Proofs on Inductive Predicates in Why3

Jean-Christophe Filliâtre, Andrei Paskevich, Henri Saudubray

To cite this version:

Jean-Christophe Filliâtre, Andrei Paskevich, Henri Saudubray. Proofs on Inductive Predicates in Why3. Big Specification: Specification, Proof, and Testing at Scale, Neel Krishnaswami, Peter Sewell, Natarajan Shankar, Oct 2024, Cambridge, United Kingdom. ⟨hal-04734466⟩

HAL Id: hal-04734466
https://hal.science/hal-04734466v1

Submitted on 14 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

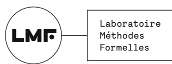
L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

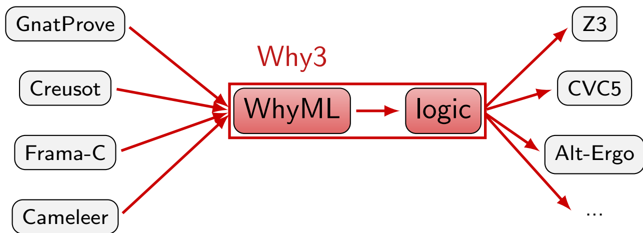
HAL Authorization

Proofs on Inductive Predicates in Why3

Jean-Christophe Filliâtre, Andrei Paskevich, Henri Saudubray

Isaac Newton Institute
Big Specification
Cambridge, October 8, 2024




GnatProve
Creusot
Frama-C
Cameleer
Why3
WhyML
logic
Z3
CVC5
Alt-Ergo
...

1. the logic of Why3
2. proofs by induction
3. a proposal: `match inductive`
4. an application

this is a logic of compromise:

- expressiveness is good for users
- simplicity is good for SMT solvers (and for users, too)

we end up with an extension of first-order logic of our own

One logic to use them all [JCF, CADE 13]

A Formalization of Core Why3 in Coq [Cohen, Johnson-Freyd, POPL 24]

- type polymorphism
- algebraic data types

```
type list 'a = Nil | Cons 'a (list 'a)
```

```
constant alist: list int =
  Cons 34 (Cons 55 (Cons 89 Nil))
```

- type polymorphism • algebraic data types • pattern matching
- recursive functions and predicates (with termination check)

```
function length (l: list 'a) : int
= match l with
  | Nil      -> 0
  | Cons _ t -> 1 + length t
  end
```

```
lemma L:
  forall l: list 'a. length l >= 0
```

- type polymorphism • algebraic data types • pattern matching
- recursive functions and predicates (with termination check)
- inductive predicates

```
inductive nim_step int int =
| RemoveOne: forall n. n >= 0 -> nim_step (n+1) n
| RemoveTwo: forall n. n >= 0 -> nim_step (n+2) n

inductive nim int int =
| Step: forall x y. nim_step x y -> nim x y
| Path: forall x y z. nim x y -> nim y z -> nim x z
```

```
lemma L:
  forall x y. nim x y -> 0 <= y < x
```

using SMT solvers

when it comes to calling SMT solvers, the logic of Why3 is translated to first-order logic

which means encoding

- polymorphism
- algebraic data types
- pattern matching

possibly losing completeness in the process

an algebraic data type such as

```
type list 'a = Nil | Cons 'a (list 'a)
```

is sent to SMT solvers as

- an uninterpreted type `list`
- a constant `Nil` and a function `Cons`
- injectivity + inversion principle
- but no induction principle

similarly, an inductive predicate such as

```
inductive nim int int =
| Step: forall x y. nim_step x y -> nim x y
| Path: forall x y z. nim x y -> nim y z -> nim x z
```

is sent to SMT solvers as

- an undefined predicate `nim`
- two axioms `Step` and `Path`
- an inversion principle
- but no induction principle

2

proofs by induction

```
lemma L:
  forall l: list 'a. length l >= 0
```

can't be proved by SMT solvers *

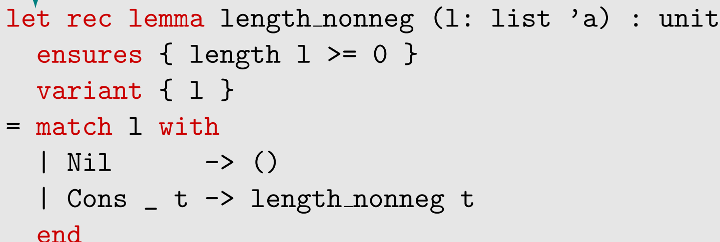
we resort to lemma functions, which are ghost program functions whose sole purpose is to build proofs
(see for instance Program Proofs [Leino, 2023])

```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
  end
```

lemma function

we resort to lemma functions, which are ghost program functions whose sole purpose is to build proofs
(see for instance Program Proofs [Leino, 2023])

program function



```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil       -> ()
  | Cons _ t -> length_nonneg t
  end
```

we resort to lemma functions, which are ghost program functions whose sole purpose is to build proofs
(see for instance Program Proofs [Leino, 2023])

program function *ghost*

```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
  end
```

we resort to lemma functions, which are ghost program functions whose sole purpose is to build proofs
(see for instance Program Proofs [Leino, 2023])

program function *ghost* *returns nothing*

```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
  end
```

lemma function

we resort to lemma functions, which are ghost program functions whose sole purpose is to build proofs
(see for instance Program Proofs [Leino, 2023])


```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
  end
```

lemma function

we resort to lemma functions, which are ghost program functions whose sole purpose is to build proofs
(see for instance Program Proofs [Leino, 2023])

program function — *ghost* — *returns nothing*

```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }      ← postcondition
  variant { l }      ← must terminate
= match l with
  | Nil       -> ()
  | Cons _ t -> length_nonneg t
  end
```

we resort to lemma functions, which are ghost program functions whose sole purpose is to build proofs
(see for instance Program Proofs [Leino, 2023])

program function *ghost* *returns nothing*

```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }        ← postcondition
  variant { l }        ← must terminate
= match l with
  | Nil       -> ()
  | Cons _ t -> length_nonneg t        ← recursive call
  end
```

1. first, the lemma function is verified, like any program function

2. then it generates a hypothesis in the context

```
axiom length_nonneg:
  forall l: list 'a. length l >= 0
```

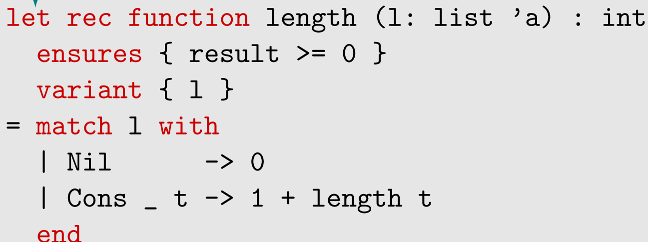
3. besides, it can be called like any ghost function

```
  ...
  let n = length mylist in
  length_nonneg mylist;
  ...
```

alternatively, we can define and prove at the same time

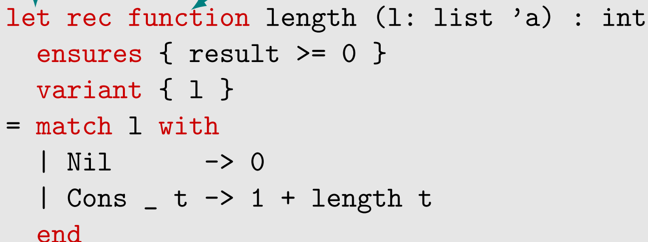
```
let rec function length (l: list 'a) : int
  ensures { result >= 0 }
  variant { l }
= match l with
  | Nil      -> 0
  | Cons _ t -> 1 + length t
  end
```

alternatively, we can define and prove at the same time




```
let rec function length (l: list 'a) : int
  ensures { result >= 0 }
  variant { l }
= match l with
  | Nil      -> 0
  | Cons _ t -> 1 + length t
  end
```

alternatively, we can define and prove at the same time




```
let rec function length (l: list 'a) : int
  ensures { result >= 0 }
  variant { l }
= match l with
  | Nil      -> 0
  | Cons _ t -> 1 + length t
  end
```

alternatively, we can define and prove at the same time

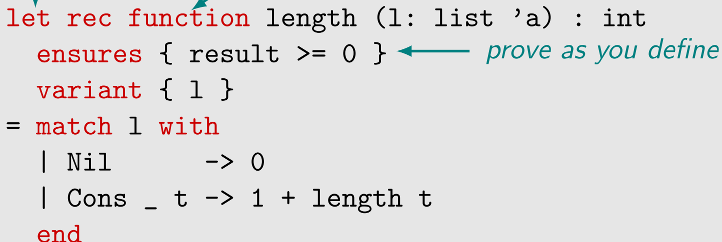
program function *and logic function*

```
let rec function length (l: list 'a) : int
  ensures { result >= 0 }
  variant { l }
= match l with
  | Nil      -> 0
  | Cons _ t -> 1 + length t
  end
```

prove as you define

alternatively, we can define and prove at the same time

program function *and logic function*




```
let rec function length (l: list 'a) : int
  ensures { result >= 0 }
  variant { l }
= match l with
  | Nil       -> 0
  | Cons _ t -> 1 + length t
  end
```

```
axiom length_def : forall l. length l = ...
axiom length_spec: forall l. length l >= 0
```

when a lemma involves an inductive predicate and requires induction, *e.g.*,

```
lemma L:
  forall x y. nim x y -> 0 <= y < x
```

we would like to prove it with a lemma function as well

3

a proposal: `match inductive`

rationale

- stick to the usual practice in Why3
  - lemma functions
  - user-guided induction via explicit recursive calls
- do not pollute the logical context
  - if possible, do not introduce any type/function/axiom

- a new construct `match inductive` to perform case analysis on inductive proofs
- a new kind of `variant` to ensure that recursive calls are legal

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z    }
  ensures    { 0 <= z < x }
= match inductive H with
  | Step _ _ _     -> ()
  | Path _ y _ _ _ -> nim_bounds x y; nim_bounds y z
  end
```

named hypothesis

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z    }
  ensures    { 0 <= z < x }
= match inductive H with
  | Step _ _ _     -> ()
  | Path _ y _ _ _ -> nim_bounds x y; nim_bounds y z
  end
```

case analysis on H

```
let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures  { 0 <= z < x }
= let ghost Step (x1 y1: int)
    requires { nim_step x1 y1 }
    requires { x = x1 /\ z = x1 }
    ensures  { 0 <= z < x } = ()
  in
  let ghost Path (x1 y z1)
    requires { nim x1 y }
    requires { nim y z1 }
    requires { x = x1 /\ z = z1 }
    ensures  { 0 <= z < x } =
    nim_bounds x y;
    nim_bounds y z
  in
  assume { false }; absurd
```

```
let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures  { 0 <= z < x }
= let ghost Step (x1 y1: int)      <------- first case
    requires { nim_step x1 y1 }
    requires { x = x1 /\ z = x1 }
    ensures  { 0 <= z < x } = ()
  in
  let ghost Path (x1 y z1)      <------- second case
    requires { nim x1 y }
    requires { nim y z1 }
    requires { x = x1 /\ z = z1 }
    ensures  { 0 <= z < x } =
    nim_bounds x y;
    nim_bounds y z
  in
  assume { false }; absurd
```

```
let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures  { 0 <= z < x }
= let ghost Step (x1 y1: int)
    requires { nim_step x1 y1 }
    requires { x = x1 /\ z = x1 }
    ensures  { 0 <= z < x } = ()
  in
  let ghost Path (x1 y z1)
    requires { nim x1 y }
    requires { nim y z1 }
    requires { x = x1 /\ z = z1 }
    ensures  { 0 <= z < x } =
    nim_bounds x y;
    nim_bounds y z
  in
  assume { false }; absurd
```

```
let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures  { 0 <= z < x }
= let ghost Step (x1 y1: int)
    requires { nim_step x1 y1 }
    requires { x = x1 /\ z = x1 }
    ensures  { 0 <= z < x } = ()
  in
  let ghost Path (x1 y z1)
    requires { nim x1 y }
    requires { nim y z1 }
    requires { x = x1 /\ z = z1 }
    ensures  { 0 <= z < x } =
    nim_bounds x y;
    nim_bounds y z
  in
  assume { false }; absurd
```

```
let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures  { 0 <= z < x }
= let ghost Step (x1 y1: int)       <------  first case
    requires { nim_step x1 y1 }
    requires { x = x1 /\ z = x1 }   <------  unification
    ensures  { 0 <= z < x } = ()    user code
  in
  let ghost Path (x1 y z1)          <------  second case
    requires { nim x1 y }
    requires { nim y z1 }
    requires { x = x1 /\ z = z1 }   <------  unification
    ensures  { 0 <= z < x } =
    nim_bounds x y;
    nim_bounds y z                  user code
  in
  assume { false }; absurd          <------  nothing else to prove
```

since postconditions for the cases are taken from the surrounding function, the `match inductive` must be in a tail position

```
match inductive H with ... end; ...
```

```
File "test.mlw", line 42, ...
match inductive should be in tail position
```

ghost code must terminate, and lemma functions make no exception

we extend Why3 with a new kind of variant

```
variant { inductive H }
```

meaning: on a recursive call, we must be able to match the parameters of smaller instances of the predicate collected so far (possibly transitively)

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z     }
  ensures    { 0 <= z < x  }
  variant    { inductive H }
= match inductive H with
  | Step _ _ _     -> ()
  | Path _ y _ _ _ -> nim_bounds x y; nim_bounds y z
  end
```

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z     }
  ensures    { 0 <= z < x  }
  variant    { inductive H }
= match inductive H with
  | Step _ _ _      -> ()
  | Path _ y _ _ _ -> nim_bounds x y; nim_bounds y z
  end
```

user-provided variant

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z     }
  ensures    { 0 <= z < x  }
  variant    { inductive H }
= match inductive H with
  | Step _ _ _       -> ()
  | Path _ y _ _ _ -> nim_bounds x y; nim_bounds y z
  end
```

user-provided variant

collected instances

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z      }
  ensures    { 0 <= z < x   }
  variant    { inductive H }
= match inductive H with
  | Step _ _ _      -> ()
  | Path _ y _ _ _ -> nim_bounds x y; nim_bounds y z
  end
```

user-provided variant

collected instances

$$x = x \wedge y = y \vee x = y \wedge y = z$$

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z      }
  ensures    { 0 <= z < x   }
  variant    { inductive H }
= match inductive H with
  | Step _ _ _       -> ()
  | Path _ y _ _ _ -> nim_bounds x y; nim_bounds y z
  end
```

user-provided variant

collected instances

$x = x \wedge y = y \vee x = y \wedge y = z$

$y = x \wedge z = y \vee y = y \wedge z = z$

as for any variant, an inductive variant can be part of a lexicographic tuple variant

e.g.,

```
variant { n, inductive H, t }
```

mutually inductive predicates

```
inductive even int = ...
with      odd  int = ...
```

induce mutually recursive lemma functions, with suitable variants,

```
let rec lemma evenP (n: int)
  requires E { even n      }
  variant    { inductive E }
  ...
with lemma oddP (n: int }
  requires O { odd n       }
  variant    { inductive O }
  ...
```

4

an application

λ-calculus

Constructive Computation Theory [Huet, 1988]

- course notes on λ-calculus
- OCaml implementation (de Bruijn indices, parsing, printing, computation strategies, Church numerals, lists, etc.)
- Prism Theorem, Standardisation, Separability
- 3 kloc

Residual theory in λ-calculus: a formal development [Huet, 1994]

- a Coq formalization of the above, up to the Prism Theorem and its corollaries
- 2 kloc

a good candidate for us, as it contains

- 2 algebraic data types
- 11 inductive predicates
- 78 proofs by induction

Coq

```
Lemma red_abs:
  forall M M': lambda,
  red M M' → red (Abs M) (Abs M').
Proof.
  simple induction 1; intros.
  - apply step; apply abs; trivial.
  - apply refl.
  - apply trans with (Abs N); trivial.
Qed.
```

Why3

```
let rec lemma red_abs (m m': lambda)
  requires r { red m m' }
  ensures    { red (Abs m) (Abs m') }
  variant    { inductive r }
= match inductive r with
  | step  _ _ _       -> ()
  | refl  _           -> ()
  | trans m n m' _ _ -> red_abs m n; red_abs n m'
  end
```

the why3 proof is roughly two times smaller than the Coq proof

Coq	Why3
81 kb	42 kb
78 `induction`	72 `match inductive`

SMT solvers are useful, even for such kind of proofs!

note: Why3 has no `nat` type and suitable preconditions were added at various places

Why3's ghost code is now extended with

- a new construct `match inductive` to perform case analysis on inductive proofs
- a new kind of `variant` to ensure that recursive calls are legal

future work: support infinite branches (*e.g.* accessibility predicate)