Faire chauffer Why3 avec de l'induction*

Henri Saudubray[1], Jean-Christophe Filliâtre[1,2] et Andrei Paskevich[1,2]

[1]LMF — Université Paris-Saclay, CNRS, ENS Paris-Saclay
[2]Toccata — Inria Saclay

Cet article présente une extension de l'outil Why3 pour permettre des preuves par récurrence sur des instances de prédicats inductifs, c'est-à-dire sur des dérivations finiment construites. Elle se compose d'une nouvelle construction de filtrage pour analyser la forme d'une telle dérivation, d'une part, et d'une nouvelle notion de variant pour justifier la terminaison d'une fonction récursive qui procède selon la taille d'une dérivation, d'autre part. Nous montrons comment cette extension peut être implémentée de façon conservative, sans presque rien modifier dans le générateur de conditions de vérification de Why3. Enfin, nous illustrons la solidité de cette contribution en traduisant de Coq vers Why3 une preuve non triviale contenant un grand nombre de raisonnements par récurrence sur des prédicats inductifs.

1 Introduction

Why3 est un environnement pour la vérification déductive de programmes [FP13]. Il fournit un langage de programmation, appelé WhyML, et un langage logique avec lequel on va pouvoir construire des théories et annoter les programmes. La vérification proprement dite passe alors par une première étape de génération de conditions de vérification, via un calcul classique de plus faibles préconditions, puis par la traduction de ces énoncés vers les langages d'entrée d'un grand nombre de démonstrateurs pris sur l'étagère [BFMP11], notamment des démonstrateurs de type SMT comme Alt-Ergo [CCIM18], Z3 [dMB] ou encore CVC5 [BBB+22]. En particulier, WhyML peut être utilisé comme un langage intermédiaire dans un outil de vérification déductive, à l'instar de Boogie [BDJ+05] ou de Viper [MSS16]. Why3 est ainsi utilisé pour vérifier du code C, Rust, Ada ou encore OCaml.

La logique de Why3 est une extension de la logique du premier ordre avec notamment des types algébriques, des définitions récursives et des prédicats inductifs [Fil13, CJF24]. Lorsqu'une preuve par récurrence est requise, que ce soit pour montrer la validité d'un lemme ou pendant la preuve d'un programme, elle ne pourra pas être faite par un démonstrateur automatique de type SMT. Cette limitation est inhérente au fait que le principe d'induction derrière un type algébrique récursif ou un prédicat inductif ne peut être exprimé dans la logique du premier ordre d'un démonstrateur SMT. Dit autrement, Why3 ne parvient à transmettre son langage logique que de façon incomplète aux démonstrateurs automatiques. Il revient donc à l'utilisateur de Why3 de construire sa preuve par récurrence en amont de

*Cet article présente le travail de stage de M1 d'Henri Saudubray, entre avril et juillet 2024, encadré par Jean-Christophe Filliâtre et Andrei Paskevich. Ce travail a été financé par le projet ANR "GOSPEL" et par le projet Décysif soutenu par la région Île-de-France et par le gouvernement français dans le cadre du "Plan France 2030".

ces derniers. Quand il s'agit d'une récursion sur un entier naturel, une liste ou encore un arbre, la pratique courante avec Why3 consiste à présenter cette preuve sous la forme d'un programme récursif écrit en WhyML. Or, il n'est actuellement pas possible de construire un tel « programme-preuve » pour une récurrence sur une instance d'un prédicat inductif. Dans cet article, nous levons cette limitation en apportant à Why3 une double extension, à savoir :

— une nouvelle construction de filtrage sur une instance inductive ;
— une nouvelle sorte de variant, associée à une instance inductive.

Il devient alors possible d'écrire dans le langage WhyML une fonction récursive bien fondée qui procède par récursion sur une instance inductive dans le but de construire la preuve d'une propriété.

Le reste de cet article est organisé de la façon suivante. La section 2 présente la logique de Why3 (2.1), sa traduction vers les démonstrateurs SMT (2.2) et la façon traditionnelle de réaliser des preuves par récurrence dans ce contexte (2.3). La section 3 présente alors notre contribution, puis la section 4 l'applique à une preuve de grande ampleur. Enfin, la section 5 compare notre contribution à d'autres approches.

L'extension à Why3 présentée dans cet article est disponible dans le dépôt de Why3, dans la branche `https://gitlab.inria.fr/why3/why3/-/tree/match_inductive` en attente de son intégration définitive. La preuve présentée dans la section 4 se trouve également dans cette branche, dans le sous-répertoire `examples/lambda`.

2 Contexte

Cette section présente la logique de Why3, sa traduction vers les démonstrateurs automatiques et la façon dont sont faites les preuves par récurrence.

2.1 La logique de Why3

La logique de Why3 est une logique de *compromis*. Elle se veut à la fois suffisamment expressive, afin que les utilisateurs soient dans le confort pour définir leurs théories logiques et spécifier leurs programmes, et suffisamment simple, afin que sa traduction vers les démonstrateurs automatiques soit la plus directe et la plus efficace possible. Il y a là une tension, bien évidemment, qui a amené les développeurs de Why3 à choisir une logique du premier ordre sur laquelle viennent se greffer un certain nombre d'extensions : polymorphisme paramétrique à la Hindley-Milner, définitions récursives, types algébriques et filtrage, prédicats inductifs [Fil13, CJF24].

Nous donnons ici quelques exemples de ces diverses extensions à la logique du premier ordre. On déclare ainsi un type `list` pour des listes polymorphes avec deux constructeurs `Nil` et `Cons`.

```
type list α = Nil | Cons α (list α)
```

C'est en tout point semblable à ce que l'on ferait dans un langage comme OCaml, à la syntaxe près. On peut ensuite utiliser les deux constructeurs de ce type pour introduire une valeur particulière, ici une liste de trois entiers :

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

Pour définir une fonction sur les listes, comme par exemple le calcul de la longueur, on utilise un mélange de récursivité et de filtrage, là encore à l'instar de ce que l'on ferait dans un langage fonctionnel :

```
function length (l: list α) : int
= match l with
  | Nil      → 0
  | Cons _ t → 1 + length t
  end
```

Le mot clé `function` introduit une fonction logique. Celle-ci peut être récursive, comme c'est le cas ici, et Why3 s'assure alors de sa terminaison. La logique de Why3 est en effet totale : les définitions récursives doivent être bien fondées. Dans l'exemple ci-dessus, la terminaison est assurée par la décroissance structurelle de l'argument, ce que Why3 a vérifié par lui-même. Dans des cas plus complexes, un variant peut être proposé par l'utilisateur, et des conditions de vérification expriment alors sa décroissance. Une fois la fonction `length` introduite, on peut l'utiliser dans tout contexte où un terme logique apparaît, comme par exemple le but suivant :

```
lemma L1: length alist = 3
```

Il est trivialement déchargé par tout démonstrateur automatique supporté par Why3, par exemple Alt-Ergo :

```
$ why3 prove -P alt-ergo file.mlw
File "file.mlw", line 14, characters 9-25:
Goal L1.
Prover result is: Valid (0.01s, 15 steps).
```

En revanche, si on énonce un résultat un peu plus ambitieux, comme le fait que la longueur d'une liste est un entier positif ou nul,

```
lemma L2: forall l: list α. length l ≥ 0
```

alors le but ne sera plus démontré automatiquement :

```
$ why3 prove -P alt-ergo file.mlw
File "file.mlw", line 16, characters 11-43:
Goal L2.
Prover result is: Timeout (30s, 30975 steps).
```

Cela peut être surprenant pour un résultat aussi simple, mais il faut comprendre que les démonstrateurs automatiques ne sont pas en mesure de réaliser une preuve par récurrence, ce qui est nécessaire ici, pas plus que Why3 ne le fait à leur place. La section 2.2 expliquera pourquoi et la section 2.3 apportera une solution.

La logique de Why3 distingue les termes et les formules, c'est-à-dire qu'elle n'identifie pas les formules avec les termes booléens. C'est un choix [1]. Dès lors, la commande `function` pour déclarer une fonction vient de paire avec une commande `predicate` pour déclarer un prédicat. Voici un exemple simple de prédicat non récursif :

```
predicate nim_step (x y: int) =
  (x = y+1 ∨ x = y+2) ∧ y ≥ 0
```

Comme dans le cas d'une fonction, une définition récursive d'un prédicat est également possible, pourvue qu'elle soit bien fondée.

Enfin, et c'est là le sujet de cet article, la logique de Why3 permet également de définir un prédicat de manière *inductive*, par un ensemble de clauses que l'on peut voir comme autant d'axiomes et de règles d'inférence. Ainsi, le prédicat `nim_step` ci-dessus peut-il être défini plutôt de la manière suivante :

```
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
```

Ici, on a deux clauses `RemoveOne` et `RemoveTwo` qui peuvent être vues comme deux axiomes définissant le prédicat `nim_step`. Plus précisément, le prédicat `nim_step` est *le plus petit prédicat* satisfaisant ces deux axiomes. En particulier, on peut énoncer le résultat suivant

1. Pour autant, Why3 offre des coercions implicites entre formules et booléens qui peuvent donner l'illusion que les deux sont la même chose.

```
lemma L3: forall x y. nim_step x y → 0 ≤ y < x
```

et il est même à la portée des démonstrateurs SMT.

Les prémisses des clauses définissant un prédicat inductif peuvent faire référence à ce même prédicat, pourvu que ce soit dans des positions positives. Voici un tel exemple, avec la clôture transitive du prédicat `nim_step` :

```
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
```

On a ici une clause `Step` qui s'apparente à un axiome et une clause `Path` qui s'apparente à une règle d'inférence. Une instance du prédicat `nim`, comme par exemple `nim 42 39`, peut alors être vue comme une application *finie* — on parle d'arbre de dérivation — de la règle `Path` et de l'axiome `Step`. Voici une telle dérivation (mais ce n'est pas la seule) :

$$\textsc{Path}\ \dfrac{\textsc{Step}\ \dfrac{\texttt{nim\_step 42 41}}{\texttt{nim 42 41}} \qquad \textsc{Step}\ \dfrac{\texttt{nim\_step 41 39}}{\texttt{nim 41 39}}}{\texttt{nim 42 39}}$$

Comme on l'a fait plus haut avec `nim_step`, on peut énoncer un résultat sur le prédicat `nim` qui capture sa nature inductive :

```
lemma L4: forall x y. nim x y → 0 ≤ y < x
```

À la différence du lemme `L3`, cependant, celui-ci ne peut être prouvé par aucun démonstrateur SMT. En effet, la preuve nécessite une récurrence sur l'arbre de dérivation, ce qui est inaccessible aux démonstrateurs SMT, comme nous allons l'expliquer dans la section suivante. Tout l'objet de cet article est de parvenir à vérifier un lemme comme `L4` avec Why3.

2.2 Traduction vers les démonstrateurs SMT

Avant de montrer comment réaliser une preuve par récurrence avec Why3, il convient de rappeler pourquoi cela ne peut pas être fait directement par les démonstrateurs SMT.

La logique de Why3 a beau être relativement simple, sa traduction vers celle des démonstrateurs SMT, à savoir la logique SMT-LIB [BFT16], reste un processus relativement complexe. Il implique notamment de traduire le polymorphisme paramétrique vers des types simples [BP11], mais aussi de traduire les types algébriques, les définitions récursives et les prédicats inductifs vers une logique où ces notions n'existent pas [2]. Ainsi, une déclaration d'un type comme celui des listes polymorphes

```
type list α = Nil | Cons α (list α)
```

donne lieu, du côté du démonstrateur SMT, à de nombreuses déclarations, dont [3] :

- un type `list` ;
- des fonctions non interprétées `Nil` et `Cons` ;
- des fonctions de discrimination et de projection, qui expriment notamment que les constructeurs sont injectifs et deux à deux disjoints ;
- une fonction `match_list` qui sera utilisée pour traduire le filtrage sur le type `list` ;
- un axiome d'inversion qui exprime que toute liste est formée par `Nil` ou `Cons`.

2. Il y a bien une notion de type algébrique dans SMT-LIB, mais il est très difficile de l'utiliser dans un contexte où le polymorphisme de Why3 doit être traduit vers des types simples.
3. On simplifie un peu les choses ici en omettant toute la complexité liée à l'encodage du polymorphisme.

Une exception notable est *le principe d'induction* sur le type `list`, qui ne saurait être exprimé en logique du premier ordre.

Si on considère maintenant le lemme `L2` vu plus haut énonçant que la longueur d'une liste est positive ou nulle

```
lemma L2: forall l: list α. length l ≥ 0
```

alors la définition de `length` est bien envoyée au démonstrateur SMT (sous la forme d'un axiome utilisant la fonction `match_list`) mais rien ne permet le raisonnement par récurrence nécessaire à cette preuve. La traduction de WhyML vers SMT-LIB est donc incomplète.

On retrouve une incomplétude similaire dans la traduction de la déclaration Why3 d'un prédicat inductif. Si on reprend l'exemple du prédicat `nim`,

```
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
```

on trouve du côté SMT les déclarations suivantes :

— un prédicat non interprété `nim` ;
— deux axiomes correspondant à `Step` et `Path` ;
— un axiome d'inversion qui dit que toute instance de `nim` est obtenue par `Step` ou `Path`.

Là encore, le principe d'induction associé à la définition de `nim` ne peut être exprimé en logique du premier ordre, ce qui explique l'incapacité des démonstrateurs SMT à prouver un lemme comme `L4`.

2.3 Preuves par récurrence

Comme on vient de l'expliquer, une preuve par récurrence n'est pas à la portée des démonstrateurs SMT. Il faut donc la réaliser en amont, au niveau de l'outil Why3. À l'instar de ce qui existe dans Dafny [LL23], Why3 propose ainsi une poignée de transformations logiques qui appliquent d'une manière heuristique le principe d'induction associé à un type algébrique ou à un prédicat inductif sur un but universellement quantifié. De telles transformations doivent être explicitement déclenchées par l'utilisateur, par exemple au travers de l'interface graphique Why3 IDE.

Une autre approche, très répandue dans le domaine de la vérification déductive, consiste à matérialiser une preuve par récurrence sous la forme d'un *programme*. Dans Why3, les programmes associés sont dénommés *fonctions-lemmes* (en anglais *lemma functions*). Ainsi, on peut démontrer que la longueur d'une liste est positive ou nulle avec la fonction suivante :

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

Il s'agit là d'une fonction récursive écrite dans WhyML, le langage de programmation de Why3, ainsi que l'annonce `let rec`. Le mot clé `lemma` précise que la portée de cette fonction reste cependant limitée au *code fantôme*. Cette fonction ne renvoie rien (son type de retour est `unit`) car son seul objectif est de montrer la validité de sa postcondition (la clause `ensures`). Une fonction-lemme se doit de terminer, ce qui est ici justifié par un variant. Enfin, le corps de la fonction est laissé libre à l'utilisateur, dans les limites toutefois d'un code fantôme, c'est-à-dire exempt d'effet de bord observable (exception non rattrapée ou modification d'un état global). Ici, le corps de la fonction fait un filtrage sur la liste `l`, puis un appel récursif sur la queue de la liste dans le cas d'une liste non vide.

Une fois cette fonction écrite, elle est vérifiée avec Why3 comme tout autre programme. En particulier, le générateur de conditions de vérification de Why3 ignore complètement le caractère fantôme ou non d'un morceau de code [4]. Ici, la condition de vérification demande de vérifier la postcondition dans les deux branches du filtrage, exprime la décroissance du variant lors de l'appel récursif et apporte en contrepartie la postcondition de cet appel récursif. Au final, si on omet la décroissance du variant que Why3 a été capable de vérifier par lui-même et de ne pas inclure dans la condition de vérification, cette dernière se réduit à l'énoncé suivant :

```
length Nil ≥ 0 ∧ forall x, t. length t ≥ 0 → length (Cons x t) ≥ 0
```

C'est maintenant complètement à la portée d'un démonstrateur SMT, car il possède la définition de `length` d'une part et des capacités arithmétiques d'autre part.

Une fois la fonction-lemme `length_nonneg` vérifiée, l'énoncé correspondant, à savoir `forall l. length l ≥ 0`, est ajouté au contexte logique. C'est là le rôle du mot clé `lemma`. Par ailleurs, `length_nonneg` reste disponible comme fonction fantôme, que l'on peut appeler explicitement dans n'importe quel code WhyML pour obtenir que la longueur d'une liste donnée est positive ou nulle, épargnant ainsi au démonstrateur d'avoir à appliquer le lemme.

Comme on l'a compris, la fonction-lemme `length_nonneg` ne fait que réaliser une preuve par récurrence, avec un cas de base lorsque la liste est `Nil` et une hypothèse de récurrence explicitement invoquée par l'appel récursif sur la sous-liste `t` lorsque la liste est `Cons _ t`. À la différence d'une preuve faite avec un système comme Coq, cependant, on n'a pas implicitement appliqué un principe d'induction qui serait déduit du type `list`, mais on a explicitement donné un argument de bonne fondation (le variant) et explicitement invoqué l'hypothèse de récurrence.

Notre objectif dans cet article est de pouvoir faire la même chose pour une preuve par récurrence sur une instance d'un prédicat inductif. Si on reprend l'exemple du prédicat `nim` donné plus haut, et de la propriété que nous voulions montrer à son sujet, cela veut dire étendre l'outil Why3 avec la possibilité de définir une fonction-lemme de la forme suivante :

```
let rec lemma nim_bounds (x z: int)
  requires { nim x z    }
  ensures  { 0 ≤ z < x }
= ...
```

Cette extension à Why3 est présentée dans la section suivante.

3 Raisonner sur les prédicats inductifs

Pour atteindre notre objectif, à savoir définir une fonction-lemme qui réalise une preuve par récurrence sur une instance inductive, il faut pouvoir d'une part réaliser une analyse par cas sur cette instance et d'autre part permettre des appels récursifs sur des instances plus petites. Cela prend donc la forme de deux extensions à l'outil Why3, à savoir

— une nouvelle construction `match inductive` pour effectuer un filtrage sur une instance inductive ;
— une nouvelle sorte de variant, associée à une instance inductive.

Nous les présentons successivement, puis nous expliquons comment sont calculées les conditions de vérification associées à ces extensions.

3.1 Filtrage sur une instance inductive

Nous ajoutons une nouvelle construction `match inductive` au langage WhyML. De manière similaire à la construction `match` de filtrage sur une valeur d'un type algébrique,

4. C'est uniquement pendant le typage que Why3 fait une distinction, pour assurer la non interférence entre le code fantôme et le code matériel.

cette nouvelle construction permet un filtrage sur une instance d'un prédicat inductif. Comme pour la construction `match`, il faut pouvoir nommer l'instance sur laquelle le filtrage est réalisé. Mais contrairement à `match`, cette instance n'est pas une valeur du programme mais une hypothèse. Il se trouve qu'il existe déjà dans Why3 une notation pour nommer une clause d'un contrat de fonction. Nous pouvons la mettre à profit pour nommer une instance inductive correspondant à une précondition. Ainsi, nous pouvons démarrer comme ceci notre fonction-lemme,

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z    }
  ensures    { 0 ≤ z < x }
= match inductive H with
  ...
```

où `H` désigne l'instance du prédicat inductif `nim` sur laquelle porte la preuve par récurrence. Les différentes clauses de ce nouveau filtrage correspondent à celles de la définition inductive. Ici, il s'agit des deux clauses `Step` et `Path` et on peut donc écrire le filtrage suivant :

```
= match inductive H with
  | Step x z H1      → ...
  | Path x y z H1 H2 → ...
  end
```

On a nommé les variables universellement quantifiées dans chaque clause — en reprenant les noms `x` et `z` tant qu'à faire, mais ce n'est pas une obligation. On a également nommé les hypothèses logiques apportées par chacune des clauses. Dans le cas de `Step`, l'hypothèse `H1` correspond à `nim_step x z`. Dans le cas de `Path`, l'hypothèse `H1` correspond à `nim x y` et l'hypothèse `H2` correspond à `nim y z`.

On est maintenant en mesure de compléter notre fonction-lemme, en réalisant les deux appels récursifs qui vont nous apporter les deux hypothèses de récurrence.

```
= match inductive H with
  | Step _ _ _     → ()
  | Path _ y _ _ _ → nim_bounds x y; nim_bounds y z
  end
```

Ici, on s'est épargné la peine de nommer les variables et les hypothèses qui ne sont pas mentionnées dans les membres droits du filtrage. Il nous reste cependant à montrer la terminaison de cette fonction-lemme, ce que nous faisons dans la section suivante.

Avant cela, ajoutons trois précisions sur la construction `match inductive`. D'une part, le filtrage n'a pas besoin d'être exhaustif : si des clauses manquent, Why3 demandera de montrer qu'elles sont absurdes. D'autre part, les motifs sont à l'heure actuelle limités au premier niveau, c'est-à-dire que les clauses ne peuvent être imbriquées dans un motif. Pour autant, il est possible de le réaliser manuellement, comme ceci,

```
match inductive H with Path _ _ _ H1 _ → match inductive H1 with ...
```

c'est-à-dire en effectuant un second filtrage sur l'une des hypothèses apportées par une clause. Enfin, la construction `match inductive` doit nécessairement se trouver à une *position terminale* dans la fonction-lemme, pour des raisons qui seront expliquées dans la section 3.3.

3.2 Terminaison

Pour justifier la terminaison d'une fonction-lemme qui opère récursivement sur une instance d'un prédicat inductif, nous étendons la notion de variant de Why3 avec une nouvelle sorte, introduite par le mot clé `inductive`.

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z }
  variant    { inductive H }
= ...
```

Comme pour `match inductive`, on se sert ici du nom `H` de la précondition pour désigner l'instance qui sert de variant. Lors de l'analyse de la fonction-lemme, Why3 maintient un ensemble d'instances obtenues par déconstruction du variant inductif spécifié. Ainsi, dans la branche de filtrage

```
= match inductive H with
  | Path x y z H1 H2 → ...
```

les instances `H1` et `H2` ont été obtenues par déconstruction de `H` et seront donc considérées comme plus petites, c'est-à-dire comme décroissantes dans un appel récursif. Cet ensemble d'instances collectées est clos par transitivité : si une hypothèse `A` est obtenue par la déconstruction du variant inductif `H` avec `match inductive`, alors toute hypothèse obtenue elle-même par la déconstruction de `A` sera également considérée comme plus petite que `H`.

Au final, le code complet de notre fonction-lemme est donc le suivant :

```
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
```

Il reste à expliquer comment Why3 construit les conditions de vérification pour une telle fonction-lemme, et c'est ce que nous allons faire maintenant.

3.3 Conditions de vérification

Arrivés à ce point, il nous faut étendre le générateur de conditions de vérification de Why3 pour l'adapter à ces deux nouvelles constructions. Plutôt que d'ajouter une règle spécifique au générateur pour la construction `match inductive`, nous avons fait le choix de l'encoder en un ensemble d'instructions WhyML existantes, permettant de retrouver la condition de vérification voulue. Pour chaque branche de `match inductive`, nous devons démontrer les postconditions de la fonction-lemme parente, sous les hypothèses provenant du cas inductif traité. La condition de vérification correspondante peut être obtenue via une fonction fantôme locale, dont les paramètres et les préconditions sont fournies par la clause inductive associée, et dont la postcondition est celle de la fonction-lemme (d'où la nécessité de se trouver à une position terminale). Le corps de cette fonction locale est alors le code de la branche du filtrage.

Dans la figure 1, nous illustrons cette traduction sur l'exemple de notre fonction-lemme `nim_bounds`. À gauche, nous avons le code Why3 écrit par l'utilisateur avec `match inductive` et à droite le code traduit vers le langage interne de Why3 qui ne contient pas de construction `match inductive`. La construction `match inductive` a été remplacée par deux fonctions fantômes locales, en reprenant les noms `Step` et `Path` (lignes 6 et 12).

Pour la fonction `Step`, on a deux paramètres `x1` et `y1` qui reprennent les quantificateurs universels de la clause `Step` et une précondition ligne 7 correspondant à l'hypothèse `nim_step`. La précondition ligne 8 exprime, par unification, qu'il s'agit là de l'instance `H`. La postcondition ligne 9 reprend telle quelle la postcondition de la ligne 3. Enfin, le corps de la fonction `Step` (ligne 10) correspond au membre droit du filtrage (ici le code trivial `()`).

```
1  let rec lemma nim_bounds (x z: int)
2    requires H { nim x z }
3    ensures    { 0 ≤ z < x }
4    variant    { inductive H }
5  = match inductive H with
6    | Step _ _ _     →
7
8
9
10     ()
11
12   | Path _ y _ _ _ →
13
14
15
16
17       nim_bounds x y;
18       nim_bounds y z
19   end
20
```

```
1  let rec lemma nim_bounds (x z: int)
2    requires H { nim x z }
3    ensures    { 0 ≤ z < x }
4    variant    { inductive H }
5  =
6    let ghost Step (u v: int)
7      requires { nim_step u v }
8      requires { x = u ∧ z = v }
9      ensures  { 0 ≤ z < x }
10   = ()
11   in
12   let ghost Path (u y v)
13     requires { nim u y }
14     requires { nim y v }
15     requires { x = u ∧ z = v }
16     ensures  { 0 ≤ z < x }
17   = nim_bounds x y;
18     nim_bounds y z
19   in
20   assume { false }; absurd
```

Figure 1. Traduction de la contruction `match inductive`.

La fonction `Path` (ligne 12) exprime que l'on se trouve dans le cas `Path` (lignes 12–14) correspondant à l'instance `H` (ligne 15), avec la même postcondition (ligne 16) et là encore un corps directement repris du code initial, à savoir les deux appels récursifs (lignes 17–18).

Si on parvient à vérifier la correction de ces deux fonctions, alors on aura prouvé que la postcondition tient quel que soit la forme de l'instance `H`. Dès lors, il n'y a plus rien à faire ! Pour ne pas avoir à reprouver la postcondition de la fonction-lemme, nous ajoutons une hypothèse locale de `false`, permettant de valider trivialement la suite de la condition de vérification. Pour satisfaire la contrainte de typage imposée par le type de retour de la fonction-lemme, nous concluons par l'instruction `absurd`, qui prend un type arbitraire. Nous obtenons ainsi la condition de vérification suivante (sans obligations de preuve correspondant à la preuve de terminaison, détaillées plus bas) :

```
forall x:int, z:int. nim x z →
 (forall u:int, v:int. nim_step u v ∧ x = u ∧ z = v → 0 ≤ z < x) ∧
 (forall u:int, y:int, v:int. nim u y ∧ nim y v ∧ x = u ∧ z = v →
   nim x y ∧ (0 ≤ y < x → nim y z ∧ (0 ≤ z < y → 0 ≤ z < x))) ∧
 (false → false)
```

Si d'aventure le filtrage de `match inductive` n'était pas exhaustif, des fonctions similaires seraient néanmoins générées pour les cas manquants, mais avec un corps réduit à `absurd`. Ainsi, on aurait à prouver que ces cas sont exclus.

Décroissance du variant. Il reste à exprimer la décroissance du variant dans la condition de vérification d'une fonction comme `nim_bounds`. Pour ce faire, Why3 collecte, en tout point à l'intérieur de la fonction-lemme, les instances qu'il sait être plus petites que le variant (ici `H`) car obtenues par déconstruction de celui-ci. Dans le cas `Step`, il n'y en a pas. Dans le cas `Path`, en revanche, il y en a deux, à savoir `nim u y` et `nim y v`. Lors d'un appel récursif, Why3 exprime la décroissance du variant par une *disjonction* qui exprime que le variant correspondant à cet appel doit s'unifier avec l'une des instances collectées. Ainsi, l'appel

récursif de la ligne 17, sur `x` et `y`, conduit à l'obligation de preuve

$$(\mathtt{x} = \mathtt{u} \wedge \mathtt{y} = \mathtt{y}) \vee (\mathtt{x} = \mathtt{y} \wedge \mathtt{y} = \mathtt{v})$$

et de même l'appel récursif de la ligne 18, sur `y` et `z`, conduit à l'obligation de preuve

$$(\mathtt{y} = \mathtt{u} \wedge \mathtt{z} = \mathtt{y}) \vee (\mathtt{y} = \mathtt{y} \wedge \mathtt{z} = \mathtt{v})$$

Les deux seront trivialement vérifiées. Par ailleurs, dans le cas de deux fonctions-lemmes mutuellement récursives (traitant un couple de prédicats inductifs qui se réfèrent l'un à l'autre), il suffit de vérifier que le variant inductif correspondant à l'appel de l'une est obtenu par déconstruction du variant de l'autre, et inversement.

Un avantage de cette approche est qu'elle se marie bien avec la notion de variant lexicographique que l'on trouve déjà dans Why3. Ainsi, le variant d'une fonction-lemme pourrait être aussi complexe qu'un triplet

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

où `n` est un entier, `H` une instance inductive et `t` un arbre. Vu que l'on sait exprimer que le variant inductif `H` est inchangé (il suffit d'exprimer que l'instance est `H`) et que l'on sait exprimer sa décroissance (comme expliqué ci-dessus), alors on sait le combiner dans un variant lexicographique.

4 Application : une théorie du λ-calcul

Afin de tester cette nouvelle construction `match inductive`, nous proposons une traduction directe vers WhyML d'une formalisation en Coq du λ-calcul par Gérard Huet [Hue94]. Il s'agit là de plusieurs résultats classiques sur le λ-calcul, comme on peut les trouver dans un cours de calculabilité, initialement présentés à la fin des années 1980 par Gérard Huet dans des notes de cours mettant en avant le langage Caml (à l'époque, il ne s'appelle pas encore OCaml) comme véhicule pour programmer les différentes notions introduites autour du λ-calcul [Hue11]. Cette implémentation fut ensuite partiellement traduite en Coq, et c'est cette preuve Coq sur laquelle nous nous basons pour notre propre traduction. Cette formalisation contient un grand nombre de preuves par induction — 2 types algébriques, 11 prédicats inductifs et 78 preuves par induction — et constitue donc un parfait exemple d'une preuve que nous souhaiterions pouvoir écrire en Why3 avec `match inductive`. Nous proposons une traduction la plus fidèle possible à la preuve originale.

4.1 Présentation de la preuve Coq

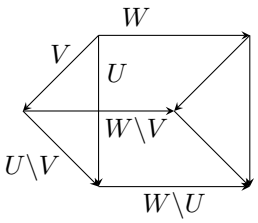
La formalisation en Coq représente les termes du λ-calcul avec des indices de de Bruijn, avec le type `lambda` qu'on trouve dans la figure 2. Sur ce type sont définis un certain nombre de fonctions et de prédicats, comme par exemple un prédicat `beta` correspondant à la β-réduction, ainsi que plusieurs prédicats inductifs, comme par exemple un prédicat `red` pour la clôture réflexive transitive de `beta` (voir figure 2). Les propriétés de ces fonctions et prédicats sont ensuite énoncées et prouvées. Ainsi, le lemme `red_abs` de la figure 2 exprime que la β-réduction peut être effectuée sous un λ. Comme on le voit, la preuve procède par induction sur l'hypothèse `red M M'` (via la tactique `induction 1` de Coq).

Au-delà de ces notions élémentaires, la preuve Coq définit également les notions de réduction parallèle, de sous-ensembles de rédexes identifiés dans un même terme ou encore de rédexes résiduels ([Hue94], section 6). Cette dernière notion est centrale dans le développement. Notée $A\backslash B$, elle représente l'évolution d'un ensemble de rédexes A à travers la réduction parallèle d'un autre ensemble de rédexes B dans le même terme.

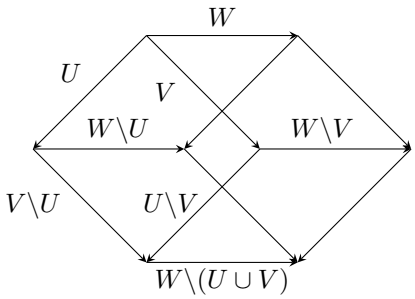
Le théorème du prisme indique alors que, pour un ensemble de rédexes compatibles U, V, et W, avec $V \subset U$, on a $W\backslash U = (W\backslash V)\backslash(W\backslash U)$, dont l'illustration justifie le nom :

```
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.
(* ... *)
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.
```

Figure 2. Un petit extrait de la preuve Coq.



On en déduit ensuite le lemme du cube de Lévy, qui indique que pour tous ensembles de rédexes compatibles U, V, et W, on a $(W\backslash V)\backslash(U\backslash V) = (W\backslash U)\backslash(V\backslash U)$, ce qui s'illustre également très bien :



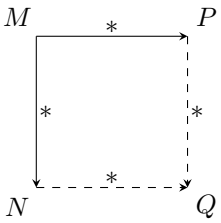
Par ailleurs, la preuve Coq inclut également la propriété de Church-Rosser pour la β-conversion, par la méthode de Tait-Martin-Löf, c'est-à-dire par le lemme des étapes parallèles, qui indique que si un terme M se réduit en deux termes N et P distincts à travers une étape de β-réduction parallèle, alors N et P se réduisent tous deux vers un terme commun Q à travers une autre étape.

```
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

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
```

Figure 3. Traduction en Why3 de la figure 2.



La preuve Coq représente au total 1263 lignes de code, avec 511 lignes de spécification et 768 lignes de preuve, tel qu'indiqué par l'outil `coqwc`. Plus intéressant dans notre cas, la preuve contient 42 occurrences de la tactique `induction` appliquée à une instance d'un prédicat inductif, représentant plus de la moitié des applications de cette tactique au total.

4.2 Traduction vers Why3

La traduction vers Why3 est une traduction la plus directe possible de la preuve Coq. La figure 3 illustre ainsi comment les éléments Coq de la figure 2 sont traduits vers Why3. Le type `lambda` devient ainsi un type algébrique, où la seule différence est l'utilisation en Why3 du type `int`, en l'absence d'un type `nat`. Le prédicat `red` reste un prédicat inductif, défini exactement de la même manière. Enfin, la preuve du lemme `red_abs` devient en Why3 une fonction-lemme, à la manière de ce que nous avons expliqué dans la section précédente. On peut constater que les deux preuves sont d'une taille comparable, mais qu'elles diffèrent néanmoins. Du côté Coq, la preuve par récurrence est concentrée dans la tactique `induction`, là où Why3 doit introduire une fonction-lemme et des appels récursifs explicites. Inversement, du côté Why3, tous les détails de la preuve sont laissés aux démonstrateurs SMT, là où la preuve Coq est relativement détaillée.

Résultat. La preuve finale en WhyML représente 997 lignes de code, avec 424 lignes de spécification et 573 lignes de preuve, tel qu'indiqué par `why3 wc`. La preuve est donc légèrement en dessous de 75% de l'équivalent en Coq. L'écrasante majorité du code WhyML

pour la preuve est composée d'instances de `match inductive` pour effectuer les analyses de cas sur les prédicats, et d'appels récursifs apportant les hypothèses de récurrence nécessaires pour la conclusion de la postcondition finale. Toutes les conditions de vérification sont prouvées par les assistants de preuve automatiques avec peu ou pas d'intervention humaine, mis à part l'application occasionnelle de la tactique `split_vc`. Nous avons bien conscience que la preuve Coq dont nous sommes partis n'est pas forcément représentative de ce qui serait fait en Coq aujourd'hui — cette preuve n'a pas été mise à jour dans les versions récentes de Coq depuis 2006. Pour autant, elle nous permet amplement de valider notre travail par le nombre de prédicats inductifs et de preuves par récurrence qu'elle contient.

5 Travaux connexes

Les prédicats inductifs de Why3 sont similaires aux définitions inductives de Coq [PM96], même si plus limités car la logique de Why3 ne contient ni types dépendants ni ordre supérieur. Dans le système Coq, une définition inductive comme `nim` est automatiquement associée à un principe d'induction d'ordre supérieur.

```
nim_ind: forall P: Z → Z → Prop,
  (forall x y: Z, nim_step x y → P x y) →
  (forall x y z: Z, nim x y → P x y → nim y z → P y z → P x z) →
  forall z z0: Z, nim z z0 → P z z0
```

Lorsque la tactique `induction` est utilisée pour réaliser une preuve par récurrence, ce principe d'induction est instancié en fonction du but courant, et les buts obtenus correspondent aux différents constructeurs de la définition inductive. Cette approche est assez différente de celle que nous avons présentée dans cet article, où c'est plutôt à l'utilisateur de choisir un principe d'induction particulier au travers des appels récursifs effectués. Cela étant, dans les cas simples où la preuve Why3 suit le principe d'induction que Coq aurait généré, les preuves Coq et Why3 sont très similaires. Il suffit en effet de simplifier le terme de preuve Coq, en expansant la définition du principe d'induction puis en effectuant quelques β-réductions, pour retrouver un programme isomorphe à la fonction-lemme Why3. L'approche Why3 par fonction-lemme trouve cependant un avantage lorsque la récurrence ne suit justement pas le principe d'induction qui se dérive naturellement de la définition du prédicat. Les utilisateurs de Coq connaissent bien le problème, et doivent parfois utiliser la commande `Scheme`, voire prouver leurs propres principes d'induction.

Pour être complet dans la comparaison des approches Coq et Why3, ajoutons que le système Coq ne fait pas de différence entre un type algébrique et un prédicat inductif. Les deux entrent dans le cadre général des définitions inductives [PM96]. Dans Coq, le principe d'induction généré pour un type algébrique comme `list` permet de définir une fonction récursive sur les listes, ou faire une preuve par récurrence sur un énoncé portant sur toute liste, et le principe d'induction généré pour un prédicat comme `nim` permet une preuve par récurrence sur une hypothèse inductive. Dans Why3, en revanche, on fait une différence entre un type comme `list`, qui existe à la fois dans le langage de programmation WhyML et dans le langage logique, et un prédicat inductif comme `nim` qui n'existe que dans le langage logique et le code fantôme.

Comme Coq, le système Isabelle/HOL génère automatiquement un principe d'induction pour chaque définition inductive, pour utilisation lors de l'application de tactiques. Dans le système F* [SHK$^+$16], une preuve par récurrence est réalisée par une fonction-lemme, comme avec Why3. Le chapitre 8 de l'excellent livre *Proof-Oriented Programming in F** [SMR24] l'illustre très bien. Notons que, comme Coq, le système F* ne fait pas de différence entre type algébrique et prédicat inductif.

Le système Dafny [Lei10, LL23] propose des prédicats dits *extrêmes*, qui représentent les plus petits ou plus grands points fixes de leurs définitions, au choix de l'utilisateur. Les preuves par récurrence sur ces prédicats sont ensuite effectuées par le biais de fonctions-lemmes

récursives, à l'instar de Why3. Par ailleurs, pour un but donné, Dafny tente d'appliquer automatiquement, par le biais d'une heuristique, le principe d'induction qui découle de la définition d'un prédicat extrême [Lei12], permettant ainsi de prouver immédiatement un grand nombre de propriétés qui ne nécessitent qu'une récurrence simple et des appels au démonstrateur SMT derrière Dafny. Si cela échoue, l'utilisateur peut fournir un corps pour la fonction-lemme permettant de guider la preuve. L'analyse de cas se fait alors par des conditions booléennes sur les arguments du prédicat, et non par analyse structurelle de la dérivation de preuve.

Les fonctions-lemmes dans Why3 sont souvent utilisées pour prouver des propriétés pour des fonctions ou prédicats définis récursivement. Ce cas d'usage se rapproche d'une technique appelée *induction fonctionnelle*, présente dans des systèmes comme Coq, Isabelle/HOL ou encore Lean4. Contrairement à ce qui est fait dans ces systèmes, Why3 ne génère pas de principe d'induction dédié à la fonction ou prédicat en question, et laisse le programmeur construire manuellement la preuve par induction sous forme de programme.

6 Conclusion et perspectives

Nous avons présenté une extension de l'outil Why3 permettant de réaliser des preuves par récurrence sur des prédicats inductifs, tout en restant dans le contexte de la logique du premier ordre. Cette extension permet à l'utilisateur d'écrire des fonctions-lemmes qui procèdent par analyse par cas sur la structure d'une instance inductive et effectuent des appels récursifs bien fondés, y compris dans le cas d'un groupe de prédicats inductifs mutuellement dépendants. En interne, cette extension se ramène à des constructions existantes de Why3 et la correction de cette réduction est assurée, de manière méta, par le principe d'induction associé à la définition inductive.

Dans son implémentation actuelle, notre extension à Why3 reste limitée. Elle ne permet pas, notamment, de définitions inductives où le branchement de l'arbre de dérivation serait infini. Il nous semble possible de couvrir également ce cas de figure, mais cela reste à étudier proprement. Nous ne traitons pas non plus le cas des prédicats coinductifs.

Pour faciliter le travail du programmeur, il serait intéressant de voir à quel point la construction de fonctions-lemmes récursives traitant des prédicats inductifs pourrait être automatisée, à l'instar de la génération automatique de principes d'induction dans le cadre de l'induction fonctionnelle.

Malgré les limitations actuelles, la nouvelle construction `match inductive` est d'ores et déjà très utile, comme nous l'avons démontré en répliquant en Why3 un développement faisant usage de très nombreuses récurrences sur une dizaine de prédicats inductifs différents. Ce genre de raisonnement est très fréquent dans le domaine des méthodes formelles, comme par exemple des preuves de sûreté de systèmes de types ou des preuves de compilateurs. Une prochaine étape de notre travail serait donc de voir comment de telles preuves pourraient être réalisées dans Why3 maintenant qu'il dispose de `match inductive`.

Remerciements. Les auteurs remercient Alain Giorgetti pour sa relecture détaillée d'une première version de cet article.

Références

[BBB+22] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli et Yoni Zohar : cvc5 : A versatile and industrial-strength SMT solver. *In* Dana Fisman et Grigore Rosu, éditeurs :

Tools and Algorithms for the Construction and Analysis of Systems, volume 13243 de *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[BDJ+05] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang et K. Rustan M. Leino : Boogie : A Modular Reusable Verifier for Object-Oriented Programs. *In FMCO*, volume 4111 de *LNCS*, 2005.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché et Andrei Paskevich : Why3 : Shepherd your herd of provers. *In Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. `https://hal.inria.fr/hal-00790310`.

[BFT16] Clark Barrett, Pascal Fontaine et Cesare Tinelli : The Satisfiability Modulo Theories Library (SMT-LIB). `https://smt-lib.org/`, 2016.

[BP11] François Bobot et Andrei Paskevich : Expressing Polymorphic Types in a Many-Sorted Language. *In* Cesare Tinelli et Viorica Sofronie-Stokkermans, éditeurs : *Frontiers of Combining Systems, 8th International Symposium, Proceedings*, volume 6989 de *Lecture Notes in Computer Science*, pages 87–102, Saarbrücken, Germany, octobre 2011.

[CCIM18] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala et Alain Mebsout : Alt-Ergo 2.2. *In SMT Workshop : International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, juillet 2018.

[CJF24] Joshua M Cohen et Philip Johnson-Freyd : A formalization of Core Why3 in Coq. *Proceedings of the ACM on Programming Languages*, 8(POPL):1789–1818, 2024.

[dMB] Leonardo de Moura et Nikolaj Bjørner : Z3, an efficient SMT solver. `http://research.microsoft.com/projects/z3/`.

[Fil13] Jean-Christophe Filliâtre : One logic to use them all. *In International Conference on Automated Deduction*, pages 1–20. Springer, 2013.

[FP13] Jean-Christophe Filliâtre et 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.

[Hue94] Gérard Huet : Residual theory in λ-calculus : a formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[Hue11] Gérard Huet : Constructive computation theory. *Notes de cours*, 1988–2011.

[Lei10] 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.

[Lei12] K. Rustan M. Leino : Automating induction with an SMT solver. *In Proc. 13th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, Philadelphia, PA, 2012.

[LL23] K.R.M. Leino et K. Leino : *Program Proofs*. MIT Press, 2023.

[MSS16] P. Müller, M. Schwerhoff et A. J. Summers : Viper : A verification infrastructure for permission-based reasoning. *In* B. Jobstmann et K. R. M. Leino, éditeurs : *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 de *LNCS*, pages 41–62. Springer-Verlag, 2016.

[PM96] Christine Paulin-Mohring : *Définitions inductives en théorie des types d'ordre supérieur*. Thèse d'habilitation, Université Claude Bernard-Lyon I, Décembre 1996.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué et Santiago Zanella-Béguelin : Dependent types and multi-monadic effects in F*. *In 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, janvier 2016.

[SMR24] Nikhil Swamy, Guido Martínez et Aseem Rastogi : *Proof-Oriented Programming in F**. Available online, 2024. `https://fstar-lang.org/tutorial/proof-oriented-programming-in-fstar.pdf`.