TP4 Bis - Test automatique de propriétés et fuzzing

Fichiers de TP : `TP4Bis_ACF.zip`

Ce TP est une introduction au test automatique de propriétés (property based testing). Ceci vous permettra de voir qu'il est possible de définir des propriétés logiques et de les tester automatiquement sur n'importe quel code, et non uniquement dans le cadre d'un assistant de preuve. Vous allez implémenter un générateur de valeurs "SmallCheck" qui est utilisé par la commande `quickcheck [tester=exhaustive]` d'Isabelle/HOL.

1 Objectif du TP

Dans ce TP, vous allez devoir trouver automatiquement des bugs dans des simplificateurs d'expressions arithmétiques. Les expressions arithmétiques sont définies dans le package `expression` par le type Scala suivant :

```
sealed trait Expression
case class Constant(a: Int) extends Expression
case class Variable(a: String) extends Expression
case class Sum(a: Expression, b: Expression) extends Expression
case class Sub(a: Expression, b: Expression) extends Expression
```

Un simplificateur d'expression est une fonction qui, à partir d'une expression `e1`, rend une expression `e2` qui est plus simple ou identique mais toujours équivalente. Deux expressions sont équivalentes, si pour toutes valeurs des variables, elles s'évaluent à la même valeur entière. Par exemple, l'expression `(0 + (x + 0)) + y` peut être simplifiée en `x + y` car ces deux expressions s'évaluent de la même façon pour toute valeur de `x` et de `y`. Si on tente de simplifier l'expression `x + y` on obtiendra la même expression. En particulier, on ne peut pas simplifier `x + y` en `x` car il existe des valeurs de `x` et de `y` telles que `x + y` et `x` s'évaluent de façon différente.

2 Les fonctions disponibles sur les expressions

Dans l'objet `Calculator` du package `calculator`, vous disposez des fonctions élémentaires suivantes :

- Une fonction permettant d'évaluer une expression :
  `def compute(e: Expression, env: Map[String, Int]): Option[Int]`
  Cette fonction prend en paramètre une expression et un environnement. Un environnement est une table associant des noms de variables à une valeur entière. Le résultat de cette fonction est un `Option[Int]` qui sera soit `Some(v)` d'une valeur entière `v` quand l'expression s'évalue en `v`, soit `None` quand l'expression n'est pas évaluable. Une expression `e` ne sera pas évaluable si elle contient une variable qui n'a pas de valeur dans `env`.
- Une fonction permettant de calculer l'ensemble des variables d'une expression :
  `def variables(e:Expression): Set[String]`

— Deux fonctions de simplification d'expression : `def simplify1(e:Expression):Expression` (très basique) et `def simplify2(e:Expression):(Expression,List[(Int,Int)])` un peu plus puissant.

Dans le package `calculator`, vous avez également accès à deux objets exécutables `MainSimplify1` et `MainSimplify2` qui permettent de jouer avec les simplificateurs sur des expressions textuelles (tapez `sbt run` pour lancer les simplificateurs).

3 Marche à suivre pour automatiser la recherche de bug

Les deux fonctions de simplification `simplify1` et `simplify2` sont incorrectes : une expression `e1` peut être simplifiée en une expression `e2` qui n'est pas équivalente. L'objectif est de trouver, pour chaque simplificateur, au moins une valeur de `e1` telle que la valeur simplifiée `e2` n'est pas équivalente **pour un environnement définissant toutes les variables de** `e1`. Or, vous ne disposez pas du code de `simplify1` et `simplify2`. Vous allez devoir les tester en **boîte noire** (tester un programme sans disposer de son code). Pour savoir si un test est réussi, il faut définir la propriété logique que l'on attend sur ces simplificateurs.

Schéma général pour tester (automatiquement) la propriété

1. A l'aide de `compute`, `variables`, `simplify1` (resp. `simplify2`), écrivez la formule logique ϕ garantissant la correction de `simplify1` (resp. `simplify2`) ;
2. Programmez, en Scala, des générateurs pour des valeurs possibles pour les variables de la formule ϕ (voir Sections 4 et 5) ;
3. Programmez, en Scala, la vérification de la formule ϕ pour chaque combinaison des valeurs générées dans l'item précédent.

4 Test de propriétés sur `simplify1`, le cas de Quickcheck/SmallCheck

Pour tester une propriété, la solution la plus connue est QuickCheck [1] qui repose sur la génération aléatoire de valeurs de tests. Ici, on propose de s'appuyer sur l'approche SmallCheck [2] qui vise à générer de façon "exhaustive" un ensemble de valeurs de taille bornée par un entier n. Dans notre cas, on souhaitera générer valeurs bornées par n pour les types `Int`, `String`, `Expression`, `Map[String,Int]`. Dans l'objet `SmallCheck`, programmez les fonctions (génératrices) suivants :

— `genInt(n:Int):Set[Int]` qui génère l'ensemble des entiers compris entre $-n$ et n ;
— `genString(n:Int):Set[String]` qui génère un ensemble de n chaînes différentes ;
— `genExpression(n:Int):Set[Expression]` qui génère l'ensemble de **toutes** les expressions de hauteur n (ou moins) que l'on peut construire à partir de tous les entiers pris dans `genInt(n)` et toutes les variables dont les noms sont pris dans `genString(n)`. Par exemple, `Constant(1)` est de hauteur 1, `Sum(Constant(1),Variable("X"))` est de hauteur 2 et `Sum(Sum(Constant(1),Variable("X")), Constant(1))` est de hauteur 3.
— `genEnvironment(n:Int):Set[Map[String,Int]]` qui génère l'ensemble de tous les environnements comprenant n associations (ou moins) que l'on peut construire à partir des variables de noms pris dans `genString(n)` et les valeurs entières prises dans `genInt(n:Int):Set[Int]`.

Vous pouvez contrôler vos générateurs à l'aide des tests unitaires dans la classe `smallCheck.TestSmallCheck` du répertoire `test`. Utilisez `sbt testOnly smallCheck.TestSmallCheck`. L'ensemble `genExpression(1)` est de taille 4 (par exemple l'ensemble `Set(Variable("A"), Constant(1), Constant(0), Constant(-1))`).

1. `https://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf`
2. `https://www.cs.york.ac.uk/fp/smallcheck/smallcheck.pdf`

L'ensemble `genExpression(2)` est de taille 105 (7 expressions de hauteur 1, 7*7 expressions `Sum` de hauteur 2 et 7*7 expressions `Sub` de hauteur 2). L'ensemble `genExpression(3)` est de taille 88210. Pour contrôler votre génération, l'ensemble `genEnvironment(1)` est, par exemple, `Set(Map(), Map("A" -> -1), Map("A" -> 0), Map("A" -> 1))`. L'ensemble `genEnvironment(2)` est de taille 36 (1 + 5*2 + 5*5).

Enfin, programmez la vérification de votre propriété de correction ϕ sur `simplify1` (voir Section 3) dans la méthode `testSmallCheck` de la classe `calculator.TestSimplify1` du répertoire `test`. On rappelle que l'on cherche une expression `e1`, un environnement `env` (donnant une valeur à toutes les variables de `e1`), tels que en évaluant `e1` sur `env` et en évaluant le résultat de la simplification de `e1` sur `env`, on obtient un résultat différent. Tapez `sbt testOnly calculator.TestSimplify1` pour exécuter les tests. SmallCheck et une génération bornée par 3 devrait être suffisant pour trouver des exemples d'expressions qui sont incorrectement simplifiées par `simplify1`. Si vous voulez trouver une expression incorrectement simplifiée par `simplify2` vous allez devoir passer à une autre technique nommée *fuzzing*.

5 Optionnel – Pour trouver le bug dans `simplify2`, le fuzzing

Pour aller plus loin, il va être nécessaire de **guider la génération** de valeurs en ne gardant les valeurs qui sont **pertinentes**, i.e., des valeurs testant des branches du programme qui n'ont pas encore été testées. L'approche de test précédente est qualifiée d'approche en "boîte noire" car on teste le programme sans disposer d'information sur le code ou sur son exécution. A l'autre bout du spectre, il existe des approches "boîte blanche" où on dispose du code et on l'exploite pour déterminer automatiquement les valeurs pour couvrir tous les chemins d'exécution possibles. Une troisième possibilité, appelée **fuzzing**, est une approche "boîte grise" où on ne dispose pas de l'intégralité du code mais on peut connaître certaines informations. Par exemple, en exécutant un programme binaire sur une donnée de test, avec des outils de débug ou de couverture de test, on peut déterminer le chemin d'exécution pour ce test. Un **chemin d'exécution** est la suite des adresses en mémoire des instructions (essentiellement les instructions conditionnelles) exécutées pour calculer le résultat à partir de cette valeur de test.

En fuzzing, on part d'une (petite) base de tests et on recueille les chemins d'exécution auxquels ils correspondent. On enrichit cette base par de nouveaux tests obtenus par mutations ou **croisements** entre les tests de la base. Un test obtenu de cette façon est intégré à la base s'il correspond à un chemin d'exécution non couvert par les tests de la base. Voici comment procéder pour appliquer le fuzzing à `simplify2`. Les fonctions sont à ajouter à l'objet `fuzzing.Fuzzing` et les fonctions de test à la classe `fuzzing.TestFuzzing` du répertoire `test`.

- Définir une fonction `breeding(e1:Expression,e2:Expression):Set[Expression]` qui donne l'ensemble des croisements entre deux `e1` et `e2`. On obtient cet ensemble de croisements en remplaçant dans `e1` toutes les sous-expressions par `e2`. Par exemple, l'ensembles des croisements de l'expression `Sum(Constant(1), Constant(2))` et l'expression **Sub(Constant(3),Constant(4))** est l'ensemble constitué des expressions `Sum(`**`Sub(Constant(3),Constant(4))`**`,Constant(2))` et de l'expression `Sum(Constant(1),`**`Sub(Constant(3),Constant(4))`**`)`.
- Pour vous simplifier le travail, `simplify2` rend un couple contenant l'expression simplifiée et son chemin d'exécution (une liste de couples d'entiers [3]). Définissez une table `allPathes` associant un chemin d'exécution (de type `List[(Int,Int)]`) à l'expression correspondante.
- Réalisez la procédure de fuzzing :

3. Le premier entier représente le point de programme conditionnel et le second le numéro de la branche suivie.

1. Pour initialiser la base de tests utilisez, par exemple, l'ensemble des expressions de `genExpression(2)` de la section précédente. Pour chaque expression, vérifiez votre théorème de correction ϕ sur `simplify2` et obtenez le chemin d'exécution. Dans la table `allPathes`, collectez les nouveaux chemins couverts et les expressions correspondantes.
2. Utilisez `breeding` pour construire de nouvelles expressions à partir de celles présentes dans `allPathes`.
3. Pour chaque nouvelle expression, vérifiez comme au dessus si elle est correctement simplifiée et collectez les nouveaux chemins couverts.
4. Itérez en reprenant au point 2.