Projet TDL : Compimlateur Micro Java

Casanova Guillaume Leroux Frédéric Palandri Rémi

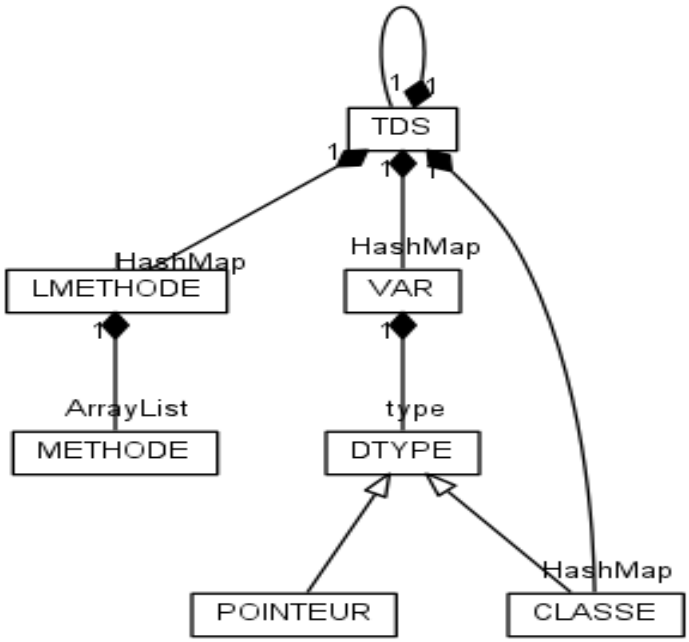
28 janvier 2012

Résumé

Table des matières

1 Les classes JAVA

1.1 Diagramme UML



1.2 La TDS

La TDS que nous avons choisie est une variation de celle qui nous a été donnée en TP de TDL. En effet, au lieu d'hériter d'une HashMap<String,INFO>, elle contient en tant qu'attributs 3 HashMaps de variables, méthodes et classes respectivement, comme il est visible sur le graphe UML. Le principe même d'une HashMap est d'être une bijection entre la clef de type String et la Valeur de type INFO. Or,il est possible en MJ de donner le même nom à une variable et une méthode, et le même nom à un classe qu'à un constructeur. Par son principe même une seule HashMap ne peut pas gérer cela. Ainsi, nos fonctions chercherGlobalement existent en trois exemplaires : pour les variables, les classes et les méthodes.

Afin d'autoriser la surcharge de méthode, la fonction chercherGlobalementMethod prend aussi en argument les arguments auquels la méthode doit corresponde pour être considérée comme acceptable. La HashMap de

méthodes est en effet, pour des soucis d'optimisation, contient des LMETHODES, une classe de liste de méthodes ayant le même nom (mais des arguments différents ; et il n'y a pas de surcharge du type de retour). Ainsi, chercherGlobalementMethode(nom,arguments) trouve donc d'abord dans la HashMap de la TDS la liste de méthodes ayant ce nom, puis trouve dans cette liste une méthode avec les bons arguments. Ce système nous permet de déclarer des constructeurs qui ont le même nom que leur classe associé, ainsi que de déclarer un attribut ayant le même nom que sa classe.

Nous avons de plus conservé le système de DTYPE ; le type POINTEUR étant une classe fille qui contient également le DTYPE pointé.

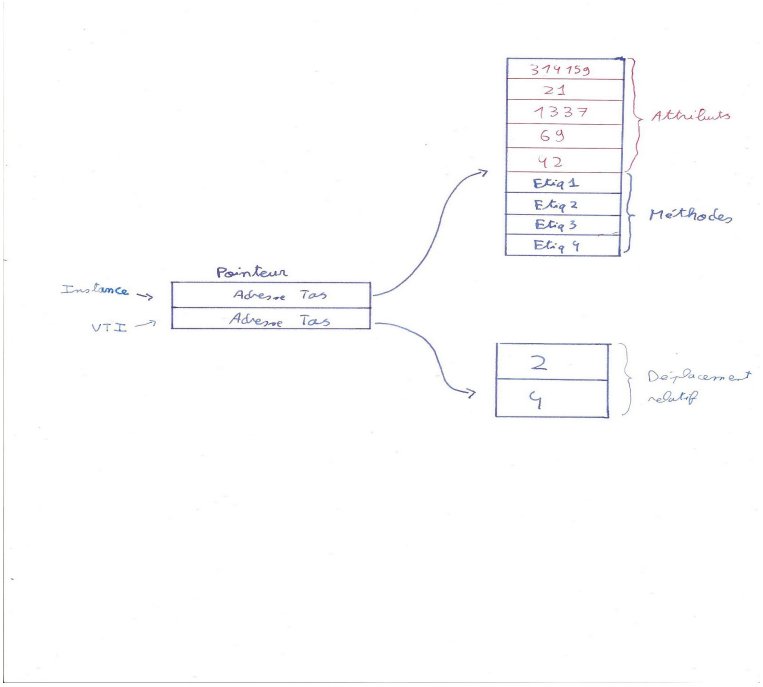
Un des problèmes que nous avons rencontré est celui du typage de MicroJava créé par la distinction entre type réel et type apparent. En effet, il est possible pour un exemple contenant une classe P ayant comme classe fille PC d'avoir une définition :

```
P p = new PC();
```

Ici, PC à pour type "Pointeur pointant vers PC" et P a pour type "Pointeur pointant vers P", mais l'affectation était alors toujours valide. Nous avons donc développé une fonction canAccept pour les DTYPE. Elle permet de déterminer si l'affectation est valide bien que les types ne soient pas égaux. C'est le cas si le pointur est nul. En effet, pour chaque classe, nous avons défini une liste (générée lors de la 2ème passe) contenant de classes susceptibles d'être Super-Classes lors d'une affectation. Cette fonction est par exemple utilisée dans ARGLIST (les listes d'argumentsde fonctions) pour autoriser le passage d'un PC en tant qu'argument dont le type était P.

La classe METHODE contient les informations des méthodes et permet de générer les vtables. Pour chaque méthode Cette classe contient une étiquette, positionnée au début du code de cette méthode. On peut ainsi générer son entrée vtable ainsi qu'un numéro qui lui est associé. En effet, chaque méthode de classe est numérotée de 0 à nb_methode-1, ce qui permet pour l'appel des méthodes de savoir dans quelle position elle est dans la vtable (plus de détail sur la liasion tardive et les interface partie partie 2).

2 Laison tardive



Le mécanisme de la liaison tardive que nous avons développé est basé sur l'utilisation d'un pointeur de taille 2. Le premier pointe vers l'instance de l'objet (attributs plus VTABLE contenant les méthodes), le second pointe vers ce que nous appelons une VTI (vtable d'interface), ce qui nous permet de gérer le pattern java où une classe, par héritage, peut implémenter de multiples interfaces n'étant pas héritées les unes des autres (voir schéma explicatif du pattern d'héritage multiple). Ce mécanisme est schématisé dans l'image ci-dessus.

Premier pointeur vers l'instance de l'objet :
Le premier pointeur est nécessaire dans tous les cas de pointeurs (contrairement au 2ème, uniquement utile dans le cas ou le type apparent est une interface). Il contient 2 types d'informations :

- En haut du pointeur (adresses supérieures),les attributs.
- En bas du pointeur, en déplacement négatif, les méthodes.

Les méthodes sont choisies à l'instantiation de l'objet en parcourant récursivement les TDS de la classe du type apparent pour déterminer quelles

méthodes seront accessibles depuis cette instance. En effet, sachant que MicroJava n'accepte pas les casts, les méthodes accessibles durant toute la vie de l'objet seront un sous-ensemble des méthodes accessibles de la classe apparente. Après la création de cette liste des méthodes accessibles provenant de la classe apparente, notre algorithme parcourt les méthodes de la classe réelle pour déterminer quelles méthodes appeller (si la méthode est redéfinie dans la classe réelle, celle de la classe réelle sera choisie). Les étiquettes des méthodes trouvées sont insérées selon le numéro de la méthode (de 0 à nb_methode-1) dans la VTABLE située en dessous du premier pointeur (en bleu dans le dessin). Dans l'ordonnancement de la VTABLE, les premières méthodes sont toujours celles des classes mères. Par exemple, si une classe A définit une méthode getx, et une classe B extends A définit gety, une VTABLE d'un élément de type apparent B contiendra, dans cet ordre, [etiquette_getx etiquette_gety]. Ce système d'ordonnancement permet de gérer automatiquement le problème du B b = new B() ; A a = b ; En effet, dans l'opération a=b, même si la VTABLE de b contiendra des méthodes en trop pour un pointeur de type apparent A, cette VTABLE commencera bien par les mêmes méthodes, ordonnancées de la même façon qu'une vtable de A. On peut donc simplement copier le pointeur sans risque.

Deuxième pointeur vers la VTI :
Le deuxième pointeur, initialement null (0), pointe vers la VTI de l'interface si le type apparent du pointeur est une interface. Sinon, il reste durant toute sa vie à 0. Cette VTI est le système choisi pour gérer le pattern java permettant à une classe, via héritage, d'implémenter un nombre quelconque d'interfaces n'ayant aucune relation d'ordre entre elles. En effet, l'instruction A a = b ; de l'explication du premier pointeur fonctionne grâce à la relation d'ordre existant entre les méthodes de A et de B, sachant que B hérite de A. Cette relation d'ordre n'existe pas entre des interfaces, qui ne sont pas forcément héritées les unes des autres. Nos VTIs sont une table contenant, comme illustré dans le schéma, les déplacements relatifs des méthodes de l'interface par rapport à la VTABLE du pointeur. Considérons, pour rester dans l'exemple du schéma, une classe A ayant les méthodes getx, gety, setx et sety (dans cet ordre, 1 2 3 4, dans la VTABLE). Une interface IA, elle, ne contient que gety et sety. Alors, comme dans l'exemple, la VTI de cette interface contiendra [2, 4]. Ceci permettra, lors d'un appel ia.gety() ; de savoir la méthode à appeller. En effet, lors du ia.gety(), on remarquera que gety est la première méthode du type apparent, ia. On ira donc chercher le premier

élément de la VTI, 2. Ceci sera le numéro de la méthode de la VTABLE à appeller. Celle ci, dans la VTABLE de A, est gety. La bonne méthode sera donc appellée. La VTI n'est donc pas générée à l'instantiation de la classe, qui elle ne génère que la VTABLE (premier pointeur), mais à l'affectation d'une valeur (de type apparent classe) à une interface (ib=a) ; Ce système nous permet de gérer une implémentation d'un nombre infini d'interfaces via le motif extends/implements java avec gestion de la liaison tardive, comme notre test SuperTestI.mj le prouve. Si nous rencontrons un cas de type ib = ic (ic étant un pointeur de type apparent d'une autre interface), alors nous savons qu'il existe une relation d'ordre entre ib et ic, car IC extends IB. Alors, comme dans les VTABLEs, on ne doit pas modifier la VTI : elle fonctionnere par défaut dans la classe mère, sachant que les méthodes sont ordonnées dans cet ordre.

3 Les trois passes

Nous avons choisi dans notre projet d'effectuer 3 passes du code pour résoudre les problèmes de déclaration en avant dans le code, et aussi d'appel du constructeur d'une classe pendant la définition de celle ci (par exemple, la classe Point a une méthode milieu qui instancie un Point avant la déclaration complète de Point).
Les 3 passes ont chacune une utilité différente :

- Dans la première passe, on introduit dans la TDS toutes les classes présentes dans le programme.
- Dans la deuxième passe, on introduit dans la TDS tous les attributs et les méthodes présents dans le programme, et on gère également les extends.
- Dans la troisième passe on effectue le reste de l'analyse syntaxique, et on génère enfin le code.

Pour gérer ce système à 3 passes, nous avons programmé 3 fichiers .egg (MJAVA, MJAVA2 et MJAVA3) et modifié certains fichiers dans le répertoire compiler/. En effet, nous avons eu à changer les fichiers MJC.java et MJavaSourceFile.java pour faire passer un argument de type TDS de passe à passe.

4 Problèmes principaux rencontrés

4.1 La surcharge

On peut rencontrer deux types de surcharge :

– Surcharge entre variables attributs, méthodes et classe.
– Surcharge des méthodes.

Concernant la première surcharge l'exemple suivant doit être acceptable :

```
class x {
  int x ;
  x (int xi) {
    x = xi ;
  }
}
```

Dans l'exemple précédent, on a : la classe x, l'attribut x et le constructeur (la méthode x) de cette classe. Il ne peut pas y avoir d'ambiguïté entre ces trois objets au niveau sémantique. Afin de simplifier les insertions et les recherches dans la TDS, on a décidé de créer 3 HashMaps différentes dans chaque TDS, une pour les classes, une pour les méthodes et une pour les variables locales et attributs, comme on le voit dans la partie TDS.

La surcharge de méthode est très utilisée en Java. En particulier la surcharge de constructeur :

```
class test {
   int x ;
   int y ;
   test (int xi) {
     x = xi ;
     y = 0 ;
   }
   test (int xi, int yi) {
     x = xi ;
     y = yi ;
```

```
   }
}
```

Dans l'exemple précédent, on doit pouvoir distinguer les deux constructeurs indépendamment. Ces deux constructeurs possèdent le même nom, et ont donc la même entrée dans la HashMap des méthodes. On a résolu ce problème en donnant comme valeur à la clé correspondante à chaque nom de méthode, une liste de méthode distinguées par le nombre et le type d'arguments.

4.2 Les règles E et ER

Deux règles nous ont posés problèmes :

```
INST → return E pv
INST → INST -> si paro  E  parf BLOC SIX.
```

Ces règles sont problématiques à cause de la gestion de la règle E → ER AFFX. On ne doit pas autoriser à priori des instructions telles que :
« return (a = 2) ;» ou « if (a = 2) ... ». On a envisagé deux solutions différentes : la première est de tout simplement changer la grammaire, en modifiant ces règles pour donner :

```
INST → return ER pv
INST → INST -> si paro  ER  parf BLOC SIX,
```

ce qui aurait résolu le problème en renvoyant une erreur syntaxique.

La deuxième solution, et celle qui a été finalement implémentée, est de renvoyer une erreur analytique, calculée lors de l'évaluation de la règle

```
E → ER AFFX.
```

5 Amélioration du code

5.1 Le print

Nous avons ajouté 2 règles de grammaire basées sur le mot-clef print dans MicroJava :

```
INST → print guil ident guil pv
INST → print E pv
```

Où guil == " . La première règle imprime la String ident sur l'afficheur TAM.
Le second imprime la valeur de E sur l'afficheur TAM.

```
class Test {
  void main() {
    Exemple y = new Exemple() ;
    int z = y.getX() ;
    print z ;
  }
}
```

Ces deux nouvelles règles de grammaire permettent de tester plus aisément les programmes, sans avoir à regarder le contenu de la pile avec le déplacement des variables pour s'assurer de la bonne exécution de nos programmes.

5.2 Le this

```
F → this pv
```

Cette règle introduit l'identificateur this qui renvoit l'instanciation de la classe courante. Cependant, cette règle ne suffit pas à simuler toutes les fonctionnalités du this en Java, on ne peut notamment pas faire de this.methode() ou this.attribut . Néanmoins cela ne restreint pas les possibilités du langage vu qu'il suffit de créer une variable globale contenant la valeur du this comme ceci :

```
class Exemple {
  int x ;
  Exemple () {
    x = 0
  }
  int getX() {
    Exemple e = this ;
    return e.x ;
  }
}
```

La structure globale du .egg permettrait de gérer les règles this.methode() ou this.attribut sans trop de difficultés, comme dans la règle

`F` $\rightarrow$ `ident Q`

seul le manque de temps nous a empêché de traiter ces cas.

5.3 Le return

Return :
On vérifie pour chaque méthode non procédurale que celle ci finit bien par exécuter une instruction return (et évidement que le type du return correspond bien au type de la méthode). La subtilité pour gérer cela est de s'apercevoir que dans le cas d'une instruction « if condition bloc1 else bloc2 », si bloc1 possède un return, bloc2 doit posséder un return également.

6 Les tests