\ zs.\ P\ a\ (y\#zs) \Longrightarrow P\ a\ (x\#y\#zs);$$
$$\rrbracket \Longrightarrow P\ a\ xs$$

Termination

Isabelle tries to prove termination automatically

- ➜ For most functions 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 [$y \leftarrow xs.y \leq x$]@[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? Option 1.

You may remember the previous explanation of how the **rec_list** constant (used by **primrec**) is defined via a relation.

For **fun** $f :: \alpha \Rightarrow \beta$, first define $f_{rel} :: (\alpha \times \beta)$ set.

- ➜ extract *recursion scheme* for equations in f
- ➜ define graph f_{rel} inductively, encoding recursion scheme
  - $f\ (\text{Suc}\ x) = f\ x * 2 \mapsto (f_{rel} x v \longrightarrow f_{rel} (\text{Suc}\ x)(v * 2))$
- ➜ prove totality (= termination)
  - recall that inductive relations are the least fixpoint
  - nonterminating recursion chains are not in the set
- ➜ prove uniqueness (automatic)
- ➜ derive f and original equations from ϵ choice and f_{rel}
- ➜ export induction scheme from f_{rel}

How does fun/function work? Option 2.

function 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
- ➜ note that for $f :: \alpha \Rightarrow \beta$, this $f\_rel :: (\alpha \times \alpha)$ set.

Demo

Proving Termination

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** (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 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):
the accessible part is everything, or (equivalent):
every nonempty set has a minimal element wrt $<_r$

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

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 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 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: $x \neq 0 \Longrightarrow x \leadsto x - 1$

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

How does Isabelle extract context information for the call?

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 $\neg b$.

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 $\in$ set ys $\Longrightarrow$ f x = g x $]\!]$ $\Longrightarrow$ 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`

Ondřej Kunčar and Andrei Popescu
A Consistent Foundation for Isabelle/HOL
In ITP 2015

`https://andreipopescu.uk/pdf/ITP2015.pdf`

Rob Arthan
HOL constant definition done right
In ITP 2014

We have see today ...

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