GL38

Bilan sur l’étape de conception

Antoine Turkie, Augustin Chenevois, Aurélien Delory, Mihaja Razafimahefa, Sarah Ramaharobandro

Grenoble INP - Ensimag · Projet GL 2023-2024

Implémentations nécessaires à l'étape A et B

Conception architecturale

Liste de classes qu'on a créé à l'aide de la grammaire des arbres de syntaxe abstraite pour compléter le package *fr.ensimag.deca.tree* :

- Les sous-classes abstraites directes de Tree avec leurs classes concrètes et leurs atributs :

    - **AbstractDeclField**
        - *DeclField* extends AbstractDeclField
            - Visibility fieldVisibility
            - AbstractIdentifier fieldType
            - AbstractIdentifier fieldName
            - AbstractInitialization fieldInitialization

    - **AbstractDeclMethod**
        - *DeclMethod* extends AbstractDeclMethod
            - AbstractIdentifier methodReturnType
            - AbstractIdentifier methodName
            - ListDeclParam methodParamaters
            - AbstractMethodBody methodBody

    - **AbstractDeclParam**
        - *DeclParam* extends AbstractDeclParam
            - AbstractIdentifier paramType
            - AbstractIdentifier paramName

    - **AbstractMethodBody**
        - *MethodBody* extends AbstractMethodBody
            - ListDeclVar declVar
            - ListInst inst

        - *MethodAsmBody* extends AbstractMethodBody
            - StringLiteral asmBody

- Les sous-classes concrètes de TreeList<> :

    - **ListDeclField** extends TreeList<AbstractDeclField>
    - **ListDeclMethod** extends TreeList<AbstractDeclMethod>
    - **ListDeclParam** extends TreeList<AbstractDeclParam>

- Les sous-classes concrètes directes ou indirectes de AbstractExpr avec leurs attributs :

    - **Instanceof** extends AbstractBinaryExpr
        - AbstractExpr leftOperand
        - AbstractExpr rightOperand

    - **Cast** extends AbstractUnaryExpr
        - AbstractIdentifier typeAfterCast
        - AbstractExpr Operand

    - **MethodCall** extends AbstractExpr
        - AbstractExpr expr
        - AbstractIdentifier identifier
        - ListExpr listArgs

    - **Selection** extends AbstractLValue
        - AbstractExpr thisExpr
        - AbstractIdentifier identifier

    - **This** extends AbstractExpr
        - boolean parsed

    - **New** extends AbstractExpr
        - AbstractIdentifier nameClass

    - **Null** extends AbstractExpr

- Les sous-classes concrètes directes ou indirectes de AbstractInst :

    - **Return** extends AbstractInst
        - AbstractExpr returnExpr
        - String className : nom de la classe qui possède une méthode qui appelle l'instruction return

        - String methodName : nom de la méthode qui appelle l'instruction return

Liste des classes qu'on a créé pour la vérification contextuelle dans le package *fr.ensimag.context* :

- **ObjectType** extends ClassType : représente le type prédéfini Object, la méthode *boolean isObject()* retourne true pour cette classe

Algorithmes et structure de données spécifiques

Dans certaines classes du package *fr.ensimag.context*, nous avons ajouté plusieurs structures de données pour stocker des informations nécessaires à la vérification contextuelle, notamment pour les environnements :

- Dans la classe **EnvironmentExp :**
    - un dictionnaire **HashMap<Symbol, ExpDefinition>** qui permet d'accéder à la définition d'un Symbo en O(1), on a également choisi cette structure de donnée car la recherche d'un Symbol se fait en O(1) ce qui permet à l'union disjointe de deux environnements d'être plus efficace
    - on a écrit la méthode **stackEnvironment** qui implémente l'empilement de deux EnvironmentExp et on a fait une surcharge de la méthode **declare** qui implémente l'union disjointe
- Dans la classe **EnvironmentType** :
    - on a ajouté un attribut **ObjectType OBJECT**, instance de la classe qu'on a créé et on l'a mis dans le dictionnaire qui stocke les Symbol et leur TypeDefinition
    - on a également ajouté un attribut **AbstractIdentifier objectClassIdentifier** qui correspond l'identifiant de la classe Object en Deca. Ainsi, nous n'avions pas besoin de créer un nouvel Identifier à chaque fois que la classe Object était déclarée en tant que superclasse dans un programme Deca

- on a implémenté les méthodes **stackOneElement** et **declare** qui implémentent respectivement l'empilement/union disjointe d'un couple dans l'environnement

- Dans la classe **ClassDefinition** :

  - on a ajouté un **constructeur** qui permet de créer une définition spécifique à la classe Object de Deca, c'est à dire qui ajoute la MethodDefinition de equals dans son environnement

  - on a ajouté un attribut **String currentMethodNameForReturn** qui permet au codeGen de return de récupérer la méthode qui l'appelle afin de créer l'étiquette de fin de la méthode. Les attributs String className et String methodName de la classe Return ont été défini dans l'étape B (quand la méthode verifyInst de Return est appelée) où on a accès à la *ClassDefinition currentClass*

Opérations et prédicats sur les domaines d'attribut

Des opérations et des prédicats doivent être définis pour les différentes passes de la vérification contextuelle, ainsi on a implémenter :

- Dans la classe **Type** :

  - La méthode **boolean isSubType(Type potentialParentType)** qui détermine si l'appelant vérifie la relation de sous-typage définie dans le polycopié. On vérifie, dans l'ordre, trois conditions dans cette méthode :

    - si l'instance de Type appelant est une instance de NullType alors la méthode renvoie true

        - si l'instance de Type appelant et le paramètre *Type potentialParentType* sont tous les deux des instances de ClassType alors on appelle la méthode **boolean isSubClassOf(ClassType potentialSuperClass)** qui renvoie le résultat

        - si l'instance de Type appelant et le paramètre *Type potentialParentType* sont tous les deux des instances du même type, c'est à dire que **boolean sameType(Type otherType)** renvoie true alors le résultat est true

On note qu'on a redéfini les méthode **boolean sameType(Type otherType)** pour les types prédéfinis IntType, FloatType, BooleanType, VoidType, NullType et ObjectType en utilisant l'égalité des noms de leur Symbol.

- Dans la classe **ClassType** :

    - La méthode **boolean isSubClassOf(ClassType potentialSuperClass)** a été écrite en respectant la définition de la relation de sous-typage. De plus, une première condition qu'on vérifie est que si l'instance de ClassType appelant la méthode est de type Object alors le résultat est false. En effet, Object ne peut pas être une sous-classe et ainsi lorsqu'on tente de récupérer la superclasse pour appeler de nouveau la méthode, cela ne renvoie pas une erreur.

- Dans la classe **EnvironmentType** :

    La méthode **boolean assignCompatible(Type t1, Type t2)** qui détermine la compatibilité pour l'affectation relativement à un EnvironmentType. Pour cela, trois conditions ont été vérifiées :

        - Si t1 est une instance de FloatType et t2 une instance de IntType alors le résultat est true

        - Si t1 et t2 sont des instances du même type alors le résultat est true

        - Sinon, le résultat est celui renvoyé par t2.isSubType(t1)

    - La méthode **boolean castCompatible(Type initialType, Type finalType)** qui détermine la compatibilité pour la conversion relativement à un EnvironmentType. Pour cela, deux conditions ont été vérifiées :

    - Si **assignCompatible(initialType, finalType)** renvoie true alors le résultat est true

    - Si **assignCompatible(finalType, initialType)** renvoie true alors le résultat est true

- En ce qui concerne la signature des opérateurs, nous n'avons pas implémenté de méthode mais les conditions qui sont décrites dans le polycopié ont été directement vérifiées au sein des méthodes verify des classes AbstractOpArith, AbstractOpBool, AbstractOpExactCmp, AbstractOpCmp.

- On a également fait une redéfinition exceptionnelle de la méthode *equals* dans la classe **Signature** pour pouvoir vérifier l'égalité entre deux List<Type> en prenant en compte l'ordre des éléments de la liste et en vérifiant l'égalité entre chaque élément de la liste. On note qu'une bonne pratique aurait été de redéfinir également la méthode *hashCode* après une redéfinition de equals mais nous avons oublié de faire cette modification.

Implémentation du this implicit

Afin de permettre l'appel de méthode implicite au sein d'une classe ou la sélection implicite d'un champ, nous avons ajouté :

Dans **DecaParser.G4 :**

- Lorsqu'on atteint la règle qui implémente l'appel de méthode implicite, un nouvel arbre de **This** est instancié avec son attribut *boolean parsed* qui vaut false, sa Location est aussi définie et on l'utilise en tant que paramètre de l'arbre **MethodCall** qui est créée.

- Pour la sélection implicite d'un champ, la règle (3.18) qui spécifie l'environnement que ListInst prend en paramètre permet déjà à l'identificateur d'un field d'être reconnu cependant si un paramètre est défini avec le même nom qu'un field, sa définition écrase celle du field dans l'environnement.

Arbre enrichi

Dans le but d'avoir toutes les décorations sur l'arbre abstrait enrichi, on a implémenté ou complété certaines méthodes dans les classes qu'on a créé pour compléter le package *fr.ensimag.deca.tree*

- Dans la classe **Visibility :**
    - on a redéfini les méthodes **String toString()** des deux enum PUBLIC et PROTECTED, cela a été utile pour les fonctions qui implémentent l'affichage de l'arbre enrichi
- Dans la classe **AbstractDeclField :**
    - on a redéfini la méthode **String printNodeLine(PrintStream s, String prefix, boolean last, boolean inlist, String nodeName)** pour que la visibilité soit affichée sur l'arbre enrichi pour un field
- Dans toutes les autres classes qu'on a créé, nous avons redéfini les méthodes **void prettyPrintChildren(PrintStream s, String prefix)** et **void iterChildren(TreeFunction f)** en appelant respectivement à chaque fois les méthodes **void prettyPrint(PrintStream s, String prefix, boolean last)** et **void iter(TreeFunction f)** sur les attributs de la classe
- En ce qui concerne le nœud ConvFloat qu'on ajoute éventuellement dans l'étape B, il s'agit pour chaque instruction d'un setOperand sur l'expression à convertir. En effet une nouvelle instance de **ConvFloat** remplace l'expression.

Implémentations nécessaires à l'étape C

Conception architecturale

Voici les classes que nous avons créé afin de réaliser l'étape de génération de code:

- **CodegenHelper** :
    - Génération de code pour la gestion d'erreur via la fonction **codeGenListError**
    - Calcul de la profondeur de pile maximale afin de de pouvoir détecter les dépassements de pile avec seulement deux instructions par blocs (**TSTO** + **BOV**)
    - Implémentation de la méthode equals de la classe **Object**
- **VTable** :
    - Crée la VTable d'une méthode. On récupère la VTable de la classe mère et on ajoute les labels liés aux nouvelles classes. Si une méthode est héritée, on la place au même index dans la VTable que dans la VTable du parent.
    - Attribution d'une adresse relative à **GB** pour chaque VTable
    - Génération du code via les instruction **LEA** et **STORE**
- **ListVTable** :
    - Cette classe contient un dictionnaire, **vTableMap** qui associe à chaque nom de classe sa VTable.

Algorithmes et structure de données spécifiques

En dehors de certaines opérations binaires, presque chaque classe de l'arbre abstrait possède une fonction **codeGenInst(DecacCompiler compiler)** qui génère le code associé à cette classe.

La classe **DecacCompiler** contient le registre que l'on utilise actuellement via la fonction **getRegister()** ainsi qu'une **DValue** utilisée pour les opérations binaires via la fonction **getDVal()**. Le registre est celui dans lequel le résultat de **codeGenInst** doit être enregistré et est initialisé à **R2**. On l'incrémente lorsque l'on doit réaliser des opérations binaires ou on utilise les instructions **PUSH** et **POP** ainsi que le registre **R0** dans le cas où on a plus aucun registre est disponible.

On peut voir cet algorithme dans la classe **AbstractBinaryExpr**, dans la fonction **codeGenInst**.

La génération de code des opérations binaires arithmétiques, booléennes et de comparaisons sont réalisées via la fonction **codeGenBinary(Decac compiler)** qui réalise la génération de code en utilisant **getRegister()** et **getDVal()** comme opérandes, qui ont été auparavant positionnés dans la fonction **codeGenInst** de **AbstractBinaryExpr**.

Aucune variable n'est stockée dans les registres. Ainsi, après la génération de code associée à une instruction, le registre stocké par le compiler redevient toujours **R2**.

La génération de code pour les méthodes se fait de la même manière que celle du main. Cependant, durant cette étape, on enregistre le registre d'indice maximal que l'on a utilisé afin de push les registres d'indice inférieur dans la pile à l'entrée de la méthode et de les restaurer à la sortie. On crée aussi un second **IMAProgram** afin de pouvoir ajouter des instructions au début.