Verification of Functional Programs
Defining Functions in Isabelle/HOL

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2026-1

Introduction

Introduction

In `Isabelle/HOL` the functions must be total.

Introduction

In `Isabelle/HOL` the functions must be total.

Functions can be defined using the following commands:

- **`primrec`**,
- **`fun`**,
- **`function`**.

Lexicographic Ordering

Definition

Let $(A, \preccurlyeq_A)$ and $(B, \preccurlyeq_B)$ be two posets. The **lexicographic ordering** $\preccurlyeq$ on $A \times B$ is defined by

$$(a_1, b_1) \preccurlyeq (a_2, b_2) \coloneqq a_1 \prec_A a_2 \text{ or } (a_1 = a_2 \text{ and } b_1 \preccurlyeq_B b_2).$$

The `primrec` Command

The `primrec` Command

Description

> *"The keyword* ***`primrec`*** *indicates that the recursion is of a particularly primitive kind where each recursive call peels off a datatype constructor from one of the arguments. Thus the recursion always terminates, i.e. the function is total."*
> *(Nipkow, Paulson, and Wenzel 2002, p. 11)*

The `primrec` Command

Example (structural recursion on one argument)

The reverse function on polymorphic lists (Nipkow, Paulson, and Wenzel [2002] 2026).

```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []       = []"
| "rev (x # xs) = (rev xs) @ (x # [])"
```

The **primrec** Command

Example (structural recursion on one argument)

The reverse function on polymorphic lists (Nipkow, Paulson, and Wenzel [2002] 2026).

```
1 primrec rev :: "'a list ⇒ 'a list" where
2   "rev []       = []"
3 | "rev (x # xs) = (rev xs) @ (x # [])"
```

The function is structurally recursive because in the (only) recursive call in line 3,

$$xs \prec x \# xs.$$

The `primrec` Command

Example (structural recursion on one of two arguments)

The append function on polymorphic lists (Nipkow, Paulson, and Wenzel [2002] 2026).

```
1 primrec app :: "'a list ⇒ 'a list ⇒ 'a list" where
2   "app []       ys = ys"
3 | "app (x # xs) ys = x # (app xs ys)"
```

The **primrec** Command

Example (structural recursion on one of two arguments)

The append function on polymorphic lists (Nipkow, Paulson, and Wenzel [2002] 2026).

```
1 primrec app :: "'a list ⇒ 'a list ⇒ 'a list" where
2   "app []       ys = ys"
3 | "app (x # xs) ys = x # (app xs ys)"
```

The function is structurally recursive because in the (only) recursive call in line 3,

$$(xs, ys) \prec (x \# xs, ys).$$

The `primrec` Command

Counterexample (structural recursion on non-primitive patterns)

The Fibonacci function cannot be defined using `primrec`.

```
1 primrec fib :: "nat ⇒ nat" where
2   "fib 0             = 1"
3 | "fib (Suc 0)       = 1"
4 | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Error:

```
Nonprimitive pattern in left-hand side at
  fib (Suc 0) = 1
```

The `primrec` Command

Counterexample (structural recursion with lexicographic order)

The Ackermann–Péter function cannot be defined using `primrec`.

```
1 primrec ack :: "nat ⇒ nat ⇒ nat" where
2   "ack 0       n       = Suc n"
3 | "ack (Suc m) 0       = ack m 1"
4 | "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"
```

Error:

```
More than one non-variable argument in left-hand side at
  ack (Suc m) 0 = ack m 1
```

The `primrec` Command

Example (structural recursion on higher-order data types)

The add function on the so-called Brouwer ordinals.

```
1 datatype on = Z | S on | L "(nat ⇒ on)"
2
3 primrec add :: "on ⇒ on ⇒ on" where
4   "add α Z     = α"
5 | "add α (S β) = S (add α β)"
6 | "add α (L f) = L (λ n. add α (f n))"
```

(continued on next slide)

The `primrec` Command

Example (continuation)

The function is structurally recursive because (Abel and Altenkirch 2002)

(i) in the recursive call in line 5,

$$(\alpha, \beta) \prec (\alpha, \mathrm{S}\ b),$$

(ii) in the recursive call in line 6,

$$(\alpha, f\ n) \preccurlyeq (\alpha, f) \prec (\alpha, \mathrm{L}\ f).$$

The `primrec` Command

Example (mutual recursion on mutual data types)

Arithmetic and booleans expressions defined by a mutual datatype (Nipkow, Paulson, and Wenzel [2002] 2026).

```
datatype 'a aexp = IF "'a bexp" "'a aexp" "'a aexp"
                 | Sum "'a aexp" "'a aexp"
                 | Diff "'a aexp" "'a aexp"
                 | Var 'a
                 | Num nat
and
         'a bexp = Less "'a aexp" "'a aexp"
                 | And "'a bexp" "'a bexp"
                 | Neg "'a bexp"
```

(continued on next slide)

The `primrec` Command

Example (continuation)

Evaluation of arithmetic and booleans expressions by mutual recursion.

```
1  primrec evala :: "'a aexp ⇒ ('a ⇒ nat) ⇒ nat" and
2          evalb :: "'a bexp ⇒ ('a ⇒ nat) ⇒ bool"
3  where
4    "evala (IF b a1 a2) env =
5       (if evalb b env then evala a1 env else evala a2 env)"
6  | "evala (Sum a1 a2)  env = evala a1 env + evala a2 env"
7  | "evala (Diff a1 a2) env = evala a1 env - evala a2 env"
8  | "evala (Var v)      env = env v"
9  | "evala (Num n)      env = n"
10
11 | "evalb (Less a1 a2) env = (evala a1 env < evala a2 env)"
12 | "evalb (And b1 b2)  env = (evalb b1 env ∧ evalb b2 env)"
13 | "evalb (Neg b)      env = (¬ evalb b env)"
```

The `fun` Command

The `fun` Command

Description

The `fun` command is a short form of the `function` command.[†]

$$\left[\begin{array}{l} \textbf{fun } \texttt{f} :: \tau \\ \textbf{where} \\ \quad \textit{equations} \\ \quad \vdots \end{array}\right] \equiv \left[\begin{array}{l} \textbf{function (sequential) } \texttt{f} :: \tau \\ \textbf{where} \\ \quad \textit{equations} \\ \quad \vdots \\ \textbf{by } \texttt{pat\_completeness auto} \\ \textbf{termination by } \texttt{lexicographic\_order} \end{array}\right]$$

[†] Figure from (Krauss 2026, p. 4).

The `fun` Command

Example (structural recursion on non-primitive patterns)

The Fibonacci function (Nipkow, Paulson, and Wenzel [2002] 2026).

```
1 fun fib :: "nat ⇒ nat" where
2   "fib 0             = 1"
3 | "fib (Suc 0)       = 1"
4 | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

```
Found termination order: "size <*mlex*> {}"
```

The `fun` Command

Example (continuation)

The function is structurally recursive because

(i) in the second recursive call in line 3,

$$\text{Suc } n \prec \text{Suc } (\text{Suc } n),$$

(ii) in the first recursive call in line 3,

$$x \prec \text{Suc } n \prec \text{Suc } (\text{Suc } n).$$

The `fun` Command

Example (structural recursion with lexicographic order)

The Ackermann–Péter function.

```
1 fun ack :: "nat ⇒ nat ⇒ nat" where
2   "ack 0       n       = Suc n"
3 | "ack (Suc m) 0       = ack m 1"
4 | "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"
```

```
Found termination order:
"(λp. size (fst p)) <*mlex*> (λp. size (snd p)) <*mlex*> {}"
```

(continued on next slide)

The `fun` Command

Example (continuation)

The function `ack m n` is structurally recursive with the lexicographic order on (m, n) because

(i) in the recursive call in line 3,

$$(m, 1) \prec (\text{Suc } m, 0),$$

(ii) in the first recursive call in line 4,

$$(m, \text{ack } (\text{Suc } m)\ n) \prec (\text{Suc } m, \text{Suc } n),$$

(iii) in the second recursive call in line 4,

$$(\text{Suc } m, n) \prec (\text{Suc } m, \text{Suc } n).$$

The `fun` Command

Example (structural recursion with lexicographic order)

The Ackermann–Péter function changing the order of arguments (Nipkow, Paulson, and Wenzel [2002] 2026).

```
1 fun ack2 :: "nat ⇒ nat ⇒ nat" where
2   "ack2 m       0       = Suc m"
3 | "ack2 0       (Suc n) = ack2 1 n"
4 | "ack2 (Suc m) (Suc n) = ack2 (ack2 m (Suc n)) n"
```

```
Found termination order:
"(λp. size (snd p)) <*mlex*> (λp. size (fst p)) <*mlex*> {}"
```

(continued on next slide)

The `fun` Command

Example (continuation)

The function `ack2 m n` is structurally recursive with the lexicographic order on (n, m) because

(i) in the recursive call in line 3,

$$(n, 1) \prec (\text{Suc } n, 0),$$

(ii) in the first recursive call in line 4,

$$(n, \text{ack } m\ (\text{Suc } n)) \prec (\text{Suc } n, \text{Suc } m),$$

(iii) in the second recursive call in line 4,

$$(\text{Suc } n, m) \prec (\text{Suc } n, \text{Suc } m).$$

The `fun` Command

Example (mutual recursion)

Functions even and odd by mutual recursion (Krauss 2026).

```
1 fun even :: "nat ⇒ bool" and
2     odd  :: "nat ⇒ bool"
3 where
4   "even 0       = True"
5 | "odd  0       = False"
6 | "even (Suc n) = odd n"
7 | "odd  (Suc n) = even n"
```

```
Found termination order: "case_sum size size <*mlex*> {}"
```

The `fun` Command

Example (higher-order recursion)

The mirror function on Rose trees.

```
1 datatype 'a rose_tree = Node "'a rose_tree list"
2
3 fun mirror :: "'a rose_tree ⇒ 'a rose_tree" where
4   "mirror (Node ts) = Node (rev (map mirror ts))"
```

```
Found termination order: "size <*mlex*> {}"
```

The `function` Command

The `function` Command

Example (using a measure function)

The Fibonacci function.

```
1 function fib :: "nat ⇒ nat" where
2   "fib 0             = 1"
3 | "fib (Suc 0)       = 1"
4 | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
5 by pat_completeness auto
6 termination fib
7   apply (relation "measure (λn. n)")
8   by auto
```

The `function` Command

Example (using a measure function)

The Ackermann–Péter.

```
1 function ack :: "nat ⇒ nat ⇒ nat" where
2   "ack 0       n       = Suc n"
3 | "ack (Suc m) 0       = ack m 1"
4 | "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"
5   by pat_completeness auto
6 termination
7   apply (relation "measures [fst , snd]")
8   by auto
```

The `function` Command

Example (mutual recursion)

Functions even and odd by mutual recursion (Krauss 2026).

```
1  function even :: "nat ⇒ bool" and
2           odd  :: "nat ⇒ bool"
3  where
4    "even 0       = True"
5  | "odd  0       = False"
6  | "even (Suc n) = odd n"
7  | "odd  (Suc n) = even n"
8  by pat_completeness auto
9  termination
10   by (relation "measure (λx. case x of Inl n ⇒ n | Inr n ⇒ n)") auto
```

"`Isabelle` internally creates a single function operating on the sum type `nat + nat` . . . Consequently, termination has to be proved simultaneously for both functions, by specifying a measure on the sum type." (Krauss 2026, p. 7)

The `function` Command

Example

An example where the lexicographic order fails for the termination proof (Krauss 2026).

```
function sum :: "nat ⇒ nat ⇒ nat" where
  "sum i N = (if i > N then 0 else i + sum (Suc i) N)"
by pat_completeness auto
termination sum
  apply (relation "measure (λ(i,N). N + 1 - i)")
  by auto
```

The `function` Command

Example

An example where using something different to `pat_completeness` (Krauss 2026).

```
function gcd :: "nat ⇒ nat ⇒ nat" where
  "gcd x 0 = x"
| "gcd 0 y = y"
| "x < y   ⟹ gcd (Suc x) (Suc y) = gcd (Suc x) (y - x)"
| "~ x < y ⟹ gcd (Suc x) (Suc y) = gcd (x - y) (Suc y)"
by (atomize_elim, auto, arith)
termination by lexicographic_order
```

References

References

Andreas Abel and Thorsten Altenkirch (2002). A Predicative Analysis of Structural Recursion. Journal of Functional Programming 12.1, pp. 1–41. DOI: 10.1017/S0956796801004191 (cit. on p. 15).

Alexander Krauss (Jan. 18, 2026). Defining Recursive Functions in Isabelle/HOL. URL: https://isabelle.in.tum.de/dist/Isabelle2025-2/doc/functions.pdf (cit. on pp. 19, 26, 31–33).

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel (2002). Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Vol. 2283. Lecture Notes in Computer Science. Springer. DOI: 10.1007/3-540-45949-9 (cit. on pp. 7, 35).

— [2002] (Jan. 18, 2026). Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Updated version of (Nipkow, Paulson, and Wenzel 2002) matching changes in Isabelle/HOL. URL: https://isabelle.in.tum.de/dist/Isabelle2025-2/doc/tutorial.pdf (cit. on pp. 8–11, 16, 20, 24).