Proofs on inductive predicates in the WhyML programming language

Henri Saudubray*

July 5, 2024

Abstract

The Why3 environment provides so called "inductive predicates" as a means to describe inductive properties. However, there is no way to analyse the structure of instances of these predicates directly in WhyML code. This case analysis is crucial in proofs by induction on these properties, and its absence from the WhyML language prevents these proofs from being automatically discharged by automatic theorem provers. We thus present a new language construct for case analysis on inductive predicate instances in WhyML, and an example use case in the translation of a Coq proof to WhyML.

1 Introduction

The Why3 environment is a tool for deductive program verification [4] based on a first-order logic with ML-style polymorphic types [2], which provides a common interface and specification language for interacting with multiple automated provers and proof assistants in order to discharge proof obligations. Why3 provides an accompanying programming language, WhyML, an ML dialect restricted to first order to be able to generate first-order proof obligations. Why3 also provides the user with multiple extensions to its core first-order logic: recursive definitions, algebraic data types, and inductive predicates [3]. The latter allow the user to express inductive properties on data through a set of constructors with access to recursion.

While automated provers cannot usually reason by induction, automatic proofs by induction on values is still possible in Why3 by representing proofs as recursive functions validating a specification, which can be seen as proof objects through the Curry-Howard isomorphism, and then proving their termination, typically through a base case and a decreasing argument. Unfortunately, case analysis on inductive predicates is not possible, preventing automatic proofs by induction on the instances of inductive predicate. We introduce a new construct to the WhyML programming language allowing for case analysis on inductive predicate instances, and allowing for automatic proofs by induction on these predicates.

All of the code and proof associated with this report is freely available online at `https://codeberg.org/17maiga/lambda`.

*Université Paris Saclay, henri.saudubray@universite-paris-saclay.fr

2 Context

2.1 WhyML: Programming and Specification Language

WhyML code lives in two separate worlds: logic and programs. Logic is used to describe program specifications and logical properties on programs, and is not included when translating WhyML code to OCaml or C. Programs are the actual executable code which we wish to reason about. Logic allows for a variety of constructs that are not available in programs, such as universal and existential quantifiers or inductive definitions. All logical definitions require proof of termination, and Why3 will attempt to find a decreasing argument for each call. If such an argument cannot be found, the user must then manually provide a variant.

2.1.1 Inductive Predicates

WhyML includes inductive predicates, which allow the user to describe inductive properties through a set of constructors with access to recursion. Similarly to Coq's inductive definitions [8], calls to these predicates, which we call instances, are theoretically associated with a proof object representing the structure of the instance. This object is constructed through a finite number of applications of the constructors associated with the inductive predicate. However, this object does not currently exist as a tangible value in Why3.

As an example of an inductive predicate, we can define the reflexive transitive closure `red` of β-reduction in the λ-calculus as an inductive predicate, with three constructors for the possible cases of this reduction, as follows:

```
inductive red term term =
  | refl:  forall x.     red  x x
  | step:  forall x y.   beta x y → red x y
  | trans: forall x y z. red  x y → red y z → red x z
```

Inductive predicates allow for the description of recursive properties based on case analysis on the structure of these properties. They are similar to recursive predicates, and a number of properties can be expressed using both, with advantages and disadvantages in either case, as seen in the example below.

```
let rec predicate even (n: int)
  variant { n }
= n = 0 || n > 1 && even (n-2)
```

```
inductive even int =
  | O: even 0
  | S: forall n. even n → even (n+2)
```

One particular advantage of inductive definition instances is that they are finite by construction. Since any proof of an inductive property is associated with a finite proof tree obtained through applications of the associated constructors, proofs by induction on an inductive predicate instance can immediately use this proof tree as a decreasing argument to prove their termination. Recursive predicates, on the other hand, need to manually specify a variant in order to prove termination, and prove that said variant is indeed decreasing for each recursive call.

Inductive definitions also have a non-deterministic nature. If we look at the example for `red` above, we can build multiple different proof trees for the same property, and some constructors can introduce variables not initially present in the

context.

$$\textsc{step}\ \frac{x \to_\beta z \qquad \textsc{refl}\ \dfrac{z = y}{z \to_\beta^* y}}{x \to_\beta^* y}$$

$$\textsc{trans}\ \frac{\textsc{refl}\ \dfrac{x = z}{x \to_\beta^* z} \qquad \textsc{step}\ \dfrac{z \to_\beta w \qquad \textsc{refl}\ \dfrac{w = y}{w \to_\beta^* y}}{z \to_\beta^* y}}{x \to_\beta^* y}$$

This ability to introduce variables will become particularly important in proofs by induction on inductive predicate instances, as they come with additional properties based on the constructor; in the example above, the variables w and z are introduced along with the properties defined in the constructors that introduce them.

In a system like Coq, an inductive definition like `even` above is automatically associated with a higher-order induction principle:

```
even_ind : forall P : nat → Prop.
             P 0 →
             (forall n. even n → P n → P (n+2)) →
             forall n. even n → P n.
```

When the `induction` tactic is used, this induction principle is then instantiated depending on the current goal, and the resulting goals correspond to the different constructors of the inductive definition.

2.1.2 Lemma Functions

Typical lemmas in WhyML are strictly logical and simply introduce a new premise in the logical context (and require the user to prove it) for further use. One can specify an implementation for the lemmas by turning it into a lemma function. These introduce both a logical premise based on the function's contract, with universally quantified variables to represent the arguments, and a ghost function usable in programs. The following definitions are therefore logically equivalent.

```
lemma even_nonneg:
  forall x: int.
    even x → x ≥ 0
```

```
let lemma even_nonneg (x: int)
  requires { even x }
  ensures { x ≥ 0 } = ()
```

In our case, lemma functions present two interesting advantages: function bodies and recursion. Since lemma functions are also programs, they require a function body, allowing the user to direct the lemma's proof through constructs like case analysis and conditions, and making the task easier for SMT solvers afterwards. A standard lemma does not permit this flexibility, instead relying on the context and interactive proof through manual application of logical transformations on the goal by the user. This is typically done in Why3's graphical interface. The nature of the lemma function as a program also permits recursive calls to itself as part of the proof, essentially allowing the user to perform proofs by induction on the arguments of the lemma function. Proof by induction on values is therefore possible, as seen in the example below.

```
let rec lemma len_nonneg (l: list α)
  ensures { length l ≥ 0 }
  variant { l }
= match l with
  | Nil      → ()
  | Cons _ r → len_nonneg r
  end
```

Similarly to this, one could see how we would want to analyse the structure of an inductive predicate instance in order to do proofs by induction on it. As an example, suppose we introduce `red` a little differently: we remove the transitivity constructor `trans`, and instead add another precondition to the `step` constructor. We then want to prove the transitivity of `red` manually:

```
inductive red term term =
  | refl: forall x.     red  x x
  | step: forall x y z. beta x y → red y z → red x z

lemma red_trans:
  forall x y z. red x y → red y z → red x z
```

To prove `red_trans`, we could do a simple proof by induction on the derivation of `red x y`. In the case `refl`, we have that `x = y`, and we can then conclude `red x z`. In the case `step`, there exists an element `w` such that `beta x w` and `red w y`. By our induction hypothesis, we have that `red w z`, and thus we can conclude `red x z`, which concludes the proof.

This reasoning cannot be done by automated solvers on their own, and thus the proof of `red_trans` is not automated; it requires manual intervention from the user, typically by applying the `induction_pr` tactic in Why3's interactive interface. Reasoning using the variable `w` is also not currently possible in WhyML, as we have no way to decompose an instance of an inductive predicate.

3 A New Construct for Case Analysis

To solve this, we introduce a new construct to the WhyML language, which we call `match inductive`. Similarly to the existing `match` construct for case analysis on algebraic data types, this new construct allows for case analysis on instances of inductive predicates. We want a way to associate the structure of the inductive predicate instance with a name, which we can then use as the value we perform our case analysis on. Syntactically, we use the existing annotation system to associate a name to a lemma's precondition. This precondition must be a single call to an inductive predicate, and the name is then associated with this call's instance. We can then use this name as the target of our case analysis.

```
let rec lemma red_trans (x y z: term)
  requires rxy { red x y }
  requires { red y z }
  ensures { red x z }
= match inductive rxy with
  | (* ... *)
  end
```

However, we still need a way to represent internally the term associated with this proof. In Coq, this problem is solved by the fact that every proof is defined by a tangible term which we can directly manipulate. We do not have access to such a term in Why3, and need to provide either a tangible term or simulate it using existing elements.

3.1 A First Approach: Proof Witnesses

The immediate approach is to create a witness type representing an instance of an inductive predicate, and internally translate proofs without this witness, written by the user, to proofs with this witness:

```
type red'witness =
  | refl'witness term
  | step'witness term term term red'witness
```

We can then generate an equivalent recursive predicate using this witness

```
predicate red'witness (x y: term) (w: red'witness) =
  match w with
  | refl'witness x0          → x0 = x ∧ x0 = y
  | step'witness x0 y0 z w0 → x0 = x ∧ z = y ∧ beta x0 y0 ∧
      red'witness y0 z w0
  end
```

and utilities establishing the equivalence relationship between a predicate and its witness, that is, a way to obtain a witness for an instance of an inductive predicate, and a proof that the recursive predicate with witnesses defined above implies the original inductive predicate:

```
val ghost function red'get_witness (x y: term) : (w: red'witness)
  requires { red x y }
  ensures { red'witness x y w }

let rec lemma red'witness_to_red (x y: term) (w: red'witness)
  requires { red'witness x y w }
  ensures { red x y }
  variant { w }
= match w with
  | refl'witness _           → ()
  | step'witness x0 y0 z w0 → red'witness_to_red y0 z w0
  end
```

We can then transform the `match inductive` expression into a standard `match` on the structure of the associated witness, obtained through a call to one of the generated utility functions.

```
let rec lemma red_trans (x y z: term)
  requires rxy { red x y }
  requires { red y z }
  ensures { red x z }
= match red'get_witness x y with
  | refl'witness _         → ()
```

```
| step'witness _ y0 _ _ → red_trans y0 y z
end
```

While this method works, it is quite cumbersome and verbose, and introduces several auxiliary definitions and premises in the logical context which negatively impacts both the provers and the user. Since some provers do not support algebraic data types, Why3 applies a series of logical transformations (most notably `eliminate_algebraic`) which introduce a number of definitions simulating algebraic data types in the prover's logic. These definitions would needlessly pollute the logical context, resulting in poorer performance compared to a solution without such constructs.

The proof of termination is also not immediate. If each recursive call uses one of the utility lemmas generated by the definition of our inductive predicate to obtain a witness, nothing ensures that the witness obtained in recursive calls (which can theoretically be considered as an argument of the lemma) is a smaller term than the witness we are currently doing case analysis on. One solution to this is to transform the entire lemma to take in an additional witness argument and pass in the witnesses obtained through our case analysis to the recursive calls. While this could work, how do we handle the case where Why3 cannot infer a decreasing argument automatically? Witnesses are hidden from the user, and thus we cannot expect the user to provide a decreasing variant using them explicitly.

3.2 A Better Approach: Ghost Functions

We can in fact find a simpler solution. The induction principle associated with our inductive predicate definition tells us that it suffices to prove a property for each constructor of our predicate in order to prove it for every possible instance of our predicate. We can therefore handle each possible constructor separately and assume our postcondition once every constructor has proved it. Each constructor must be able to conclude the postcondition independently from all other constructors in order for the proof to be valid.

This goes against our previous intuition of using the witness as an explicit decreasing argument in order to ensure termination. In fact, the issue of being able to deduce that the instance we are doing case analysis on in recursive calls is in fact a subterm of the original instance is still present. Even worse, there is no such instance anymore, only the properties provided by the lemma's preconditions! What is stopping us from reaching the conclusion using the wrong constructors?

However, we can convince ourselves that this approach works. Since each constructor is able to prove the postcondition independently from all the others (and only this postcondition), we know that even if we are able to prove our postcondition using the wrong constructor, we could always prove it using the right one. We cannot then express anything more than we would have been able to with explicit witnesses, since all constructors only allow us to conclude exactly the postcondition, and only weaken our expressivity through their added prerequisites. Thus, what we can prove without a witness, we can prove with one.

The structure used to represent a constructor must then be able to introduce variables, specify preconditions and postconditions in order to represent the variables and properties defined in the constructor as well as ensure that the constructor does prove the postcondition, while not introducing any property in the logical context it is defined in. These requirements directly match one existing structure: a

ghost function. Why3 will require a proof of its contract, but the ghost function will not introduce any premises in the logical context and will not be usable in programs, thus ensuring that all constructors are proved separately.

We can then generate a set of local ghost functions (one for each constructor defined in the inductive predicate), with arguments for the constructor's universally quantified variables, preconditions introducing the properties specified in the constructor, and identical postconditions to those of the parent scope. Since we are doing case analysis on a specific instance of an inductive predicate, we also introduce preconditions for the unification of the instance's arguments and the ghost function's arguments. We can see below what a fragment of the body of `red_trans` might look like, with its associated translation inside Why3. We analyze the precondition `rxy { red x y }` and handle the `refl` constructor, where `x` and `y` are both equal to a term `a`.

```
match inductive rxy with
| refl a → ()
| (* ... *)
end
```

```
let ghost refl (a: term)
  requires { a = x ∧ a = y }
  ensures { red x z }
= () in (* ... *)
```

In order to obtain the arguments and preconditions of the ghost function, we extract from the constructor's definition all universally quantified variables and hypotheses, as well as the final call to the inductive predicate. We can then generate local variables for each universally quantified variable and substitute them in the hypotheses. We also introduce equalities between the arguments of the inductive predicate instance we are performing case analysis on and their corresponding expressions in the constructor.

```
let rec lemma even_nonneg n
  requires e { even n }
  ensures { n ≥ 0 }
  variant { inductive e }
= match inductive e with
  | O      → ()
  | S n' _ → even_nonneg n'
  end
```

```
let rec lemma even_nonneg n
  requires e { even n }
  ensures { n ≥ 0 }
  variant { inductive e }
= let ghost O ()
    requires { n = 0 }
    ensures { n ≥ 0 }
  = () in
  let ghost S (n': int)
    requires { even n' }
    requires { n = n' + 2 }
    ensures { n ≥ 0 }
  = even_nonneg n' in
  assume { false }; absurd
```

A consequence of this method is that since each constructor must prove the postcondition of the lemma independently, `match inductive` must be in tail position, that is, it should be the last expression evaluated during execution of the lemma's body. By the induction principle, once each constructor has independently proved our postcondition, we can safely conclude our postcondition for any instance of our inductive predicate, and our lemma is proved. However, since the ghost functions do not introduce anything in the logical context, Why3 still needs to prove

the postcondition (which we now know to be true) once all ghost functions have been proven. Since we know this postcondition is true, we can simplify the work of the provers by making the conclusion trivial: we introduce a local assumption of `false`, which allows the provers to trivially conclude the postcondition. If the parent lemma is expected to return a value, we then follow this assumption by `absurd`, which can assume any type, and therefore matches the return type of the parent lemma. This is once again valid, since all ghost functions must return a value of said type in order for the postconditions to be valid, and `match inductive` is in tail position, ensuring that the value returned by it is the return value of the parent lemma.

Type inference for variables declared in match inductive patterns is possible, and is done through unification of the types obtained by instanciating the associated constructor in the inductive predicate's definition with the types of the predicate instance's arguments as well as in the lemma's requirements.

As for termination, we require the user to specify an inductive predicate instance through its associated name, and simply check for each recursive call that the property associated with this variant is obtained through case analysis of the same property in the parent recursive call. This inductive variant can be provided as part of a list of terms treated in lexicographic order. Implementation of this feature is still ongoing.

In a way, what we do here is almost identical to what the `induction` tactic does in Coq. When the `even_ind` induction principle is instantiated, the resulting goals are the proofs for each constructor of the postcondition `P`, with the induction hypothesis added to the logical context when relevant. Here, `P` is represented by our postcondition, which we generate a proof of for each constructor. The only difference is that we choose to remain faithful to Why3's usual way to do proofs by induction, that is, let the user obtain the induction hypothesis manually through a recursive call to the lemma.

4 Application: Lambda-Calculus in WhyML

As a way to test our new construct, we propose a direct translation of Gérard Huet's λ-calculus formalisation in Coq to WhyML using `match inductive`. This formalisation originates from an implementation of a variety of λ-calculus results in OCaml, as part of a lecture on computability [6]. It was followed by a partial reimplementation and proof in Coq, which we base our translation on. As this proof heavily relies on induction on Coq's inductive predicates, it is thus a typical example of a proof one would want to do with `match inductive` if translated to WhyML. We attempt to faithfully reproduce the original proof, with as few differences as possible.

4.1 A Lambda-Calculus Reminder

The λ-calculus [1] is a theory of computation based on functions and their applications, originally conceived by Alonzo Church in 1932 as part of his research around a foundational system for mathematics. Terms in the λ-calculus are inductively defined by the following grammar:

Computation is represented through the notion of transformations on terms

$$
\begin{array}{rcll}
\mathrm{T} & ::= & x & \text{Variable representing a parameter} \\
& | & \lambda x.\mathrm{T} & \text{Abstraction with parameter } x \\
& | & (\mathrm{T}\ \mathrm{T}) & \text{Application}
\end{array}
$$

called reductions, most notably the β-reduction:

$$(\lambda x.M\ N) \to_\beta M[x/N],$$

where $M[x/N]$ denotes the term M in which all free occurrences of variable x are instantiated with N. Multiple methods can be used to solve naming conflicts during substitution, such as α-equivalence or de Bruijn indices, in which variables are represented by natural numbers whose value refers unambiguously to the abstraction they are bound to:

$$\lambda x.(x\ x) \equiv \lambda.(0\ 0) \qquad\qquad \lambda x.\lambda y.(x\ y) \equiv \lambda.\lambda.(1\ 0)$$

Identifying free and bound variables is then immediate. Instantiation of a variable by a term then requires updating references in the substituted term depending on the location where the term is substituted.

$$\lambda z.\lambda y.((\lambda x.(x\ y))\ z) \to_\beta \lambda z.\lambda y.(z\ y)$$
$$\lambda.\lambda.((\lambda.(0\ 1))\ 1) \to_\beta \lambda.\lambda.(1\ 0)$$

4.2 Proven Results

The Coq formalisation represents λ-calculus terms with de Bruijn indices and implements β-reduction through an `Inductive` datatype and a `Fixpoint` definition. Parallel β-reduction, which consists in the simultaneous application of a single step of β-reduction on one or more redexes in a term, is defined. The reflexive transitive closures of β-reduction and its parallel version are implemented and proven to be equivalent.

A second representation of terms with marked redexes is also provided. These terms are identical to the first representation, except for an additional Boolean flag on applications indicating marked redexes. Terms in this representation then represent sets of such redexes. A predicate for the regularity of such terms (that is, whether marked applications truly represent a redex) is provided, as well as whether two terms are compatible (equal when removing marks). The union of two different sets of redexes on a same term is also provided.

Once again, substitution of terms is defined, this time taking into account the marked redexes, and a proof the preservation of regularity and compatibility through substitution of terms is made.

The notion of residuals of redexes, which studies the evolution of a redex through the reduction of another redex in the term, is defined through parallel β-reduction over terms with marked redexes, and relationship with union, regularity, and compatibility of these terms is established. This concept sees terms with marked redexes as a way to denote parallel β-reduction steps (all marked redexes are reduced in a single step). The residuals of a set of redexes through a step of parallel β-reduction is defined. The equivalence of residuals and the first definition of parallel β-reduction on terms is established, and the Prism Theorem and Lévy's Cube Lemma are then proven as seen in Gérard Huet's original article on the matter [5].

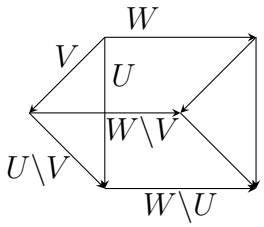

Figure 1: The Prism Theorem.

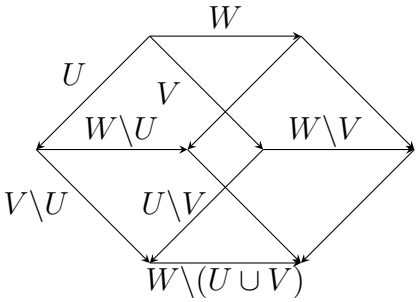

Figure 2: Lévy's Cube Lemma.

The Prism Theorem states that for a compatible set of redexes U, V, and W,

$$V \subset U \implies W\backslash U = (W\backslash V)\backslash(B\backslash U),$$

where $A\backslash B$ denotes the residuals of A through the step of parallel β-reduction B. Lévy's Cube Lemma, which states that for every compatible set of redexes U, V, and W,

$$(W\backslash V)\backslash(U\backslash V) = (W\backslash U)\backslash(V\backslash U),$$

can then be seen as a corrollary of this.

The confluence of multi-step β-reduction is proven using the Tait-Martin-Löf method, that is, through the parallel-moves lemma, which states that if a term M reduces to terms N and P through one step of parallel β-reduction, then N and P both reduce to a common term Q through one more step. The confluence of multi-step parallel β-reduction and multi-step β-reduction are then corollaries of this lemma. Finally, the Church-Rosser theorem is proved with β-conversion.

The entire proof in Coq represents 1263 lines of code, with 511 for the specification and 768 for the proof, as reported by the `coqwc` tool. More interesting in our case, the proof contains 42 occurrences of the `induction` tactic applied to an instance of an inductive predicate, more than half of all applications of this tactic.

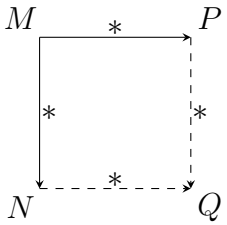

Figure 3: The Church-Rosser Theorem.

4.3 Translation to WhyML

The translation to WhyML is mostly a direct translation of the Coq version with WhyML constructs. Inductive `Set` definitions are directly represented by algebraic data types, and inductive `Prop` definitions by inductive predicates, as seen below:

```
(* Coq *)
Inductive lambda : Set :=
  | Ref : nat → lambda
  | Abs : lambda → lambda
  | App : lambda → lambda → lambda.

Inductive red : lambda → lambda → Prop :=
  | step:  forall M N,   beta M N → red M N
  | refl:  forall M,     red  M M
  | trans: forall M N P, red  M N → red N P → red M P.

(* WhyML *)
type lambda =
  | Ref int
  | Abs lambda
  | App lambda lambda

inductive red lambda lambda =
  | step:  forall m n.   beta m n → red m n
  | refl:  forall m.     red  m m
  | trans: forall m n p. red  m n → red n p → red m p
```

Properties whose proof does not rely on induction are proven by automatic provers with little to no need for manual intervention, and are thus only stated as lemmas in the WhyML source code. When a proof by induction is required, however, we instead state the proof as a recursive lemma-function with the appropriate specification. If the induction is done on a value (in Coq, an inductive `Set` definition), we simply do case analysis with `match` and perform recursive calls when needed. If the induction is done on an inductive `Prop` definition, we associate a name to this property either through the lemma's preconditions or through one of the binders in a parent `match inductive` branch, and perform case analysis with `match inductive`.

```
(* 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.

(* WhyML *)
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
```

One notable difference between the two proofs is due to the absence of a data type for natural numbers in WhyML, which only has integers; some results therefore assume that some integers are non-negative as an added hypothesis. Another, perhaps more striking difference is the way the proof of confluence has been handled. In the Coq version, Gérard Huet first defines the notion of confluence with both datatype and relation as parameters, and then instantiates this with the `lambda` datatype representing λ-terms and multiple different reduction relations.

```
Definition confluence (A : Set) (R : A → A → Prop) :=
  forall x y : A, R x y →
  forall z   : A, R x z →
  exists u   : A, R y u ∧ R z u.

Lemma lemma1 : confluence lambda par_red → confluence lambda red.
```

We cannot express this definition in WhyML without relying on Why3's module and cloning system, since we are in a first-order logic, as we cannot directly abstract over a type in a definition as in Coq. However, we can abstract over a relation like `R` above, thanks to Why3's extensions and encodings to first-order logic. Fortunately for us, the proof in Coq does not use its definition of confluence with any other data type, and so we can define the confluence in Why3 specifically for λ-terms, with the relation as a parameter, and proceed from there. We do not end up having to define it more than once, but the limitations of Why3's expressivity compared to Coq are made clear in this example.

Aftermath. The final proof in WhyML stands at 914 lines of code, with 521 for the specification and 393 for the proof, just over 50% of the Coq version. The overwhelming majority of the code for the proof is simply composed of uses of `match inductive` to handle the case analysis of the predicates, and recursive calls which provide the induction hypotheses necessary for the conclusion of the final postcondition. All proof obligations are discharged by automatic theorem provers

with little to no manual intervention, with the exception of the application of the `split_vc` tactic at times.

5 Related Work

Describing properties through inductive definitions is done in systems like Coq, Isabelle/HOL, or Agda, which all provide a way to reason on properties in a direct manner. Implementations vary; Coq directly provides a proof term with the property as type and allows the user to manipulate this term directly, thus allowing for structural case analysis and recursion; both Coq and Isabelle/HOL automatically generate an induction rule with each inductive definition to be used in tactic applications.

Systems like Dafny [7] do not have an equivalent construct to inductive predicates, and instead express such properties through recursive functions over regular data types. Proofs by induction on these properties are then done through recursive lemmas, and a structurally decreasing argument must be provided as a variant for each recursive call.

6 Conclusion and Perspectives

We provide a possible encoding of case analysis on inductive properties in Why3's first-order logic, relying on the induction principle as a metatheoretical argument for the soundness of our approach. However, it is not the only way to proceed, as we have seen from our first attempts with explicit witnesses, and further work exploring how these witnesses might be represented in systems such as Why3 without resorting to a full representation of proofs as tangible terms *à la* Coq could provide alternatives.

However, despite its limitations, this method already shows its usefullness, as seen in the translation of Gérard Huet's work. A large number of proofs relying on inductive properties, such as formalisations of type systems and program semantics, can now be expressed in Why3 with little to no difficulty, giving access to all of Why3's advantages as a platform. Such work would be an excellent way to test the limitations of `match inductive`, and open up new research paths and ideas.

Aknowledgements. I wish to thank both Jean-Christophe Filliâtre[1] and Andrei Paskevich[2], the supervisors of the internship which resulted in this work, for their crucial insight and support.

References

[1] Henk P Barendregt. Lambda calculi with types. *Handbook of Logic in Computer Science*, 2:118–413, 1993.

[2] Joshua M Cohen and Philip Johnson-Freyd. A formalization of Core Why3 in Coq. *Proceedings of the ACM on Programming Languages*, 8(POPL):1789–1818, 2024.

[1]Laboratoire Méthodes Formelles, jean-christophe.filliatre@cnrs.fr
[2]Laboratoire Méthodes Formelles, andrei.paskevich@universite-paris-saclay.fr

[3] Jean-Christophe Filliâtre. One logic to use them all. In *International Conference on Automated Deduction*, pages 1–20. Springer, 2013.

[4] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*, pages 125–128. Springer, 2013.

[5] Gérard Huet. Residual theory in λ-calculus: a formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[6] Gérard Huet. Constructive computation theory. *Course notes on lambda calculus, University of Bordeaux I*, 2011.

[7] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[8] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types*. PhD thesis, Université Claude Bernard-Lyon I, 1996.