Projet Génie Logiciel

Documentation de Conception

Aurélien VILMINOT
Damien CLAUZON
Guilherme KLEIN
Léon ROUSSEL
Pierre ARVY

26 janvier 2022

Résumé

Le document présent est une documentation de la conception du compilateur Deca développé lors du projet de Génie Logiciel. Cette documentation s'adresse à un développeur souhaitant maintenir et/ou ajouter des fonctionnalités au compilateur. Elle décrit l'organisation générale de l'implémentation :

— Les architectures comprenant la liste des classes et leurs dépendances

— La description des algorithmes et structures de données utilisées

Il sera laissé au lecteur de ce document le soin de connaître le sujet fourni par le corps professoral. Les algorithmes et autres architectures de code déjà implémentés ne seront donc pas commentés et considérés comme compris.

Table des matières

1 Architecture

1.1 Environnements

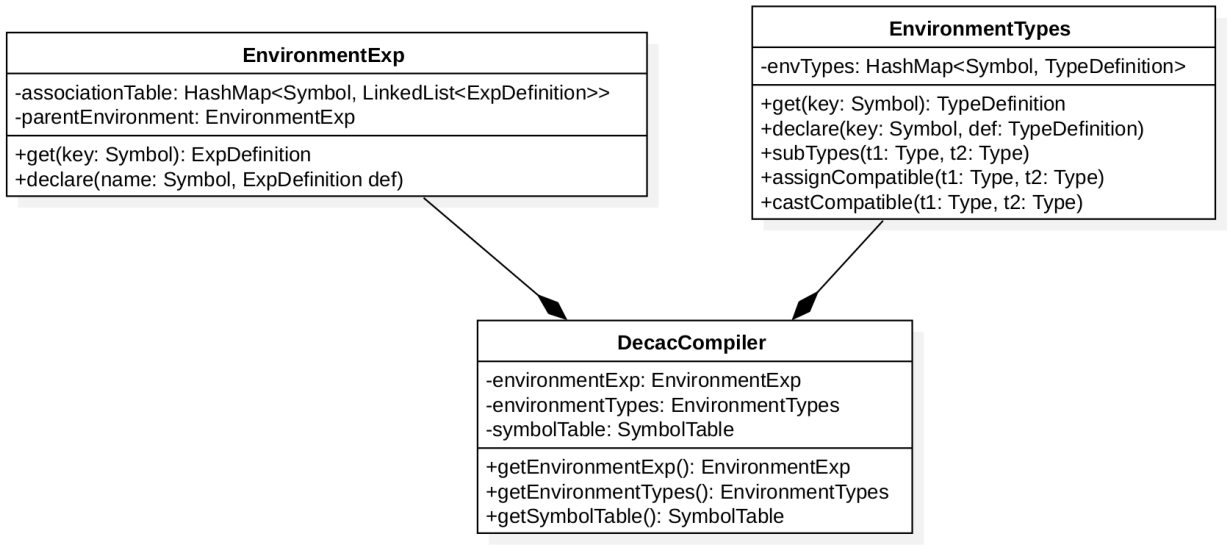

Figure 1 – Diagramme UML des environnements

Les environnements d'identificateurs d'expressions et de types sont instanciés par `DecacCompiler`. Ainsi, étant donné qu'une instance de `DecacCompiler` est transmise à travers toutes les méthodes utilisées dans le package `tree`, il est aisé d'accéder à ces deux environnements tout au long de l'enrichissement de l'arbre. Lors de l'initialisation de l'objet `DecacCompiler`, pour chaque fichier Deca, les environnements sont agrémentés des types et méthodes prédéfinis, c'est-à-dire : `void`, `boolean`, `float`, `int`, `Object` pour les types et `equals` pour la méthode.

1.2 Génération de code

`codeGenInit`

Cette méthode, appelée par `Program`, génère le code au début du programme pour tester le débordement de la pile, et ajoute à la taille de la pile le nombre de variables globales.

`codeGenMethodTable`

Passe 1 : génération de la table des méthodes
Cette méthode, appelée par `Program`, génère le code de la méthode `Object.equals`, le tableau des étiquettes de méthodes, et le code de la table des méthodes. Elle est implémentée par les quatre classes suivantes :

`ListDeclClass` : Initialise la méthode `Object.equals` en la plaçant dans la pile. Appelle la méthode de même nom sur chaque `DeclClass` de la liste.

`DeclClass` : Alloue une place dans la pile, pour une adresse mémoire qui pointe vers la classe mère. Appelle la méthode de même nom dans `ListDeclMethod`.

`ListDeclMethod` : Initialise le tableau des étiquettes de méthodes de la classe actuelle en y ajoutant les méthodes de la classe mère, puis complète le tableau par les méthodes de la classe actuelle, et enfin alloue une place dans la pile pour chaque méthode. Appelle la méthode de même nom sur chaque `DeclMethod` de la liste.

`DeclMethod` : Définit l'étiquette de la méthode et complète le tableau des étiquettes.

codeGenListDeclClass

Passe 2 : initialisation des classes et code des méthodes
Cette méthode, appelée par `Program`, génère le code pour l'initialisation des classes et pour le contenu des méthodes. Elle génère le code de la méthode `Object.equals` puis appelle la méthode `codeGenDeclClass` sur chaque `DeclClass` de la liste.

codeGenDeclClass

Cette méthode génère le code pour l'initialisation de la classe et pour le contenu de ses méthodes. Initialise les champs, puis les méthodes comme indiqué dans le polycopié, avec les méthodes `codeGenListDeclField` et `codeGenListDeclMethod`.

codeGenMain

Passe 2 : codage du programme principal
Cette méthode, appelée par `Program`, génère le code du programme principal tout en appelant les méthodes `codeGenListDeclVar` et `codeGenListInst`.

codeGenListDeclVar

`ListDeclVar` : Génère le code de la liste des déclarations. Appelle la méthode `codeGenDeclVar` sur chaque `AbstractDeclVar` de la liste.

codeGenDeclVar

`DeclVar` : Alloue une place dans la pile pour la variable, enregistre l'adresse dans la pile dans l'environnement actuel (global ou local), puis génère le code qui enregistrera dans cette espace mémoire la variable avec la méthode `codeGenInit`

codeGenInit

`Initialization` : Génère le code qui calcule l'expression dans le registre R_2 avec la méthode `codeGenExpr`, puis sauvegarde ce registre dans l'espace mémoire donné en paramètre.
`NoInitialization` : Ne génère aucun code, l'espace mémoire donné en paramètre reste vide

codeGenListInst

`ListInst` : Génère le code de la liste des instructions. Appelle la méthode `codeGenInst` sur chaque `AbstractInst` de la liste.

codeGenInst

Par défaut, cette méthode génère l'expression dans le registre R_2. Cela permet à l'utilisateur d'écrire des instructions qui ont un effet de bord, et au programme de lever des erreurs par exemple sur des expressions arithmétiques.
Elle est redéfinie par la plupart des instructions, en suivant le comportement spécifique à la classe décrit dans le polycopié (par exemple, `MethodCall` qui réalise l'appel de méthode).
`Assign` : Génère le code de l'expression à droite dans R_2, et sauvegarde celle-ci avec la méthode `codeGenStore` de la classe `AbstractLValue`.

codeGenStore

Cette méthode permet de sauvegarder une expression dans une `AbstractLValue`. Elle sert aux classes `Identifier` et `Selection`.

codeGenExprBool

Cette méthode génère le code qui effectue un branchement vers une étiquette donnée en paramètre si le booléen donné en paramètre est identique à l'expression booléenne évaluée. Elle prend un entier n en paramètre, signifiant que le registre R_n sera utilisé si un calcul d'expression est nécessaire.
Elle est redéfinit par la plupart des expressions qui peuvent avoir le type `boolean`, pour implémenter le comportement décrit ci-dessus.
Par défaut, si l'expression possède déjà une valeur stockée en mémoire, la méthode compare cette valeur à l'entier 0.
L'algorithme utilisé pour le codage des structures de contrôle est celui détaillé dans le polycopié.

`codeGenExpr`
Cette méthode permet de charger la valeur de l'expression dans le registre R_n, où n est un entier donné en paramètre. Elle est redéfinie par la plupart des expressions, pour implémenter le comportement décrit ci-dessus.

`dval`
Cette méthode permet de renvoyer une opérande qui contient une valeur associée à une expression. Cette valeur diffère selon la classe appelante : dans le cas général, elle renvoie `null` pour préciser qu'aucune valeur n'est associée. Pour un littéral entier, flottant ou booléen, elle renvoie la valeur du littéral. Pour un identifiant, elle renvoie son adresse mémoire. Pour le mot-clé `this`, elle renvoie l'adresse $-2(LB)$.

`codeGenError`
Cette méthode, appelée par `Program`, génère les étiquettes d'erreurs utilisées pendant le programme (à l'exception de l'erreur de retour d'une méthode, qui est spécifique à chaque méthode).

2 Algorithmique

2.1 Algorithmes

2.1.1 Sous-classes

Une méthode implémentée dans la classe `ClassType` permet de savoir si une classe est une sous-classe d'une autre classe. Le prototype de la méthode est le suivant :

```
1 public boolean isSubClassOf(ClassType potentialSuperClass);
```

L'algorithme implémenté est le suivant :

1. La définition de la classe courante est récupérée
2. **Tant que** la définition de classe courante n'est pas nulle :
   (a) Si la définition de la classe courante est égale à la la définition de la potentielle classe mère
   (b) $\rightarrow$ Renvoie **vrai**
   (c) La définition de la classe courante devient celle de sa propre classe mère
3. $\rightarrow$ Renvoie **faux**

Pour tester l'égalité entre deux définitions de classe, la méthode `equals()` de la classe `ClassDefinition` a été redéfinie. Pour deux classes distinctes, leurs définitions sont égales si et seulement si leurs noms sont égaux. En effet, le test d'égalité du nom est suffisant du fait de l'unicité des symboles de classe au sein d'un programme Deca.

2.2 Structures de données

2.2.1 Environnement d'identificateurs d'expressions

Afin de représenter l'environnement d'identificateur d'expressions, il est nécessaire d'avoir une structure de données représentant une association nom $\rightarrow$ définition avec possibilité d'empilement. Dans un soucis d'efficacité, nous avons choisi d'utiliser une structure permettant une recherche d'éléments en $\mathcal{O}(1)$. Nous avons donc utilisé un `HashMap` ayant pour clé le nom, de type `Symbol`, et pour valeur une liste chaînée `LinkedList` contenant l'empilement des définitions.

Ainsi, lors de la recherche d'un symbole, la définition de l'expression de ce dernier est retourné en temps constant : le premier élément de la liste chaînée est retourné. Lors de l'insertion d'un nouveau symbole, si ce dernier est déjà présent dans l'environnement parent, la liste chaînée correspondante est alors récupérée. La nouvelle définition est ajoutée en tête de cette liste chaînée et cette dernière est associée au symbole dans l'environnement courant comme

demandé.

Enfin, une méthode supplémentaire a été implémentée pour l'environnement d'identificateurs d'expressions. Cette méthode, `addSuperExpDefinition(EnvironmentExp superEnvironmentExp)`, permet de récupérer l'ensemble des éléments de l'environnement parent passé en paramètre. Ces identificateurs d'expressions sont ensuite ajoutés directement dans l'environnement courant si ces derniers ne sont pas déjà présents.

2.2.2 Environnement de types

De la même façon que pour l'environnement d'identificateurs d'expressions, il est nécessaire d'avoir une structure de données représentant une association nom $\rightarrow$ définition. Néanmoins, la notion d'empilement n'étant pas présente, nous avons donc fait le choix d'utiliser un `HashMap` ayant pour clé le nom, de type `Symbol`, et pour valeur la définition du type. Les complexités algorithmiques pour la recherche et l'insertion sont en temps constant en $\mathcal{O}(1)$.

2.2.3 Tableau des étiquettes de méthodes

Un tableau des étiquettes de méthodes est associé à chaque classe. L'intérêt est de pouvoir hériter des méthodes de la classe mère avec un coût linéaire en nombre de méthodes héritées, de les écraser lorsqu'une méthode est redéfinie dans la classe fille avec un coût constant, puis de les parcourir avec un coût linéaire pour construire la table des méthodes.
Ce tableau est défini dans l'attribut privé `labelArrayList` dans la classe `ClassDefinition`. Il est notamment utilisé dans la passe 1 de la génération du code dans les classes `ListDeclMethod` et `DeclMethod`.

2.2.4 Génération d'étiquettes

Les étiquettes étant uniques et ne prenant pas la casse en compte, un outil est nécessaire pour leur création. C'est la classe `LabelGenerator` qui gère l'unicité des étiquettes en y rajoutant un suffixe quand nécessaire, à l'aide de deux `HashMap` - l'un qui prend en compte la casse, l'autre qui met en minuscule. De cette façon les ajouts, les accès et la recherche d'étiquettes sont en $\mathcal{O}(1)$. Elle gère aussi les étiquettes d'erreur (`getOverflowLabel`, `getIoLabel`...) pour permettre de ne générer que celles utilisées pendant le programme, afin de ne pas surcharger ce dernier par du code inutile.