HardenedGolo : pour augmenter le niveau de confiance en un code Golo

Oscar CARRILLO Nicolas STOULS Raphael LAURENT
Nikolai PLOKHOI Qifan ZHOU Julien PONGE Frédéric LE MOUËL

Univ Lyon, INSA Lyon, CITI, F-69621 Villeurbanne, France
prenom.nom@insa-lyon.fr

Résumé

Cet article décrit un travail préliminaire autour du langage de programmation Golo. Notre objectif est de fournir aux développeurs des outils permettant de renforcer leur confiance en leur code. Pour ce faire, nous avons expérimenté plusieurs approches (test dynamique, analyse de type et preuve de programme) et nous cherchons maintenant des choix pertinents pour avancer dans chacune de ces pistes.

Mots clefs : Golo – typage – test – preuve.

1 Introduction

Golo [13] est un langage et un outil développé dans le cadre du projet *Eclipse* éponyme [1]. Ce langage est dynamiquement typé et s'exécute sur une machine virtuelle Java standard. Son typage dynamique permet notamment de créer des objets dont la structure peut être mutée durant l'exécution. Cela signifie qu'il est par exemple possible d'ajouter de nouveaux attributs ou de nouvelles méthodes à un objet pendant son cycle de vie.

Initialement pensé pour exploiter certaines fonctionnalités de la JVM non exploitables dans le langage Java (instruction `invokeDynamic` du bytecode Java), le cas d'usage de prédilection de Golo est l'IoT. En effet, l'aspect dynamique de l'environnement où de nouveaux objets communicants peuvent apparaitre et disparaitre peut être plus facilement intégré pour faire de l'exécution contextuelle [12].

L'exemple de la Figure 1 est un code Golo mettant en évidence l'évolution de la définition du type d'une variable, avec les contraintes d'exécution que cela pose. On y voit notamment que la fonction `genereDiviseurPar` ajoute une méthode `divise` au paramètre `v` reçu en paramètre. Ainsi, suivant le chemin pris par la fonction `TestDivision`, la variable `v` peut, ou non, posséder la méthode appelée sur sa dernière ligne. Le `main` de cet exemple fait 3 appels successifs à cette fonction, pour mettre en évidence 3 cas de figure : la division a lieu, une méthode inexistante est appelée et une division par 0 est réalisée. Il est intéressant de noter que si aucun des deux derniers cas n'est souhaitable, seul le dernier génère une exception. Le cas 2 renverra le résultat d'un appel à `toString`.

1. `https://projects.eclipse.org/projects/technology.golo`

Dans le cadre de la présente communication, nous décrivons une réflexion préliminaire visant à fournir aux développeurs Golo des outils permettant de renforcer leur confiance en leur code [11]. Pour ce faire, nous avons commencé à expérimenter différentes approches : analyse symbolique du code pour générer de cas de tests, analyse de type et preuve de programme. Les avantages tirés et la complexité de mise en œuvre de chacune de ces solutions ne sont pas les mêmes. C'est pourquoi, nous voudrions pouvoir laisser le choix au développeur de vérifier les différentes parties de différentes manières sans qu'une politique globale d'évaluation soit formellement définie. Dans un monde idéal, notre objectif serait de pouvoir maintenir un *bilan de confiance* en le logiciel, en maintenant à jour tout ce qui a été vérifié et ce qui devrait l'être. De même, les outils de vérification étant intégrés dans le compilateur lui même, il serait envisageable que les éléments non vérifiés puissent être compilés avec l'ajout automatique d'une instrumentation vérifiant à l'exécution le respect d'une propriété. En conservant également la description de la méthode utilisée, nous espérons qu'il serait ensuite possible de rejouer les vérifications, pour faire de la non-régression.

```
module hardenedgolo.dynamic
local function genereDiviseurPar = |v,denum| {
    v:define("divise", |this,num| -> num/denum)
}

local function TestDivision = |n,d| {
    let v = DynamicObject()
    if(d!=1) { # Erreur. Ça aurait du être d!=0
        genereDiviseurPar(v,d)
    }
    return v:divise(n)
}

function main = |args| {
    # Cas 1 : Ça fonctionne
    var res = TestDivision(42,12)
    println("42 / 12 = "+res)

    # Cas 2 : appel d'une méthode qui n'existe pas
    res = TestDivision(42,1)
    println("42 / 1  = "+res)

    # Cas 3 : division par 0
    res = TestDivision(42,0)
    println("42 / 0  = "+res)
}
```

FIGURE 1 – Exemple

L'implémentation de nos expérimentations est disponible dans l'outil HardenedGolo [2], diffusé sous licence *Eclipse*. Les parties bilan de santé, rejeu et ajout de code de vérification à l'exécution n'ont pas encore été étudiées.

Dans la suite de cette communication, nous présentons nos expérimentations initiales sur les différentes approches de vérification, puis nous établissons un lien avec l'état de l'art qui nous semble pertinent pour avancer dans ce travail, avant de conclure sur les perspectives de ce travail.

2 Pistes explorées

Dans nos expérimentations visant à augmenter la confiance en un code Golo, nous avons exploré trois pistes développées ci-après : le test symbolique, vérification de typage et la preuve de programme. L'ajout de spécifications permettant d'aider les outils d'analyse, nous avons également proposé une adaptation d'un langage de spécification par annotations.

2. https://github.com/dynamid/HardenedGolo

2.1 Test Symbolique

Nous avons exploré la génération de cas de tests par exécution symbolique [9]. Cette analyse va générer des jeux de valeurs d'entrée qui permettront ensuite d'exécuter les chemins sélectionnés.

Pour pouvoir générer ces valeurs d'entrée, nous avons besoin d'un *solveur de contraintes* qui permette de définir un domaine de définition de chaque donnée. Cependant, certaines expressions ne sont pas solvables en arithmétique linéaire [2]. Ainsi, pour diminuer le nombre de conditions sans solution, nous proposons d'utiliser des tests symboliques dynamiques, et notre choix s'est porté sur le *Concolic Test* [6]. Ce type de test permet de trouver des valeurs pour les fonctions complexes à partir de l'exécution du programme avec un mix des valeurs concrètes et symboliques.

À ce jour, le compilateur HardenedGolo intègre un outil de génération de tests symboliques dynamique (`golo symtest`), qui explore les différents chemins et en même temps trouve une solution possible pour le chemin exploré. Pour résoudre les conditions des valeurs nous utilisons *Choco solver* [8] pour sa facilité d'utilisation dans Java. À ce stade, notre génération de tests contient encore de grosses limites et le non respect de ces limites arrêtera l'exploration du code :

- les variables manipulées ne peuvent être que des valeurs entières ;
- seuls des opérateurs simples peuvent apparaitre dans les conditions mathématiques envoyées au solveur (`+,-,*,/,%,=,>,<,!=`) ;
- pas de support pour l'appel de fonctions ;
- pas de support pour les boucles ;
- seul le critère de couverture *tous les chemins* a été implémenté ;
- pas d'utilisation des annotations de spécification présentes dans le code (et présentées dans la suite).

2.2 Vérification de typage

Étant donnée la nature dynamique de Golo et l'impossibilité dans le langage de pouvoir typer les paramètres d'une fonction, il est impossible de pouvoir toujours déterminer le type d'une variable et donc de savoir s'il est autorisé d'y appeler une méthode ou d'y lire un attribut. D'autant plus, qu'en plus de la mutabilité des types, il est possible de réaffecter une variable avec une donnée d'un autre type. Cependant, avoir la connaissance d'informations de type est indispensable pour certaines analyses. C'est pourquoi nous avons commencé à développer les prémisses d'une analyse de type pour vérifier leur concordance, en commençant par certains cas simples.

Pour l'instant, l'analyse implémentée se base sur les annotations, qui contiennent des informations de typage et sur la concordance des usages en se limitant aux trois familles de types : numérique, booléen et autre. Il nous est ainsi possible de lever des `warning` dans certains cas simples, tel que par exemple l'usage d'une valeur booléenne dans une expression arithmétique.

2.3 Ajout d'annotations de spécification

Pour permettre au développeur d'exprimer des propriétés sur le programme vérifié, nous proposons d'utiliser des annotations dans le programme. Cela n'est pas une technique de vérification, mais un outil permettant au concepteur d'exprimer des propriétés à vérifier sur son système. Mais cela permet également de gagner en connaissances sur le programme notamment par rapport au typage des éléments.

Au jour d'aujourd'hui, notre développement HardenedGolo supporte l'ajout d'annotations de spécifications aux fonctions. Le langage utilisé est celui de WhyML [5], encapsulé entre des balises `spec/ ... /spec`. Ces annotations sont actuellement exploitées pour faire une analyse de type simple des variables. Elles sont également propagées dans la traduction en WhyML de fonctions Golo, comme discuté dans la section suivante. L'exemple de la figure 3 montre une fonction de valeur absolue dont la pré et la post-condition (resp. `requires` et `ensures`) sont décrites avec pour objectif de vérifier son correct usage. Excluant donc de fait l'entier représentable minimum (`MinInt`) des valeurs autorisées en entrée.

Cependant, pour prendre en compte la dynamique du langage, certains cas particuliers sont à considérer. Par exemple, la figure 2 montre un exemple de quelque chose que l'on veut pouvoir faire, mais où nous n'avons pas encore décidé comment le faire ou l'exprimer. Dans cet exemple, la fonction `calculateur` reçoit trois paramètres et nous voulons exprimer que le troisième paramètre est un objet fournissant au moins la fonction `additionneur`. Cette fonction doit avoir comme pré-condition que ses deux paramètres sont des entiers 32bits et que son résultat est un entier 32bits. Cela peut ensuite exploser si, par exemple, cette fonction doit elle même renvoyer une fonction renvoyant un objet devant posséder une méthode dont le type de retour doit être compatible avec un type donné.

```
local function calculateur = |x,y,a| spec/
        requires{
            x:Int32 /\ y:Int32 /\
            hasmethod(a,                    # Nom de l'objet
                      additionneur |v1,v2|, # Méthode qu'il doit posser
                      v1:Int32 /\ v2:Int32 /\ result:Int32 # Spec de additionneur
                     )
        }
        ensures { (result :Int32) }
    /spec {
    return a:additionneur(x,y)
}
```

Fonction dont la spécification tente d'exprimer qu'un paramètre doit posséder une méthode respectant un certain cahier des charges.

FIGURE 2 – Exemple de pré-condition avec propriété sur l'objet reçu

2.4 Preuve de programme

Le langage Golo étant ce qu'il est, il ne nous semble pas raisonnable de croire que les développeurs Golo auront l'envie ou le besoin de prouver des pans entiers de programme. Cependant, l'objectif étant d'augmenter la confiance que l'on peut avoir en un code, il nous semble intéressant de permettre la preuve de parties de programme. Dans ce cas là, plusieurs

questions se posent. L'une d'entre elle est de savoir quel crédit accorder à un programme partiellement prouvé.

En nous inspirant de l'approche de Krakatoa [4], nous avons proposé la traduction d'un code Golo vers du code WhyML. En adressant ainsi l'outil Why, nous espérons pouvoir bénéficier de tout son écosystème.

Dans la version actuelle de nos développements [10], la traduction vers WhyML ne supporte que les fonctions, les types entiers et les conditionnelles simples. Les annotations de fonction sont préservées dans le WhyML produit, ce qui permet de prouver le respect de certaines propriétés. La figure 3 est un exemple de fonction pouvant bénéficier de cette vérification, où l'absence de la pré-condition ne permettrait plus de vérifier la post-condition.

```
function myAbs = |x| spec/
        requires{ x >= -2147483647 } # 2147483647 = MinInt+1
        ensures{ (result >= 0) /\ (result = x \/ result = -x) }
    /spec {
    if (x < 0){
        return (0 - x)
    } else {
        return x
    }
}
```

Fonction pouvant être prouvée et où l'oubli de la pré-condition pourra bloquer la preuve.

FIGURE 3 – Exemple de fonction Golo prouvable

3 État de l'art

Dans la littérature, le problème de la vérification de programmes décrits dans des langages dynamiques est assez largement abordée. Par exemple, les travaux Quist, et. al [1] portent sur la vérification de type par induction et la génération de cas de test pour des programmes dynamiques, décrits en Dart. Ils exploitent un mécanisme d'analyse statique pour obtenir des assurances de complétude sur le taux de couverture des tests produits.

De même, Csallner et Smaragdakis [3] ont proposé une approche de vérification de programmes Java combinant l'analyse statique et la génération de tests. Leur approche consiste à partir d'une sur-approximation des cas de test et de les élaguer par résolution de contraintes.

Ces différents résultats sont très intéressants par rapport à nos objectifs. Mais dans la plupart des cas, les langages dynamiques étudiés sont polymorphes de niveau 1 (prenex). C'est notamment le cas des langages Dart ou Java. À l'inverse, le polymorphisme de Golo, permettant l'extension dynamique, est de niveau N, comme pour Lisp, JavaScript, Python, Self ou Ruby.

Le problème d'inférence de type d'un système 2-polymorphe a été prouvé impossible dans le cas général par Schubert [14]. Mais sans information de typage, les vérifications statiques seront impossibles. Parmi les solutions proposées pour aborder ce problème, nous avons identifié la thèse de Don Stewart [15], qui propose notamment de décomposer le programme à vérifier en fragments, puis à vérifier dynamiquement la compatibilité de type entre eux.

L'analyse proposée dans leur outil est faite en plusieurs passes (certaines statiques et d'autres dynamiques), mais c'est finalement dynamiquement que l'analyse finale est faite.

Une autre approche utilisée pour vérifier statiquement le respect d'une politique de sécurité par un programme Javascript consiste à considérer un sous-ensemble vérifiable du langage [7]. Nous ne devrons pas négliger cette solution, par exemple pour permettre l'usage d'outils de preuve de programme sur des parties du système qui seraient identifiées comme critiques.

4 Conclusion et perspectives

Notre objectif n'est pas d'avoir un outil permettant de prouver intégralement un programme Golo. Non seulement cela serait impossible, mais en plus, cela ne correspondrait pas aux besoins liés à son usage. Notre objectif est de permettre aux développeurs de renforcer leur confiance en leur code. Pour ce faire, nous voulons pouvoir exploiter les avantages des différentes approches explorées.

Ainsi, un concepteur pourrait rapidement lancer des tests structurels de son code. Puis après ajout de spécifications, pourrait vérifier des éléments de typage et des tests fonctionnels. Enfin, le développeur pourra vouloir garantir différentes propriétés particulières sur certaines parties de son code par de la preuve. Afin de pouvoir cumuler les différentes approches en laissant la main au développeur, il sera important que nous ayons un moyen de produire un bilan de santé de l'application vérifiée, qui listerait l'ensemble des actions réalisées avec succès et l'ensemble des parties peu ou pas vérifiées. Par ailleurs, si ce rapport contient une mémoire des actions réalisées et des résultats obtenus, alors il serait intéressant de permettre de rejouer les actions faites, dans une démarche de non-régression.

Actuellement, presque tout reste à faire. Seuls de petits prototypes de chaque piste ont été expérimentés et nous sommes preneurs de tous les avis. Notamment, nous n'avons pas trouvé dans la littérature de manière propre de décrire des propriétés faisant référence à des éléments de type d'une donnée plutôt qu'à son type. L'objectif serait notamment d'exprimer une propriété que devrait remplir une fonction prise en paramètre, ou de spécifier une fonction ajoutée durant l'exécution à une variable.

Références

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Analyzing test completeness for dynamic languages. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 142–153. ACM, 2016.

[2] Cristian Cadar and Koushik Sen. Symbolic execution for software testing : three decades later. *Communications of the ACM*, 56(2) :82–90, 2013.

[3] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash : Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM.

[4] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *International Conference on Computer Aided Verification*, pages 173–177. Springer, 2007.

[5] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[6] Patrice Godefroid. DART : Directed Automated Random Testing. In *conference on Programming language design and implementation (PLDI '05)*, pages 213–223, 2005.

[7] Salvatore Guarnieri and V Benjamin Livshits. Gatekeeper : Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security Symposium*, volume 10, pages 78–85, 2009.

[8] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco : an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP'08)*, pages 1–10, Paris, France, 2008.

[9] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1976.

[10] Raphael Laurent. Hardened Golo : Donnez de la confiance en votre code Golo. Master's thesis, INSA Lyon, June 2016.

[11] Yannick Loiseau. API for unified tests in Golo. https ://github.com/golo-lang/rigolo/blob/master/RiGolo/accepted/tests-api.adoc, 2016.

[12] Baptiste Maingret, Frédéric Le Mouël, Julien Ponge, Nicolas Stouls, Jian Cao, and Yannick Loiseau. Towards a decoupled context-oriented programming language for the internet of things. In *Proceedings of the 7th International Workshop on Context-Oriented Programming, COP 2015, Prague, Czech Republic, July 4-10, 2015*, pages 7 :1–7 :6. ACM, 2015.

[13] 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*, pages 153–158. ACM, 2013.

[14] Aleksy Schubert. Second-order unification and type inference for church-style polymorphism. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 279–288. ACM, 1998.

[15] Don Stewart. *Dynamic extension of typed functional languages*. PhD thesis, PhD thesis, School of Computer Science and Engineering, University of New South Wales, 2010.