Hardened Golo : Donnez de la confiance en votre code Golo

Raphael Laurent

Rapport de Projet de Fin d'Études

Hardened Golo: Donnez de la confiance en votre code Golo

Raphael Laurent

21 juin 2016

Table des matières

1 Remerciements

Je tiens à remercier mon tuteur Nicolas Stouls, pour ses conseils tout au long de mon stage et son suivi rigoureux. Il m'a permis d'aller plus loin dans ma réalisation, ainsi que de comprendre différentes problématiques de mon sujet plus en profondeur. Son encadrement m'a permis de structurer ce projet correctement et de garantir une contribution riche.

Je remercie également Julien Ponge, pour ses indications éclairées dans le fonctionnement du compilateur Golo, ainsi que le langage lui-même.

Finalement, je remercie le laboratoire du CITI à l'INSA Lyon pour m'avoir accueilli pendant ce stage.

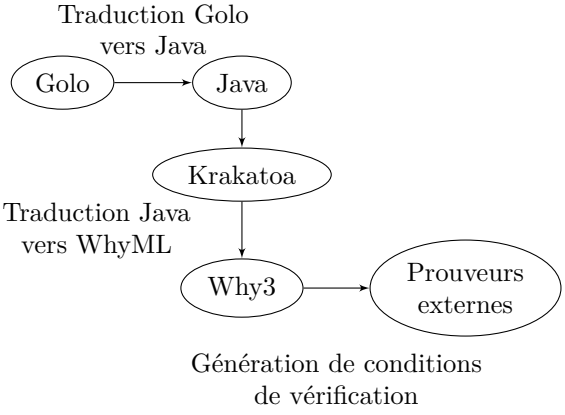
2 Introduction

Golo [1] [2] est un langage dynamique construit autour de l'instruction "invokedynamic" de la machine virtuelle Java. Originellement développé au laboratoire CITI de l'INSA de Lyon par le groupe Dynamid, Golo est open-source et dédié à la réalisation d'applications dynamiques.

L'objectif de mon stage est de pouvoir augmenter la confiance que l'on peut avoir en un programme Golo. L'approche retenue est d'utiliser l'outils Why3 [3] développé par l'équipe Toccata de l'INRIA/LRI/CNRS, originellement conçu pour la vérification de programmes en C, Java et Ada. En s'appuyant sur le travail fait par le plugin Krakatoa [4] pour analyser du code Java avec cet outil, une traduction du code Golo vers du code WhyML [5] exécutable a été implémentée dans le compilateur Golo. Cette implémentation vise à poser une base pour pouvoir analyser statiquement du code Golo avec Why3, ainsi que pour faire des vérifications statiques lors de la traduction. La spécification dans le code Golo devait originellement se faire dans un langage inspiré de JML [6], cependant l'approche implémentée permet au programmeur de spécifier le code Golo dans le langage de spécification de Why3 directement. Le détail de la méthode d'implémentation, des fonctionnalités supportées ainsi que des limitations sera abordé dans une partie ultérieure.

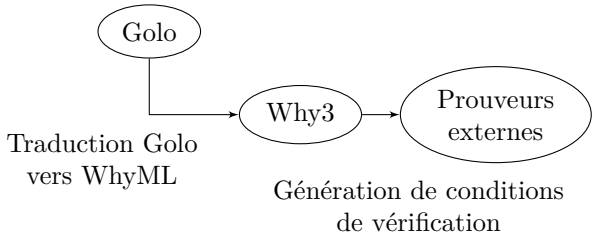
Un travail précédent sur le même sujet réalisé par Bertrand Cayeux montre qu'il est possible de modéliser les objets dynamiques de Golo en Java. Il a étudié la possibilité de traduire Golo en Java pour prendre avantage du plugin Krakatoa pour l'analyse statique en Java, comme montré sur la figure 1.

FIGURE 1 – Solution proposée par Bertrand Cayeux



Cela peut inspirer une traduction similaire directement en WhyML par la suite, cependant ce n'est pas abordé dans mon travail qui vise à mettre en place les briques de base pour traduire Golo en WhyML pour faire de l'analyse statique, comme montré sur la figure 2.

FIGURE 2 – Solution proposée pour ce stage



Dans un premier temps, je vais présenter un état de l'art de la vérification de programmes, et plus spécifiquement de la vérification statique.

3 État de l'art

La vérification de programmes a pour objectif de valider que le comportement d'un programme est et sera celui attendu. Il existe différents langages de spécification, avec chacun ses particularités. Ainsi, les automates (ensemble d'états et de transitions possibles) permettent d'exprimer une séquentialité d'actions et un ensemble d'état atteignables, les spécifications logiques permettent de décrire la forme que doit avoir un paramètre ou un résultat de calcul, les algèbres de processus permettent de mettre l'accent sur les communications et l'entrelacement des actions dans les systèmes concurrents ou distribués. Par ailleurs, les spécifications peuvent être placées dans des fichiers à part entière ou bien liées fortement à un code, comme c'est le cas des langage de spécification par annotation. Dans ce travail, nous verrons que c'est cette dernière catégorie qui nous intéressera plus particulièrement.

Il y a ensuite trois méthodes générales pour valider un programme et sa spécification, et toutes reposent sur la comparaison du comportement du programme au référentiel attendu défini au préalable :

- Tests : la vérification est opérée par l'exécution de cas spécifiques et de portions de code.
- Analyse dynamique : la vérification est opérée lors de l'exécution du programme dans un cas normal d'exécution. L'analyse dynamique est parfois combinée à l'analyse statique pour pouvoir obtenir une information plus précise sur l'origine d'une erreur de vérification [7].
- Analyse statique : les vérifications sont effectuées sur le programme sans exécution. L'une des manières de faire consiste à réduire la combinaison du programme et de son modèle à un ensemble de conditions de vérifications : après analyse, si ces conditions sont validées alors le programme est certifié comme valide, cependant un échec de vérification ne permet pas de différencier entre une erreur du programme et un manque d'information pour résoudre l'analyse.

Une différence significative parmi ces méthodes réside dans le fait que les tests et l'analyse dynamique nécessitent l'exécution de tout ou partie du pro-

gramme, alors que l'analyse statique permet de garantir le respect d'une spécification pour toute exécution, et avant toute exécution.

Pouvoir garantir un fonctionnement avant exécution possède de nombreux avantages, notamment dans des environnements ou tester une application n'est pas sans danger (comme dans l'aéronautique par exemple). Cette approche est également complémentaire aux tests ou à l'analyse dynamique, et c'est pourquoi le choix de travailler sur l'analyse statique a été fait.

Je vais maintenant détailler le fonctionnement de la vérification statique de programmes, et étudier les solutions existantes pour intégrer ces méthodes à des langages orientés objets de haut niveau.

3.1 Vérification statique de programmes

Je m'intéresse ici à l'analyse statique de programmes en général, et les différents outils qui la rendent possible. Je présente d'abord brièvement le fonctionnement des prouveurs SMT, puis les outils qui les utilisent, avant de finir sur la spécification de programmes, nécessaire pour pouvoir analyser statiquement des programmes avec les outils présentés.

3.1.1 Preuve SMT

Les prouveurs SMT (Satisfiabilité Modulo Théories) tels que Alt-Ergo [8] sont le bloc de base de l'analyse statique. Leur objectif est de résoudre la satisfiabilité de formules logiques (que l'on appelle ici conditions de vérifications), en fonction de prédicats énoncés au préalable. A ce jour, les prouveurs sont limités à la logique de premier ordre, ce qui aura des répercussions sur certaines limitations de l'analyse statique présentées plus loin. Notamment, cela implique que tout problème du second ordre doit être ramené au premier ordre pour pouvoir être résolu avec un prouveur SMT.

Différents prouveurs SMT ont différentes capacités de résolution (méthode naïve, apprentissage de clauses par conflit, analyse de conflits, retour en arrière). Cela implique que certaines formules peuvent être résolues par un prouveur SMT, alors que d'autres non. La conclusion directe que l'on peut en tirer est qu'utiliser différents prouveurs SMT peut donc améliorer le nombre de conditions de vérifications prouvables.

Bien que les prouveurs SMT puissent être utilisés manuellement pour vérifier des formules logiques, ils sont principalement utilisés avec des outils qui génèrent des conditions de vérifications à partir d'un programme et d'une spécification [9]. Les conditions de vérification générées permettent d'assurer la logique suivante :

- Si la condition de vérification est vraie, alors le programme et la spécification sont vrais.
- Si la condition de vérification n'est pas vraie, alors on ne peut pas conclure sur la justesse du programme et de sa spécification.

Toujours concernant la preuve SMT, on peut également noter un effort pour automatiser certaines tâches : la génération d'invariants automatiquement par exemple, pour décharger le programmeur de cette tâche parfois très complexe

[10]. Je m'intéresse de plus près aux outils qui permettent la génération de conditions de vérifications automatiquement dans la partie suivante.

3.1.2 Vérificateurs de programmes

Je présente ici deux vérificateurs de programmes, qui sont tous deux basés sur la traduction d'un langage intermédiaire vers des conditions de vérifications pour prouveurs SMT. Nous reviendrons sur l'utilisation d'un langage intermédiaire dans la partie suivante.

Le premier outil est Why3 [3], développé par l'équipe Toccata de l'INRIA/LRI/CNRS, originellement conçu pour la vérification de programmes en C, Java et Ada. Le second outil est Boogie [11], développé par Microsoft Research, originellement conçu pour la vérification de programmes Spec#.

Ces outils ont en premier lieu de nombreux points communs :

— Ils ont pour objectif principal d'automatiser la génération de conditions de vérification pour des prouveurs SMT.
— Ils se basent sur l'utilisation d'un langage intermédiaire, ainsi qu'un langage de spécification (WhyML pour Why3, et Boogie2 pour Boogie).
— Ils utilisent une modélisation explicite des structures et de la mémoire. L'utilisateur peut ainsi utiliser sa propre modélisation si besoin.
— Ils travaillent en logique de premier ordre.

Cependant, ils sont également pensés différemment, notamment sur deux aspects que je considère important pour le choix d'un outil pour ce stage : Boogie utilise Z3 comme prouveur SMT, alors que Why3 est développé autour du support d'un grand nombre de prouveurs SMT, via un système de drivers. De plus, Boogie a été développé pour C# originellement, alors que Why3 pour C/Java. Sachant que Golo produit du bytecode qui s'exécute sur la machine virtuelle Java (comme Java), Why3 semble être un choix plus judicieux au premier abord.

Ces outils reposant tous sur la traduction d'un programme et d'une spécification dans un langage intermédiaire, nous allons d'abord étudier brièvement un langage de spécification, nécessaire pour pouvoir effectuer des vérification plus poussées d'un programme par analyse statique.

3.1.3 Spécification JML

Dans ce travail, je m'intéresse à JML (Java Modelling Language) [6], une interface de spécification de comportement pour Java .

L'objectif principal de JML est de pouvoir masquer des modèles mathématiques derrière des classes facilement compréhensibles par tout programmeur Java. Le principe de base de ce langage de spécification est le "design par contrat" : un tuple (obligations, bénéfices) peut s'exprimer par un tuple (pré-conditions, post-conditions). Ce langage utilise la logique de Hoare, les pré-conditions et post-conditions ainsi que les invariants, et permet l'ajout de quantifieurs, de variables de spécification pures (qui n'ont pas d'impact sur le déroulement du programme), et de "frame conditions" (définit un comportement

normal et un comportement exceptionnel). JML peut être utilisé pour générer des docs améliorées, générer des tests unitaires, où faire de la vérification statique.

Pour pouvoir effectuer de l'analyse statique en Golo, il est nécessaire de se munir d'un langage de spécification. Golo étant similaire à Java, une variante de JML pour spécifier le code Golo semble être une approche logique. Cependant, mon implémentation ne se concentre pas sur cet aspect et utilise directement le langage de spécification de Why3 dans le code Golo.

3.2 Analyse statique - en pratique

Dans cette partie, je me concentre sur les langages intermédiaires des vérificateurs, ainsi que la traduction de langages. Il existe de nombreux exemples de traducteurs de langages, tel que Spoon [12], mais je me concentre ici uniquement sur ce qui touche aux langages intermédiaires pour l'analyse statique.

3.2.1 Langages intermédiaires

Je m'intéresse dans cette partie aux langages intermédiaires, et plus particulièrement ceux utilisés avec les outils de vérification que sont Why3 et Boogie.

L'avantage d'un langage intermédiaire pour un outil d'analyse statique est qu'il donne accès à toutes les fonctionnalités de cet outil, tout en offrant un langage de programmation et de spécification de haut niveau. Les trois langages présentés ici ont pour point commun de supporter quelques blocs de base pour écrire un programme : constantes, variables mutables, code "ghost" (spécification pure), séquences, boucles, fonctions, pré/post-conditions, invariants.

Boogie2 (anciennement BoogiePL) est le langage utilisé dans l'outil Boogie [11]. Il se démarque en étant un langage typé dont le typage n'est pas forcé ("any" peut être utilisé partout). Boogie2 supporte des procédures, mais pas de méthodes. Il n'y a également pas de heap pour les objets, pas de classes ni d'interfaces, les effets de bords ne sont pas permis, ni les appels par référence. Il n'y a pas d'exceptions ni de contrôle de flow structuré. Tous ces éléments manquants vont largement contribuer au choix de ne pas utiliser Boogie2 comme langage intermédiaire par la suite, étant donné que Golo les utilisent tous (cela sera discuté dans la section suivante).

Dafny [13] est un autre langage intermédiaire, qui lui est pensé pour supporter la programmation orientée objet, avec cependant des limitations fortes, tel que l'absence d'héritage. Le langage est typé, et nécessite au programmeur de déclarer explicitement quelles parties du programme ont le droit de changer (contrainte forte). Ce langage intermédiaire est traduit en Boogie2 pour l'analyse dans l'outil Boogie.

Enfin, WhyML [5] est le langage intermédiaire de l'outil Why3. Le langage est typé, cependant propose également de l'inférence de type. Il ne possède pas de modèle mémoire, et supporte uniquement les alias statiques. Il n'y a également pas d'objets, cependant ils peuvent être modélisés avec des structures de la librairie standard.

Les langages intermédiaires peuvent être utilisés directement pour prouver des algorithmes ou des théorèmes (bénéficier de la puissance des prouveurs SMT et en même temps de langages de plus haut niveau), cependant ils sont également beaucoup utilisés comme langage de traduction cible. Nous allons regarder de plus près la traduction vers un langage intermédiaire dans la partie suivante.

3.2.2 Traduction de langages

Dans cette section, je m'intéresse à la traduction de langages vers un langage intermédiaire pour l'analyse statique. Je montre l'utilité de traduire vers un langage intermédiaire, bien qu'il y ait des limitations qui doivent être considérées.

Je considère le travail fait pour traduire Dafny vers Boogie, Dafny étant construit autour de cette traduction [13]. La transformation cherche à garantir qu'un programme Boogie2 prouvé correct implique que le programme Dafny d'origine est correct, cependant la garantie inverse n'est pas vraie : un programme Boogie2 qui ne vérifie pas ne permet pas de savoir à coup sûr que le programme Dafny est invérifiable. C'est une limitation assez importante : lorsqu'on "empile" les intermédiaires entre le langage source et le langage de destination, cela multiplie les possibilités qu'un élément non supporté dans une traduction d'un langage vers un autre apparaisse.

Un effort a également été fait pour traduire Boogie2 vers WhyML [14] : l'idée étant de profiter de l'utilisation importante de Boogie dans le domaine de la vérification, tout en bénéficiant des avantages de Why3 permettant l'interfaçage avec de nombreux prouveurs externes. Un second avantage cité réside dans le fait que le code WhyML peut être exécuté. Il y a cependant des limitations dues au fait que le système de typage de WhyML est plus restrictif que celui de Boogie2, tous les programmes ne peuvent pas être traduit avec succès.

Finalement, le plugin Krakatoa pour Why3 [4] permet la traduction de programmes Java annotés en JML vers des programmes WhyML. La méthode propose notamment une définition d'un modèle mémoire dans le code, ainsi qu'une modélisation des classes, deux éléments qui manquaient à WhyML. Une des limitations du plugin réside dans le fait que le code WhyML produit est destiné à de la vérification pure, et ne peut pas être exécuté.

Ces éléments viennent confirmer le choix de traduire directement de Golo vers WhyML, tout en générant du code WhyML exécutable.

3.3 Bilan de cet état de l'art

Cet état de l'art me permet de justifier les choix d'implémentation d'une solution pour augmenter la confiance en du code Golo. L'approche choisie se base sur l'outil d'analyse statique Why3, en utilisant WhyML comme langage cible d'une traduction directe depuis Golo. La spécification est faite directement avec le langage de spécification de Why3 dans le code Golo. L'analyse statique peut ensuite être effectuée dans Why3 avec des prouveurs externes. Quelques éléments de vérification sont présents dans le traducteur directement (syntaxe

Golo, existence de fonction). Le code WhyML produit est exécutable, et produit les même résultats que le code Golo.

Dans la partie suivante, je m'intéresse plus en détail au langage Golo et à ses spécificités. Je détaillerai ensuite la traduction de certains éléments de Golo vers WhyML, avant d'expliquer des éléments clés de mon implémentation dans le compilateur Golo.

4 Golo

Golo [1] [2] est un langage simple pour la machine virtuelle Java (JVM), créé pour la recherche et l'éducation. Le trait principal de ce langage est le fait qu'il est construit autour de l'instruction invokedynamic de la JVM 7+. Invokedynamic permet notamment de customiser à l'exécution un lien entre un site d'appel et une implémentation de méthode. Cette instruction simplifie grandement la création d'un langage typé dynamiquement pour la JVM.

Golo est similaire à Java sur beaucoup d'aspects, et supporte même l'API Java standard : les librairies Java peuvent être utilisées en Golo. Les éléments tels que les boucles et le contrôle de flux sont similaires également. Cependant, le langage vise à offrir une dynamicité qui n'est pas atteignable facilement en Java. Pour cela, Golo offre notamment :

— L'augmentation de classes : permet d'ajouter des méthodes à une classe dynamiquement.
— Les objets dynamiques : des objets dont les propriétés et les méthodes peuvent être définis pour chaque instance.
— Un typage dynamique

Je montre dans la partie suivante comment certains éléments de Golo peuvent être traduits en WhyML.

5 Éléments de traduction Golo vers WhyML

Je montre ici comment peuvent être traduits des éléments de Golo en WhyML, un langage proche syntaxiquement de OCaml, pour l'outil Why3. J'ajoute la contrainte que le code WhyML soit exécutable, ce qui permet de vérifier facilement sur un programme simple que le comportement est le même qu'en Golo, et permet d'identifier de possibles erreurs qui apparaissent uniquement à l'exécution, tout cela sans complexifier la traduction de façon importante. Je me concentre sur les fonctionnalités générales des éléments traduits, l'objectif étant de pouvoir poser les bases pour la traduction d'un programme simple. Les éléments traduits ne supportent pas toutes les fonctionnalités Golo, l'objectif étant de pouvoir traduire un nombre suffisant d'éléments pour la traduction automatique de programmes simples (plus de fonctionnalités peuvent être ajoutées par la suite une fois les bases posées).

L'ensemble des éléments pour lesquels une traduction est visée est composé de : module, fonction, variables, opérateurs, spécification, return. Ces éléments

ne sont parfois traduits que partiellement, comme précisé dans les parties suivantes qui montrent comment traduire des portions de code Golo en WhyML. Dans la section **??**, je présenterai l'outil développé dans le cadre de ce travail, qui implémente les éléments présentés ici.

5.1 Module et fonction

Les fonctions Golo font toujours parties d'un module, élément de base de tout programme. Le code 1 montre la définition d'un module et d'une fonction simple, renvoyant son premier paramètre. Une fonction Golo est définie avec le mot clé "function", et les paramètres sont listés entre deux symboles "|".

```
1 module fonction.Main
2
3 function main = |a, b| {
4         return a
5 }
```

Code 1: Module Golo

Le code WhyML correspondant 2 est très similaire. Une déclaration de fonction peut se faire avec le mot clé "let", et les paramètres apparaissent entre parenthèses. Notons qu'en WhyML le résultat de la dernière instruction d'une fonction est la valeur retournée implicitement.

En raison de limitations de la ligne de commande Why3, ce code WhyML n'est pas directement exécutable car il n'est pas possible de lancer un programme par une fonction attendant des paramètres.

```
1 module FonctionMain
2     let main (a)(b)
3         = a
4 end
```

Code 2: Module WhyML

Je considère tout au long de ce travail que nous nous intéressons à un unique module Golo (dans un seul fichier). Je montre dans la prochaine partie comment peut être traduit un appel de fonction, et ne traite donc pas le cas des fonctions appartenant à un autre module.

5.2 Appel de fonction

Un appel de fonction en Golo peut se faire de plusieurs façon :
— Appel de fonction sans paramètre
— Appel de fonction avec paramètres

— Appel de fonction avec arité variable
— Appel de fonction avec des paramètres nommés : permet d'utiliser le nom des paramètres explicitement lors de l'appel d'une fonction (permet ainsi de ré-ordonner les paramètres à l'appel).

Je m'intéresse ici à montrer comment implémenter en WhyML un appel de fonction avec ou sans paramètres pour pouvoir supporter une grande quantité de programmes simples. WhyML ne permettant pas les appels de fonctions à arité variable, il faut adapter les appels de fonction lors de la traduction pour que l'arité de l'appel dans le code WhyML corresponde à l'arité de la fonction appelée.

L'exemple Golo du code 3 montre comment une fonction peut être appelée avec un paramètre. L'équivalent en WhyML dans le code 4 affiche une syntaxe similaire.

```
function test = |a| {
    return a
}

function appel_test = {
    test(1) # renvoie 1
}
```

Code 3: Appel de fonction Golo

```
let test (a) =
    (a)

let appel_test =
    test (1) (* renvoie 1 *)
```

Code 4: Appel de fonction WhyML

Les appels de fonctions en Golo peuvent générer des crashs au moment de l'exécution si une fonction appelée n'existe pas. Je m'intéresse donc ici à vérifier statiquement l'existence d'une fonction lors d'un appel. Cette vérification est opérée directement lors de la traduction, et est détaillée dans la description de mon implémentation.

Une difficulté importante de la traduction des appels de fonctions vient du fait que les fonctions en WhyML on besoins d'être définies avant d'être utilisées dans le code, alors que ce n'est pas le cas en Golo. Il est possible d'utiliser des prototypes en WhyML pour les cas ou les fonctions sont définies après être appelées, cependant il est nécessaire de préciser les types des paramètres ainsi que le type de retour de la fonction. Je montre dans le code 5 un exemple de prototype.

```
val test (a: int): int  (* prototype *)

let appel_test =
    test (1) (* renvoie 1 *)

let test (a) =
    (a)
```

Code 5: Prototype WhyML

Un autre élément nécessaire à tout programme Golo simple est la déclaration de constantes et variables. Je m'intéresse à leur traduction en WhyML dans la partie suivante.

5.3 Déclarations

Golo fait la différence entre les variables et les constantes. Le compilateur possède déjà un certain nombre de vérification statiques à ce niveau, tel que la vérification qu'une constante n'est pas modifiée : il n'est donc pas nécessaire de ré-implémenter ces vérifications. Un autre élément clé est le fait que toute variable ou constante déclarée dans un programme Golo doit être initialisée : une déclaration va toujours de pair avec une assignation, dont la traduction est détaillée dans la partie suivante.

Un exemple de déclaration d'une constante et d'une variable est montré dans le code 6 .

```
let myCons = 1
var myVar = 2
```

Code 6: Déclarations Golo

L'équivalent à ce code Golo peut être écrit en WhyML comme montré dans le code 7 .

```
import ref.Ref
(* ... *)
let myCons = 1 in
let myVar = ref 2 in
```

Code 7: Déclarations WhyML

Les variables WhyML sont en réalité des références (de la librairie standard ref.Ref) avec un élément mutable. Cela implique que l'assignation d'une nouvelle

valeur à une variable se fait différemment que l'assignation lors de la déclaration, détaillé dans la prochaine partie.

5.4 Assignation sur une variable

Je décris ici comment l'assignation d'une nouvelle valeur à une variable est faite. Nous avons vu dans la partie précédente l'assignation lors de la déclaration, je ne reviens donc pas déçu.

Je montre dans le code 8 comment assigner une nouvelle valeur à une référence WhyML. Il faut noter ici une limitation importante de WhyML : lors de la déclaration d'une référence, un type est défini (implicitement), il n'est donc pas possible d'assigner un élément d'un type différent à cette variable. Golo étant dynamiquement typé et permettant ce comportement, je propose une alternative pour pouvoir traduire un code équivalent en WhyML : re-déclarer la variable avec le nouveau type, comme montré dans le code 9 . Il faut cependant faire attention à la portée de la nouvelle déclaration qui peut être moindre que la déclaration originale, ainsi qu'aux cas ou des effets de bords disparaîtraient à cause de la redéfinition.

```
1 import ref.Ref
2 (* ... *)
3 let myVar = ref 2 in
4 myVar := 5 (* myVar est maintenant 5 *)
```

Code 8: Assignation WhyML

```
1 import ref.Ref
2 (* ... *)
3 let myVar = ref (1, 2) in  (* myVar est un tuple d'int *)
4 let myVar = ref 5 in (* myVar est maintenant un int *)
```

Code 9: Assignation WhyML avec type différent

Je montre dans la partie suivante comment peuvent être utilisées des variables et constantes (accès en lecture).

5.5 Consultation de constantes et variables

En Golo, la lecture d'une constante ou d'une variable est effectuée de la même manière, il suffit d'utiliser le nom de cette constante/variable pour y accéder. En WhyML cependant, les variables étant construites avec des références, leur méthode d'accès est différente des constantes pour lequel utiliser le nom (comme en Golo) est suffisant.

Je montre dans le code 10 comment accéder à la valeur d'une variable en WhyML. Ce constat d'un accès différent entre constantes et variable a un impact lors de l'implémentation du traducteur, car un accès ne se traduira pas de la même façon en fonction du type d'élément accédé. Je détaille cette problématique dans la description de mon implémentation.

```
1 import ref.Ref
2 (* ... *)
3 let myVar = ref 1 in
4     !myVar (* renvoie la valeur 1 *)
```

Code 10: Accès à une variable en WhyML

Dans la partie suivante je montre comment effectuer des opérations simples sur ces éléments, avec le support de quelques opérateurs.

5.6 Opérateurs

Parmi les fonctionnalités de base du langage Golo figurent les opérateurs. Ils sont au nombre de dix [1], et je vise ici à traduire les plus communs (opérations et comparaisons numériques) pour poser une première base, la traduction des autres opérateurs pouvant être étudiée ultérieurement.

Le problème de traduction des opérateurs est en réalité plus complexe qu'il n'y parait au premier abord en raison du typage strict de WhyML, et du fait qu'il n'est pas possible de surcharger une fonction en WhyML. Ainsi, l'opérateur "+" en Golo peut effectuer une addition sur des nombres et des chaînes de caractères, cependant en WhyML chaque opération sur des types différents doit être définie dans des fonctions différentes. J'ai donc décidé de restreindre le problème aux opérations sur les nombres entiers (int) dans un premier temps, pour simplifier le traitement de la traduction.

La libraire standard WhyML propose une implémentation des opérateurs de base dans différentes théories. Certaines de ces théories viennent ajouter des vérifications statiques, comme la division définie dans "mach.int" qui ajoute une pré-condition sur le dénominateur (doit être non-nul). L'utilisation de ces théories permet déjà d'ajouter des vérifications statiques sur du code sans spécification. Le code 11 montre un exemple de code WhyML d'une division arithmétique d'entiers.

Dans le cas d'une fonction effectuant une division, et avec une définition de la division ajoutant une pré-condition sur le dénominateur, il faut permettre au programmeur d'ajouter de la spécification au programme pour que Why3 puisse valider le code. Je détaille dans la partie suivante comment la spécification est traduite entre Golo et WhyML.

1. addition numérique et de chaînes de caractères, soustraction numérique, multiplication numérique et de chaînes de caractères, division numérique, modulo numérique, comparaison numérique et d'objets, comparaison de qualité de références, opérateur booléen, vérification de type, valeur si null

```
import mach.int
(* ... *)
let a = 4 in
let b = 2 in
let c = Int.(/) a b in (* c = 2 *)
```

Code 11: Division WhyML

5.7 Spécification de programme

Pour pouvoir spécifier le code Golo, j'ai choisi d'utiliser directement la spécification de l'outil Why3 en Golo, la syntaxe étant claire et simple à comprendre. Il n'y a ainsi pas de traduction à faire, il s'agit simplement de pouvoir parser la spécification (l'implémentation dans le parseur est détaillée dans une partie suivante). Je choisi d'utiliser des balises "spec/" et "/spec" pour placer la spécification, ce qui simplifie la lecture et évite le mélange avec le code Golo exécutable. Le code 12 montre un exemple d'une fonction avec de la spécification.

```
function max = |a, b| spec/ ensures{ (result >= a) /\
                             (result >=b) /\
                             (forall z:int. z>=a /\ z>=b -> z >= result) }
                      /spec {
    if(a >= b) {
        return (a)
    } else {
        return (b)
    }
}
```

Code 12: Spécification dans Golo

Il faut noter ici que la spécification Why3 fait correspondre directement la valeur de retour d'un programme à la variable de spécification "result". Il faut donc éviter d'utiliser des variables ou constantes appelées "result" dans le programme, ce qui pourrait créer des conflits lors de l'analyse statique (une alternative étant de renommer lors de la traduction).

Un élément manquant pour pouvoir traduire des programmes simples en Golo est une gestion de "return", c'est à dire pouvoir retourner une valeur et quitter une fonction avant la fin de celle-ci. Je détaille comment traduire cela en WhyML dans la partie suivante.

5.8 Gestion du "return"

Une fonction Golo renvoie toujours une valeur. Si une fonction ne renvoie rien explicitement, alors la valeur "null" est renvoyée. De plus, comme dans la plupart

des langages de programmation orientés objets, il est possible de renvoyer une valeur avant la fin d'une fonction, sans exécuter le reste. WhyML ne possède pas de "return", et pour pouvoir définir un comportement identique j'utilise le mécanisme d'exceptions, ainsi que la variable "return" qui est utilisée par défaut pour définir la valeur de retour d'une fonction.

Je rappelle ici que je prends le cas particulier d'une fonction qui ne peut renvoyer que des entiers. Pour pouvoir traduire les fonctions qui ne renvoient pas de valeur explicitement, je défini également une nouvelle constante de type null. Un exemple de fonction avec une valeur de retour est montré dans le code 13 (cette fonction renvoie null si a >= 10, ou a sinon).

```
1  exception Return ()
2  constant null : int
3
4  let lowerThanTen (a) =
5    let return = ref 0 in try begin (
6      if (Int.(≥) a 10) then (
7        return := null; (* attribue la valeur null à return *)
8        raise Return ) (* raise l'exception *)
9      else (
10       return := a;
11       raise Return )
12   ) end with Return → !return   (* exception renvoie valeur de
       return *)
```

Code 13: Return WhyML

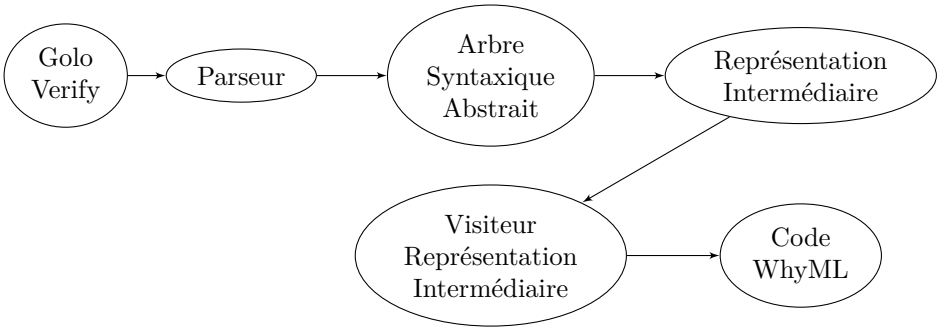
Les éléments de base de la traduction de Golo vers WhyML étant posés, je décris dans la partie suivante l'implémentation qui a été faite dans le compilateur Golo.

6 Implémentation

L'implémentation de la traduction automatique de code Golo vers du code WhyML est faite directement dans le compilateur Golo, qui est écrit en Java. La version utilisée pour le développement est le snapshot 3.2.0-M1.

La traduction de code Golo pour la vérification statique dans l'outil Why3 étant un nouvel élément du compilateur, une nouvelle commande appelée "golo verify" lui est associé. Dans la figure 3 apparaissent les éléments principaux de la fonction "golo verify" implémentée pendant ce stage.

FIGURE 3 – Éléments clés de Golo Verify



Le compilateur Golo utilise un générateur de parseur qui se base sur une définition de grammaire pour générer un arbre syntaxique abstrait. Une fois cet arbre généré, des visiteurs le parcourent pour effectuer quelques vérifications (par exemple, tester si une constante n'est pas ré-assignée), ainsi que pour désugariser le code. Cela produit une représentation intermédiaire qui représente les éléments Golo sous forme d'objets Java.

Mon implémentation est concentrée en grande partie sur la représentation intermédiaire, après l'appel à des visiteurs sur l'arbre abstrait. Travailler sur cette représentation intermédiaire a également pour avantage de respecter le langage Golo sans avoir à ré-implémenter la sémantique, contrairement à l'arbre syntaxique abstrait qui lui est issu du parseur et n'a pas été validé. Je décris dans les parties suivantes les éléments clés de l'implémentation réalisée, ainsi que leurs implications.

6.1 Grammaire JJTree et arbre syntaxique abstrait

Pour pouvoir ajouter un nouveau type de code Golo qui soit reconnu par le parseur, il faut modifier la définition de la grammaire Golo. Le compilateur utilise JJTree, un pré-processeur pour JavaCC qui génère un parseur à partir d'une définition de grammaire. Ce parseur produit ensuite un arbre syntaxique abstrait.

Dans un premier temps, pour reconnaître un bloc de spécification, un nouveau token SPEC est défini comme montré dans le code jjt 14 .

Ce token est utilisé pour récupérer directement la spécification dans un String, comme montré dans le code 15 .

Cet élément de spécification est appelé dans la définition d'une fonction, qui est actuellement le seul endroit ou l'on est autorisé à mettre de la spécification dans le code Golo. Pour pouvoir supporter de la spécification n'importe ou dans le code, il faudrait définir un nouveau noeud pour l'arbre abstrait ainsi que pour la représentation intermédiaire pour la spécification.

Dans les parties suivantes, je détaille l'implémentation de quelques éléments du visiteur de la représentation intermédiaire qui s'occupe de générer la traduc-

```
MORE :
{ < "spec/" > : WithinSpec }

<WithinSpec>
TOKEN :
{ < SPEC : "/spec" > : DEFAULT }

<WithinSpec>
MORE :
{ < ~[] > }
```

Code 14: Token SPEC

```
String Specification() #void:
{Token specToken;}
{specToken = <SPEC>
  {return specToken.image.substring(5,
              specToken.image.length()-5).trim();}
}
```

Code 15: Token SPEC

tion en WhyML.

6.2 Module

L'implémentation de la traduction d'un module est un bon exemple pour montrer comment le visiteur de la représentation intermédiaire fonctionne. La fonction "visitModule" est la première appelée et est donc le point d'entré de la traduction. Les éléments les plus importants de cette fonction sont montrés dans le code 16 .

"whyMLcode" est un ArrayList de String qui permet de stocker le code WhyML généré. La ligne 2 déclare le module en WhyML, et les lignes 4 à 9 ajoutent dans le code WhyML les imports de base pour les fonctions qui sont actuellement supportées, ainsi que les déclarations de l'exception Return et de la constante "null". Une amélioration nécessaire sur cette partie serait d'enlever les imports statiques, et de les ajouter lorsqu'ils sont nécessaires (par exemple, importer mach.int.Int uniquement si des opérations arithmétiques sont effectuées).

La ligne 11 appel le visiteur sur la totalité du contenu du module. Une fois la ligne 11 exécutée, le code du module est normalement entièrement traduit, et la ligne 12 vient marquer la fin du module. Le bloc try/catch écrit le code généré dans le fichier de destination. Finalement, un dernier test est effectué avant de quitter le visiteur à la ligne 18 : les définitions de fonctions ainsi que les appels de fonctions ont été enregistrés durant la visite, et ce test vérifie l'existence de toute fonction appelée. Une des limitations actuelles de cette implémentation est le fait que les fonctions intégrées de base ne sont pas gérées (cependant, leur traduction non plus, il est donc raisonnable d'afficher une erreur). Cette vérification fonctionne également lorsqu'une fonction est appelée avec une arité supérieure à celle de la fonction définie, étant donné que Golo rend cela possible.

Étant donné qu'un module est très souvent suivi de la définition d'une fonction, je présente dans la partie suivante l'implémentation de la traduction d'une fonction.

```
/* ... */
whyMLcode.add("module " + moduleName)
/* ... */
whyMLcode.add(space() + "use import int.Int");
whyMLcode.add(space() + "use import ref.Ref");
whyMLcode.add(space() + "use import mach.int.Int");
whyMLcode.add(space() + "type null");
whyMLcode.add(space() + "constant null : int");
whyMLcode.add(space() + "exception Return ()");

module.walk(this);
whyMLcode.add(space() +"end");
try {
  Files.write(Paths.get(destFile), whyMLcode, Charset.forName("UTF-8"));
} catch (IOException e) {
  e.printStackTrace();
}
testFunctionsCalledExist();
```

Code 16: Éléments de la fonction visitModule

6.3 Fonction

La traduction d'une fonction est un élément un peu plus complexe que la simple définition du module. Le code est donc présenté par morceau pour des raisons de clarté.

Les premiers éléments d'une fonction, décrits dans le code 17 montrent principalement l'ajout de la fonction à la liste des fonctions déclarées (ligne 1, 2), puis la déclaration de la fonction dans le code WhyML (ligne 3 et 5). La ligne 4 vient ajouter les nombres de lignes auxquelles cette fonction apparaît dans le fichier Golo pour les afficher dans l'outil Why3. On peut remarquer que l'implémentation est légèrement différente de la traduction d'une fonction définie dans la partie théorique précédente : l'utilisation de "let function = fun ->" est nécessaire pour pouvoir insérer les numéros de lignes.

```
functionsDefined.add(new functionNameArity(function.getName(),
                                           function.getArity()));
whyMLcode.add(space() + "let " + function.getName() + " ");
appendWhyMLLastString(getSourceCodeBlocksLines(function) + " = ");
whyMLcode.add(space() + "fun ");
```

Code 17: Définition de fonction

Le code 18 montre la traduction des paramètres de la fonction. Deux cas

sont possibles : si la fonction possède des paramètres, ceux-ci sont ajoutés entre parenthèses, sinon des parenthèses vides sont écrites. Cet ajout est nécessaire à WhyML, une fonction sans paramètre en Golo ne nécessitant pas de balises vides. La dernière ligne vient copier la spécification (si présente) depuis ce qui a été parsé directement dans le code WhyML.

```
1 for (String param : function.getParameterNames()) {
2   appendWhyMLLastString("( " + param + " )");
3 }
4 if (function.getParameterNames().isEmpty()){
5   appendWhyMLLastString("()");
6 }
7 whyMLcode.add(space()+function.getSpecification());
```

Code 18: Paramètres de la fonction

Le code 19 montre la déclaration des références locales, effectuées avant le corps de la fonction. Cette implémentation est nécessaire, les variables (références) en WhyML nécessitant d'être déclarées avant leur initialisation/utilisation. Il y a une limitation majeure cependant : la portée des variables est automatiquement toute la fonction. Ainsi, une variable qui en Golo ne serait visible qu'à l'intérieur d'une boucle est visible en WhyML dans la fonction entière avec cette implémentation. Des vérifications additionnelles sur la réutilisation des noms de variables ainsi que sur l'existence d'une variable sachant sa portée sont nécessaire pour rendre l'implémentation complètement correcte.

```
1  LinkedList<LocalReference> refList = function.returnOnlyReference(this);
2  for (LocalReference ref: refList) {
3    boolean isParam = false;
4    for (String param: function.getParameterNames()){
5      if (ref.getName().equals(param)) {
6        isParam = true;
7      }
8    }
9    if(!isParam && !ref.isConstant()){
10     whyMLcode.add(space() + "let " + ref.getName() + " = ref 0 in ");
11   }
12 }
```

Code 19: Déclaration des références locales

Finalement, le code 20 montre les derniers éléments de la traduction d'une fonction : la référence return est déclarée en ligne 1, avant de lancer la visite du corps de la fonction (et donc la génération de sa traduction). La ligne 3 vient fermer la fonction et ajouter l'exception qui gère l'appel à "return".

```
whyMLcode.add(space() + "let return = ref 0 in try begin");
function.walk(this);
whyMLcode.add(space() + "; end with Return -> !return end");
```

Code 20: Return et corps de la fonction

6.4 Constantes et variables

Dans la présentation théorique de la traduction, une difficulté autour de la gestion différente des constantes et variables en WhyML, comparé à Golo, avait été relevée. Je montre ici comment l'implémentation de la traduction permet de résoudre ces différences.

Dans un premier temps, lors d'une assignation, il suffit de gérer deux cas pour que la traduction génère du code WhyML correct. Ces cas sont montrés dans le code 21. Dans le cas ou la référence est définie comme constante dans la représentation intermédiaire, le code "let refName =" est généré avant de visiter ce qui est assigné avec la fonction "walk". Sinon, la référence est donc variable et l'opérateur " := " est utilisé avant de visiter l'élément assigné.

```
if (assignmentStatement.getLocalReference().isConstant()) {
  whyMLcode.add(space() + "let " + refName + " = ");
  assignmentStatement.walk(this);
  whyMLcode.add(space() + " in");
} else {
  whyMLcode.add(space() + "( "  + refName + " := ");
  assignmentStatement.walk(this);
  whyMLcode.add(space() + " );");
}
```

Code 21: Assignation de constantes et variables

D'une façon similaire lors de la lecture d'une constante ou variable, le code 22 montre le même type de procédé pour séparer les deux cas possibles : copier uniquement le nom de la constante dans le code WhyML, ou ajouter "!" devant s'il s'agit d'une variable.

```
if (reference.isConstant()) {
  whyMLcode.add(space() + "( " + referenceLookup.getName() + " ) ");
} else {
  whyMLcode.add(space() + "( !" + referenceLookup.getName() + " ) ");
}
```

Code 22: Lecture de constantes et variables

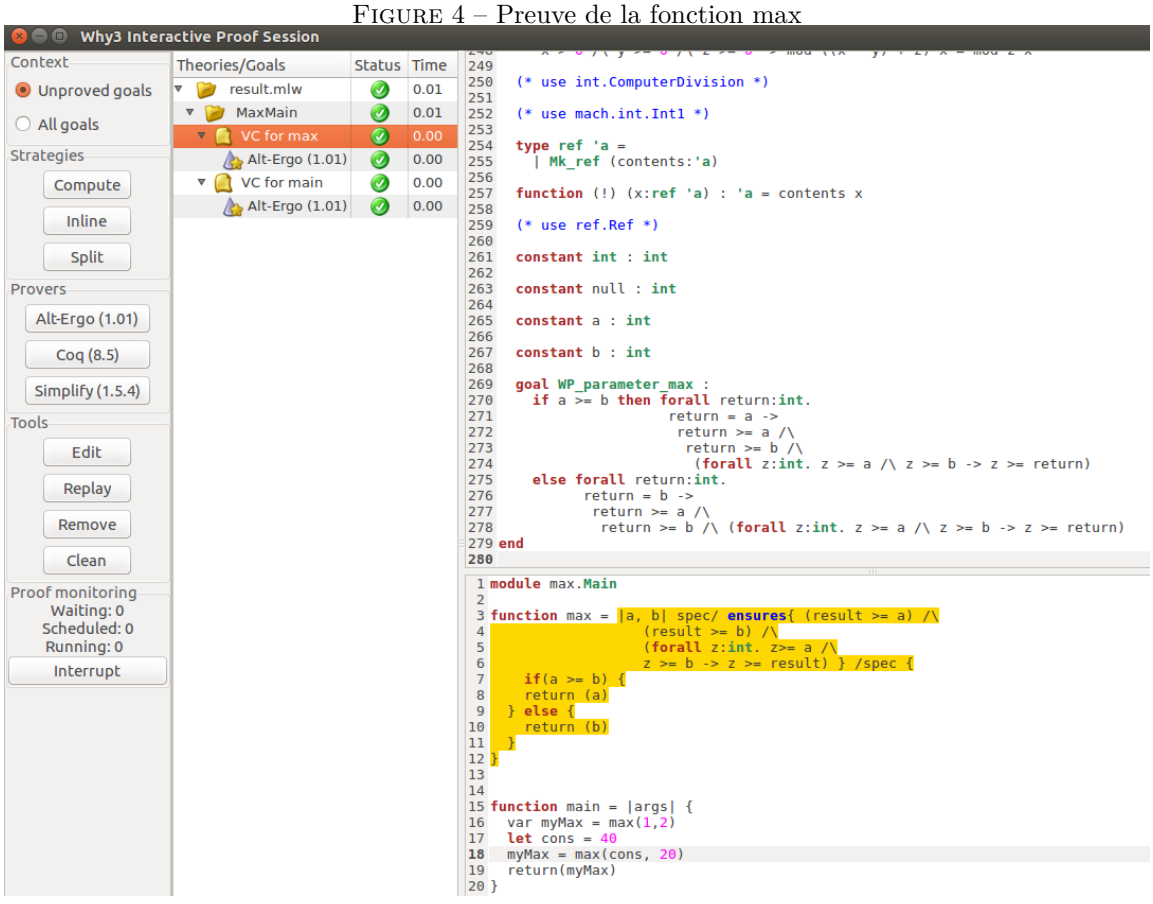
Les autres éléments sont traduits avec des méthodes similaires et ne sont donc pas détaillés dans ce rapport (le code source complet étant fourni). Je montre dans la prochaine partie quelques exemples de programmes Golo traduits en WhyML et analysés dans l'outil Why3.

7 Preuve dans Why3

Je montre dans cette partie des exemples de code Golo analysés dans l'outil Why3. L'objectif est de pouvoir identifier quelques applications rendues possible par ma contribution.

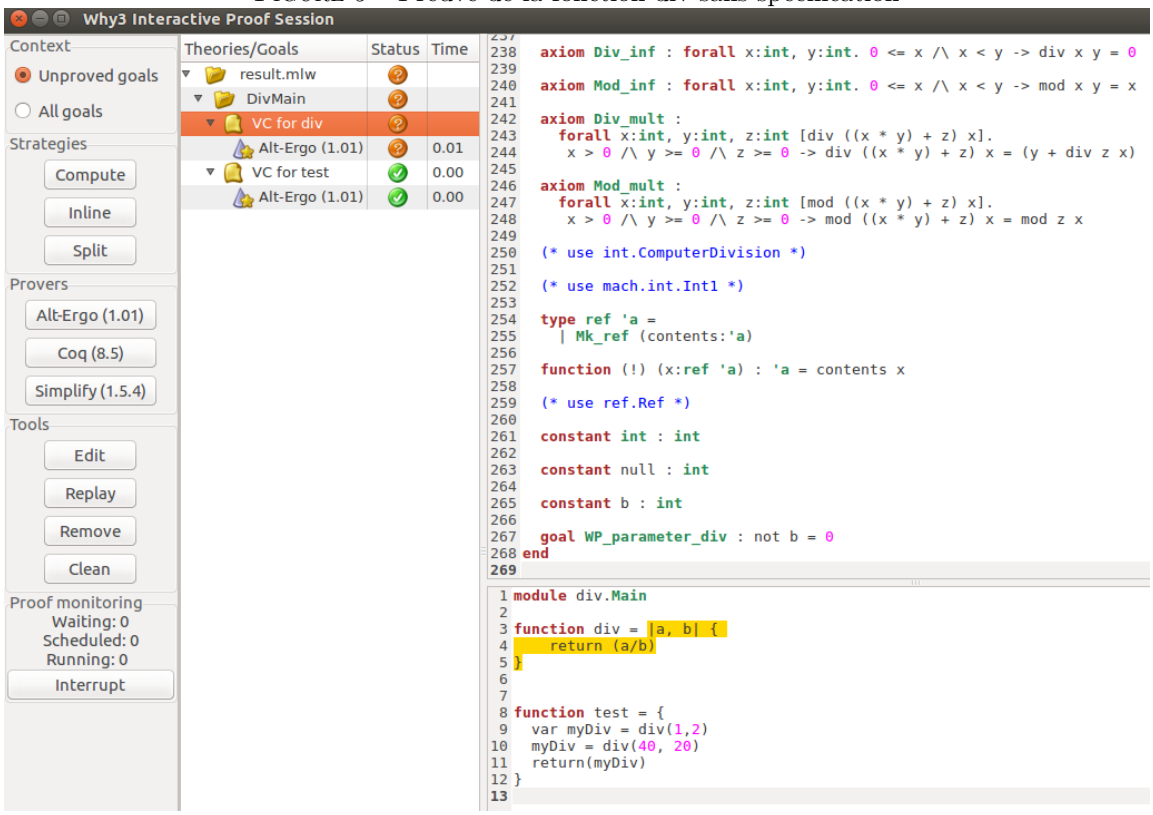
Le premier exemple montré sur la figure 4 est la définition d'une fonction "max". On peut lire dans le coin en bas à gauche le code Golo accompagné de la spécification de la fonction. Dans le coin supérieur droit apparaissent les conditions de vérification générées par Why3. Le prouveur Alt-Ergo a été lancé pour la fonction et a pu la valider.

FIGURE 4 – Preuve de la fonction max



Les trois exemples suivants sont basés sur la même fonction, effectuant la division de deux entiers. La figure 5 montre la fonction effectuant la division sans spécification. On peut voir que le preuve n'est pas validée par le prouveur. Cela est dû à la condition de vérification "not b = 0" de la fonction div, en effet, rien n'empêche d'appeler la fonction avec b=0 ce qui ferait crasher le programme.

FIGURE 5 – Preuve de la fonction div sans spécification

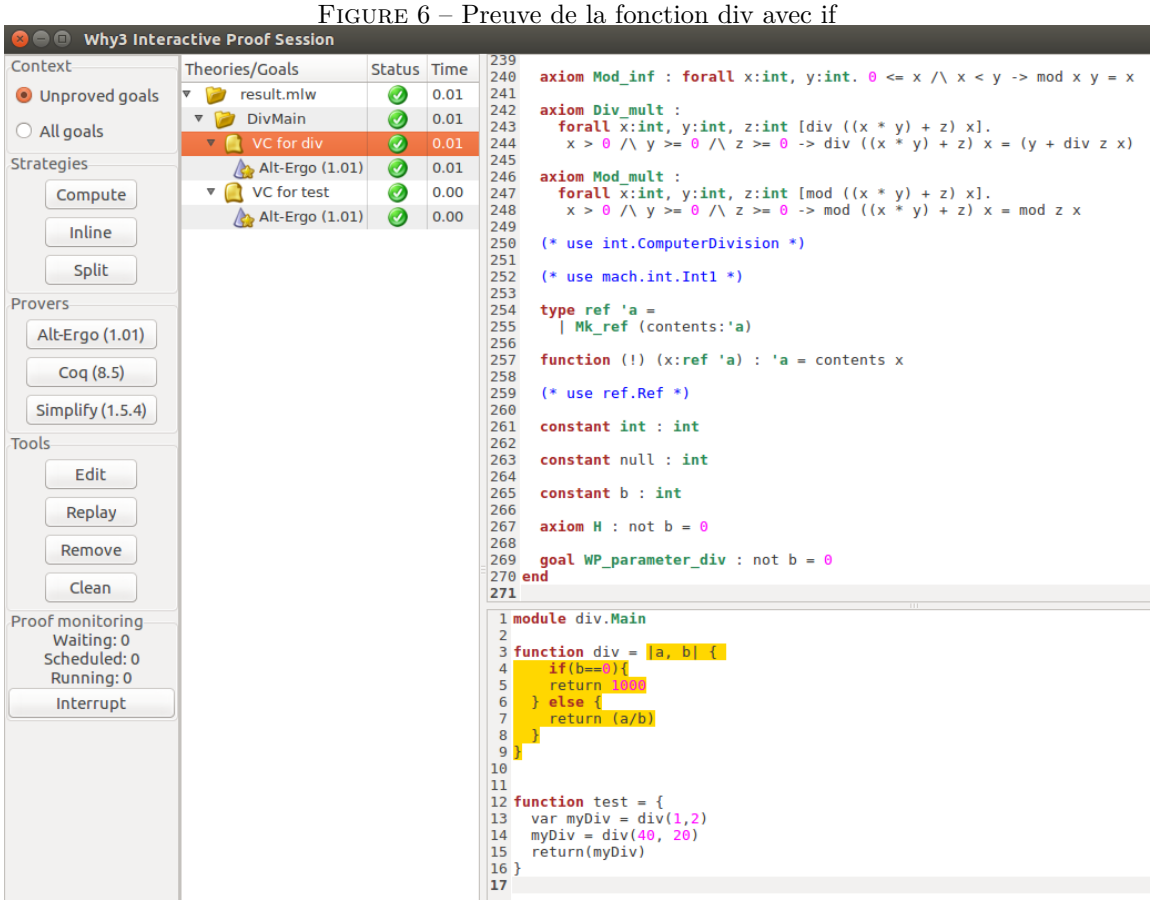


Il est attendu que si le programme prends en compte le cas particulier où b=0, alors la preuve doit être valide. On peut vérifier cela avec la figure 6 : cette fois la vérification n'échoue pas, le cas ou b=0 étant traité séparément.

Dans le cas où l'on souhaite conserver la fonction sans la modifier, il y a la possibilité de rajouter une ligne de spécification pour permettre la vérification. La figure 7 montre dans un premier temps qu'avec une pré-condition sur b, alors la fonction div passe la vérification. Cependant, on voit maintenant que la fonction test ne passe pas entièrement la vérification : la deuxième pré-condition n'est pas résolue. Ce comportement est attendu, la seconde ligne de la fonction appelant la fonction div avec le paramètre b=0, ce qui est désormais interdit par la spécification ajoutée.

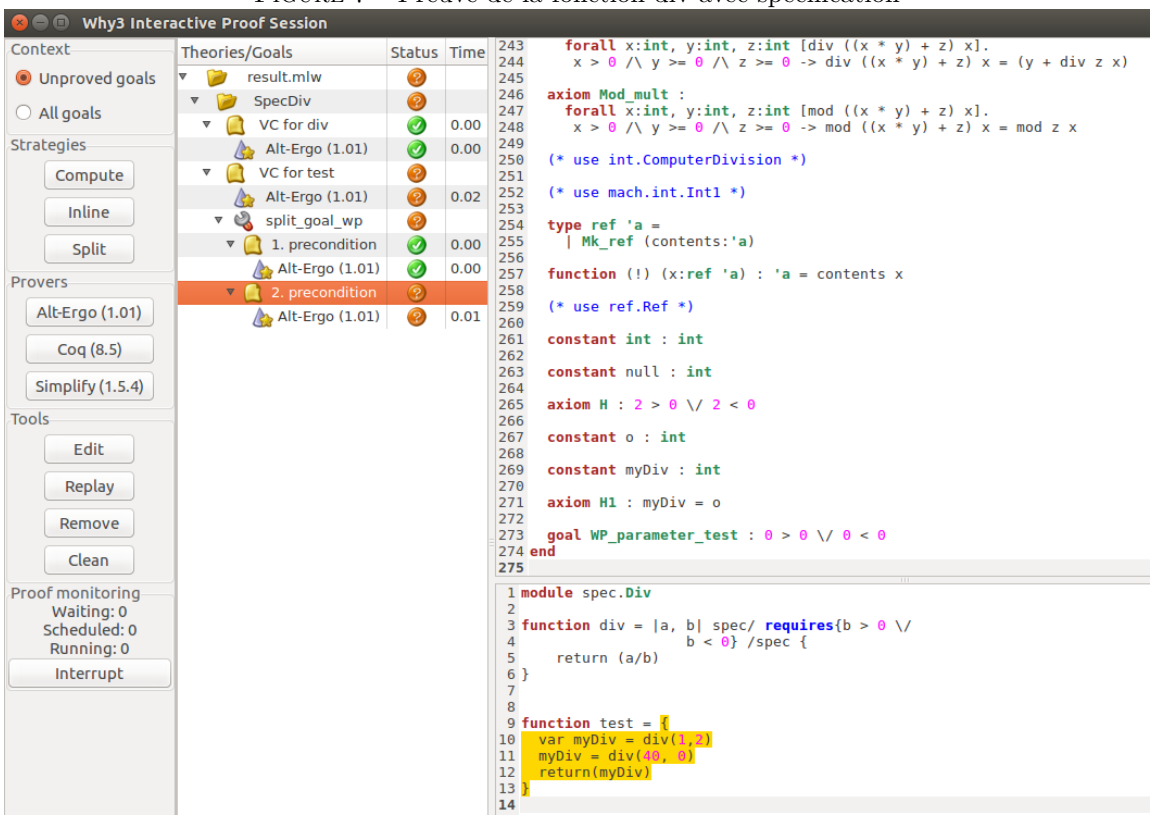
On peut donc voir que spécifier une fonction permet de vérifier que ses appels

FIGURE 6 – Preuve de la fonction div avec if



sont corrects et ne généreront pas de crash.

FIGURE 7 – Preuve de la fonction div avec spécification



8 Bilan

Pour répondre à la problématique d'ajout de confiance dans du code Golo, j'ai pris l'approche de traduire le code Golo en WhyML pour l'analyser statiquement dans l'outil Why3. J'ai détaillé dans ce rapport mes choix techniques, ainsi que ma contribution, principalement centrée autour de l'implémentation d'un traducteur automatique de code, intégré dans le compilateur Golo.

Parmi les fonctionnalités principales, on retrouve la vérification de l'existence des fonctions appelées dans le code, l'ajout de pré-conditions automatiquement (tel que dénominateur d'une division non-nul), et permis l'ajout de spécification de code. Cette implémentation est une première base qui permet d'être étendue pour supporter plus de fonctionnalités dans le futur. Parmi les limitations actuelles, on retrouve notamment : la limitation à un unique module Golo, les arité de fonctions devant être fixes, la limitation de la spécification aux fonctions, la surcharge des opérateurs, la portée des variables, ainsi que "result" étant un nom réservé.

Une des grandes difficultés qui doit être résolue pour pouvoir étendre la

portée de ce projet réside dans la gestion du typage : l'implémentation actuelle ne supporte que les entiers non bornés. Il est nécessaire, dans un premier temps, de définir une théorie pour utiliser des entiers bornés, puis de supporter d'autres types. Une piste de réflection pourrait être l'utilisation de structures génériques WhyML pour décrire tout élément avec une description du type : genericType{mutable type ; mutable var}. L'utilisation de ce type de structures nécessiterait également une redéfinition des éléments de la librairie standard pour redéfinir le comportement sur ce nouveau type (notamment les opérateurs).

9 Retour d'expérience

Ce stage m'a permis de découvrir plus en détail le monde de la recherche, ainsi que des méthodologies que je n'avais pas rencontrées auparavant. La séparation très claire durant les premières semaines entre l'état de l'art et l'implémentation m'a permis de bien séparer la définition du problème et les solutions envisagées de l'implémentation.

N'ayant jamais travaillé sur l'analyse statique auparavant, j'ai appris beaucoup durant ce stage sur ce sujet. J'ai également découvert le fonctionnement d'un compilateur et d'un générateur de parseur écrit en Java. Ces différents aspects techniques nouveaux ont rendu ce stage encore plus intéressant, car ils ont permis une montée en compétences. Ces aspects totalement inconnus furent au début la source de nombreuses difficultés, mais également la source d'une grande motivation une fois les premières barrières tombées.

La contribution proposée est satisfaisante au niveau personnel, avec un état de l'art riche et varié, et une implémentation préliminaire dont j'estime les résultats encourageants pour le futur.

Références

[1] Julien Ponge, Frédéric Le Mouël, and Nicolas Stouls. Golo, a dynamic, light and efficient language for post-invokedynamic jvm. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform : Virtual Machines, Languages, and Tools*, PPPJ '13, pages 153–158, New York, NY, USA, 2013. ACM.

[2] Golo - a lightweight dynamic language for the jvm. `http://golo-lang.org/`.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wroclaw, Poland, 2011.

[4] Claude Marché and Christine Paulin-Mohring. Reasoning about java programs with aliasing and frame conditions. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, pages 179–194, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792, Rome, Italy, March 2013. Springer.

[6] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml : a java modeling language. In *In Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998.

[7] Nadia Polikarpova and Scott Furia, Carlo A.and West. *Runtime Verification : 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, chapter To Run What No One Has Run Before : Executing an Intermediate Verification Language, pages 251–268. Springer Berlin Heidelberg, 2013.

[8] François Bobot, Sylvain Conchon, E Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The alt-ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`, 2013.

[9] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order smt solvers. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 615–622, New York, NY, USA, 2009. ACM.

[10] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. Loop invariants : Analysis, classification, and examples. *ACM Comput. Surv.*, 46(3) :34 :1–34 :51, Jan 2014.

[11] *Boogie : A Modular Reusable Verifier for Object-Oriented Programs*, 2005.

[12] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon : Program analysis and transformation in java. Research Report RR-5901, Inria, 2006.

[13] K. Rustan M. Leino. *Logic for Programming, Artificial Intelligence, and Reasoning : 16th International Conference, LPAR-16, Dakar, Senegal,*

April 25–May 1, 2010, Revised Selected Papers, chapter Dafny : An Automatic Program Verifier for Functional Correctness, pages 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] Michael Ameri and Carlo A. Furia. Why just boogie ? translating between intermediate verification languages. *CoRR*, abs/1601.00516 :–, 2016.