PSTL

Ahmed Youra,Koné Daba

May 4, 2025

1 Introduction

Notre projet PSTL consiste à créer un parser qui va prendre un langage why3 simplifié et le parser dans un langage why3 complet afin de pouvoir faire des vérifications sur les fonctions qu'on a voulu écrire. L'objectif présent du projet est de créer un système capable de :

- Lire un fichier source en format `.rew` contenant des définitions de fonctions et des spécifications.
- Analyser ce fichier pour en extraire l'AST (Arbre Syntaxique Abstrait).
- Transformer cet AST en un fichier `.why` qui contient une spécification formelle en WhyML.

2 Présentation du cahier des charges

L'objectif principal du projet est de concevoir un outil capable de transformer automatiquement un fichier écrit dans un langage source simplifié (`.rew`) en un fichier au format WhyML (`.why`), utilisable directement par l'outil de vérification Why3.

Pour cela, les tâches à réaliser sont les suivantes :

1. **Création d'un lexer et parser** pour analyser le code source `.rew` et construire un AST (Arbre Syntaxique Abstrait).
2. **Définition d'une structure d'AST adaptée** au langage cible, capable de représenter les éléments de WhyML (fonctions, types, instructions, spécifications, etc.).

3. **Génération d'un fichier** `.why` à partir de l'AST, en respectant la syntaxe du langage WhyML.

4. **Gestion de la spécification partielle via le mot-clé** `spec` et génération des fonctions intermédiaires sous forme de `val`.

5. **Prise en charge progressive d'éléments complexes** comme les boucles `while`, les invariants, les variants, les blocs conditionnels `if / else`, et les appels imbriqués.

6. **Production d'un code WhyML vérifiable**, compatible avec les prouveurs automatiques tels que `Alt-Ergo`, `CVC4`, etc.

7. **Vérification fonctionnelle avec Why3** pour chaque transformation d'étape, afin de garantir que le code généré est formellement correct.

8. **Écriture d'un rapport final et préparation d'une soutenance** présentant le fonctionnement de l'outil, son architecture, et les résultats obtenus.

Le projet est réalisé en **OCaml**.

3 Premier exemple : le parsing d'un fichier `.rew`

Pour initier le projet, nous avons commencé par un premier exemple très simple : parser un petit bout de code inspiré du langage Why3. Ce fichier source, au format `.rew`, contenait la définition d'une fonction avec ses préconditions et postconditions.

L'objectif de cette étape était de :

- Lire le fichier `.rew` contenant du code simplifié,
- Le parser à l'aide d'un **lexer** (écrit avec `ocamllex`) et d'un **parser**,
- Générer un **AST** (Arbre Syntaxique Abstrait) structuré, permettant de manipuler le contenu du fichier dans notre programme OCaml.

Cette étape ne produisait pas encore de fichier `.why`, mais posait les bases du système de transformation, en validant notre capacité à lire et structurer un fichier en langage spécifique.

3.1 Construction de l'AST

L'AST que nous avons construit est une structure de données définie en OCaml. Elle nous permet de représenter :

- Le nom de la fonction et ses paramètres,
- Les types (paramètres d'entrée, type de retour),
- Les clauses `requires`, `ensures`.
- Le corps de la fonction, ou une indication que celui-ci est vide (par exemple via la notation `= <>`).

4 Étape 0 : Génération d'un fichier `.why` depuis un fichier `.rew`

Après avoir mis en place notre lexer, notre parser et notre structure d'AST, nous avons travaillé sur un premier cas de transformation concrète. Cette première étape consistait à prendre un fichier très simple en langage `.rew`, et à en générer automatiquement une version formelle en WhyML.

Voici le contenu initial du fichier `sqrt0.rew` :

```
use int.Int
use ref.Ref

let sqrt (x : ref int) : int =
  requires { x >= 0 }
  writes { x }
  ensures { result * result <= x <= (result + 1) * (result + 1) }
  ensures { !x = old !x }
= <>
```

Ce fichier définit une fonction `sqrt` avec un corps vide (`= <>`), mais incluant des préconditions et des postconditions. Il ne contient pas encore d'implémentation, uniquement la spécification.

Ajouts réalisés pour cette transformation

Pour permettre la génération d'un fichier `.why` exploitable par Why3, nous avons apporté plusieurs ajouts et choix techniques dans notre code :

- **Séparation automatique de la spécification et du corps** : nous avons introduit une logique dans notre parser pour détecter les parties contenant des spécifications (grâce au mot-clé spécial `Spec`) et distinguer cela du reste du corps.
- **Génération d'une fonction `val`** : nous avons ajouté la génération d'une déclaration `val sqrt0_spec` contenant les mêmes `requires`, `writes` et `ensures`, mais isolée du corps de la fonction.
- **Création d'une fonction `let` qui appelle la spécification** : la fonction `let sqrt` reprend les clauses de la spécification, et se contente d'appeler `sqrt0_spec x` comme corps.
- **Déréférencement automatique des références (`ref`)** : dans les expressions logiques (comme `requires`, `ensures`), les variables de type `ref` sont transformées automatiquement avec un `!`. Par exemple, `x >= 0` devient `!x >= 0`.
- **Génération syntaxiquement correcte du fichier `.why`** : le fichier produit est conforme à la syntaxe de WhyML et validé avec les outils Why3.

Résultat obtenu : `sqrt0.why`

Voici le fichier généré automatiquement par notre système :

```
use int.Int
use ref.Ref

val sqrt0_spec (x : ref int) : int
  requires { !x >= 0 }
  writes { x }
  ensures { (result * result <= !x < (result + 1) * (result + 1)) }
  ensures { !x = old !x }

let sqrt (x : ref int) : int
  requires { !x >= 0 }
  writes { x }
  ensures { (result * result <= !x < (result + 1) * (result + 1)) }
  ensures { !x = old !x } =
  sqrt0_spec x
```

Vérification avec Why3

Il est possible aussi de vérifier la validité de la fonction .why avec Why3 :

```
$ why3 prove -P alt-ergo sqrt0.why
File sqrt0.why:
Goal sqrt'vc.
Prover result is: Valid (0.01s, 0 steps)
```

Ce résultat montre que notre transformation respecte les attentes du système de vérification, et valide notre première étape de conversion.

5 Étape 1 : Spécification partielle avec implémentation simple

Après avoir géré un premier cas avec un corps vide (`= <>`) dans l'étape 0, nous avons abordé une seconde situation plus réaliste, dans laquelle une partie de l'implémentation est fournie directement dans le fichier `.rew`, précédée d'une clause `spec`.

Voici le fichier `sqrt1.rew` que nous avons utilisé :

```
use int.Int
use ref.Ref

let sqrt (x : ref int) : int =
  requires { x >= 0 }
  writes { x }
  ensures { result * result <= x <= (result + 1) * (result + 1) }
  ensures { !x = old !x }
= let r = ref 0 in
  let h = ref (x + 1) in
  spec requires { !r >= 0 }
       requires { !h >= 0 }
       ensures { (old !r) * (old !r) <= (!r * !r) <= x < (!r + 1) * (!r + 1) <= !
       writes {r, h}
  = <>
    !r
```

Cette fois, le corps de la fonction inclut :

- Deux initialisations (`let r = ref 0` et `let h = ref (x + 1)`),

- Une sous-spécification (clause `spec`) encadrée d'un bloc particulier,
- Un retour de valeur avec `!r`.

Travail réalisé

Pour gérer cette structure, nous avons apporté plusieurs modifications à notre structure :

- **Ajout du mot-clé** `spec` dans le lexer, le parser, et l'AST.
- **Séparation du corps en sous-blocs**, pour extraire la spécification du reste de la fonction.
- **Détection des variables nécessaires à la spécification** : récupération automatique des identifiants comme `r` et `h` avant `spec`.
- **Génération d'une fonction** `val sqrt1_spec` contenant uniquement les clauses `requires`, `writes` et `ensures`.
- **Génération de la fonction** `let sqrt`, avec appel à `sqrt1_spec x r h` suivi de `!r`.

6 Étape 2 : Gestion des boucles `while`, invariants et variants

Cette étape introduit le traitement des boucles `while` accompagnées de clauses d'invariant et de variant. Ces éléments sont essentiels en logique de programmation pour prouver la terminaison et la correction des boucles.

Principe de transformation

Le fichier source `.rew` que nous traitons contient une fonction avec une boucle `while`. Celle-ci est annotée avec une clause `invariant` exprimant les propriétés qui doivent rester vraies à chaque itération, et une clause `variant` représentant une mesure strictement décroissante pour garantir la terminaison.

Pour traduire cela en WhyML, nous avons opté pour une stratégie en deux étapes :

1. Génération d'une `val` appelée `sqrt2_spec` contenant la spécification formelle du corps de la boucle (les propriétés logiques qu'elle doit respecter).

2. Génération de la fonction principale `sqrt` qui déclare la boucle, intègre les invariants et variants, et appelle la fonction `sqrt2_spec` à chaque itération.

Traitements réalisés dans la conversion

Pour effectuer cette transformation, notre fonction OCaml `conv` effectue les étapes suivantes :

- **Ouverture du fichier de sortie et écriture des préliminaires** : les lignes `use ...` sont recopiées automatiquement à partir de l'AST. Si une clause `axiom` est présente, elle est également affichée à l'aide d'une fonction `affiche axiom`.

- **Séparation des blocs d'instructions** : le corps de la fonction est divisé en plusieurs parties à l'aide de la fonction `separe_list_expr`. Cela permet de distinguer l'initialisation des variables, les clauses de spécification (`spec`), et le corps de la boucle lui-même.

- **Détection de la position de la clause** `spec` : la fonction `find_spec` permet de retrouver à quel endroit la spécification est insérée dans le corps de la fonction. Cela nous permet ensuite d'identifier les instructions situées après `spec`, qui représentent généralement la boucle.

- **Création de la spécification** `val sqrt2_spec` : nous construisons manuellement une spécification complète contenant :

  - des `requires` qui expriment les préconditions du corps de la boucle,
  - des `reads` et `writes` générés automatiquement à partir des variables utilisées,
  - des `ensures` extraits à partir des invariants et variants de la boucle.

  Ce processus est effectué par la fonction `convert_spec_of_expr` qui parcourt les nœuds de type `Invariant` et `Variant` dans l'AST pour en extraire les expressions logiques à convertir en `ensures`.

- **Écriture manuelle des** `requires` : contrairement aux autres clauses, les `requires` de la spécification `sqrt2_spec` sont écrits à la main dans une chaîne de caractères. En effet, nous n'avons pas trouvé de règle systématique permettant de décomposer ou reformuler automatiquement un `requires` complexe. Par exemple, une condition comme `!r +`

`1 < !h` pourrait être imbriquée dans une structure logique qui empêche toute généralisation.

- **Génération de la fonction `let sqrt`** : une fois la spécification générée, nous produisons une version complète de la fonction `sqrt` qui :
  - effectue les initialisations des variables,
  - définit la boucle avec les `invariant` et `variant`,
  - appelle la spécification intermédiaire `sqrt2_spec` à chaque itération.

  L'appel à cette fonction dans le corps de la boucle permet d'isoler la logique de la boucle et de la rendre prouvable par Why3.
- **Écriture finale dans le fichier** : les chaînes générées sont finalement écrites dans le fichier grâce à `Printf.fprintf`, dans l'ordre suivant : `val sqrt2_spec`, son corps, `let sqrt`, et son corps.

7 Étape 3 : Ajout d'un branchement `if / else`

Dans cette troisième étape, nous avons poursuivi le développement de notre outil de traduction en nous attaquant à un autre cas plus complet. Il s'agissait cette fois de traiter une fonction contenant non seulement une boucle `while`, mais aussi un bloc `if / else`, intégré après une clause `spec`.

Le fichier source `.rew` utilisé est le suivant (non reproduit ici en entier). Il contient :

- Des clauses `requires`, `writes` et `ensures` principales sur la fonction `sqrt`.
- Une initialisation des bornes `r` et `h`.
- Une clause `spec` introduite avant une boucle `while`.
- Dans le corps de la boucle, un bloc conditionnel avec un `if`, un `else if` et un `else`.

L'objectif de cette étape est double :

1. Extraire la clause `spec` dans une fonction intermédiaire `val sqrt3 spec`, contenant uniquement des `requires`, `reads`, `writes` et `ensures`.
2. Générer une fonction `sqrt2` (sous-fonction) qui effectue les calculs dans la boucle et utilise un bloc `if / else`.

Travail réalisé

La conversion du fichier `.rew` en `.why` a demandé plusieurs étapes :

- **Détection de la clause `spec`** : nous avons utilisé notre analyseur pour parcourir la structure de la fonction et isoler le bloc contenant `spec`. L'index de ce bloc est calculé automatiquement.
- **Récupération des variables locales** : toutes les variables déclarées par `let ... in` (notamment `r`, `h`, `m`) sont extraites automatiquement afin d'être utilisées comme paramètres de la fonction intermédiaire `sqrt3 spec`.
- **Génération de `val sqrt3 spec`** :
  - Les `requires` sont écrits manuellement dans le générateur OCaml, car nous n'avons pas identifié de règle générale pour transformer un bloc spec en plusieurs `requires` à partir du fichier `.rew`.
  - Les paramètres sont les références locales nécessaires à la spécification : `r`, `m`, `h`.
  - Les clauses `reads`, `writes` et `ensures` sont également fixées manuellement dans cette étape.
- **Ajout du support du bloc `if / else`** :
  - Contrairement aux étapes précédentes, le corps de la boucle contient un véritable test logique : `if !m * !m < !x` suivi d'un `else if` et d'un `else`.
  - Nous avons enrichi l'analyse syntaxique pour supporter correctement ce type de structure conditionnelle.
  - Le bloc `if / else` est reconstruit ligne par ligne dans la génération de la fonction `sqrt2`.
- **Génération finale de `sqrt2`** :
  - Cette fonction contient les préconditions calculées à la main, les lectures/écritures (`reads`, `writes`), les postconditions, et le corps de la fonction.
  - La clause `sqrt3 spec r m h;` est insérée dans la fonction, suivie du bloc conditionnel construit avec les bonnes références et affectations.

Difficultés rencontrées

Le principal point de complexité dans cette étape a été la prise en charge du bloc `if / else`, qui nécessitait :

- une gestion plus fine de l'arbre syntaxique pour parcourir correctement les branches conditionnelles,
- une reconstruction syntaxique correcte du bloc WhyML généré.

Un autre point délicat était que la clause `spec` ne peut pas toujours être traduite automatiquement en plusieurs `requires` — par exemple, certains `requires` dépendent de propriétés implicites qu'il est difficile d'identifier sans contexte. Pour cette raison, les préconditions de la fonction intermédiaire ont été rédigées manuellement dans le générateur OCaml, en dur dans le code.

Résultat généré

Le fichier `.why` généré contient les trois blocs suivants :

1. `val sqrt3 spec (r m h : ref int) : ()`, avec ses `requires` et `ensures`.
2. `let sqrt2 (x r h : ref int) : ()`, avec le corps de la boucle, le bloc conditionnel, et les appels à `sqrt3 spec`.

Étape 4 – Génération du DIV

Dans cette quatrième étape, l'objectif est de générer automatiquement, à partir de notre AST enrichi, le corps de la fonction auxiliaire `sqrt3`, qui correspond à la sélection du pivot dans l'algorithme de dichotomie.

Extension du langage avec `div`. Nous avons d'abord étendu notre langage pour reconnaître les divisions entières utilisées dans le calcul du pivot. Pour cela :

- Le mot-clé `div` a été ajouté au lexer (`lexer.mll`) et au parser (`parser.mly`).
- Un nouveau constructeur `Div` a été introduit dans le type `expr` de l'AST.
- Une expression de la forme `div(a, b)` est désormais représentée dans l'AST sous la forme `Div(a, b)`.

Extraction ciblée du corps de `sqrt3`. Nous avons ensuite extrait automatiquement, depuis l'AST, le bloc correspondant à `sqrt3`. Pour cela :

- La fonction `ls_expr_afterSpec` récupère toutes les instructions apparaissant après la ligne `spec`.
- Puis, la fonction `ls_expr_untilDiv` sélectionne toutes les expressions jusqu'à la première apparition d'un `Div`, ce qui marque la fin du bloc à extraire.

Résultat généré. Le code produit dans le fichier `.why` est :

```
let sqrt3
  (r m h : ref int) : ()
  requires { !r + 1 < !h }
  reads {r,m,h}
  writes { m }
  ensures { !r < !m < !h }
= (* Select pivot. *)
  m := !r + (div (!h - !r) 2)
```

Conclusion

Ce projet nous a permis de concevoir un traducteur de fichiers `.rew` vers le langage WhyML, en plusieurs étapes progressives. À travers l'analyse lexicale et syntaxique, la génération d'un AST, puis la transformation structurée en code Why3, nous avons pu automatiser la production de fonctions accompagnées de spécifications formelles. Ce travail a également mis en évidence certaines limites, notamment l'impossibilité de systématiser la traduction de certaines annotations comme les `requires` complexes, ce qui a nécessité des ajouts manuels.