Sémantique et Traduction des Langages

Lena DAVID Sacha LIGUORI Thibault MEUNIER
Matthieu PERRIER

Mai 2017

Sommaire

1 Concepts de base

1.1 Nouveautés sur les blocks

Dans la gestion des blocs, nous avons ajouté la boucle **for**. Il s'agit d'une extension syntaxique simple de la boucle while. De même, avec l'ajout de la gestion des types **String** comme type élémentaire, l'instruction print a été enrichie d'une sous instruction : **println**. L'initialisation de tableaux à partir de séquence a aussi été repensée pour tenir compte du fait que celles-ci ne sont plus stockée sur la pile mais directement dans le tas.

1.2 Fonctions

Les fonctions sont la première amélioration majeure de MiniJava par rapport au langage Bloc.

1.2.1 Gestion de la Table des Symboles

La table des symboles a ici un usage simple : vérifier que l'identificateur passé correspond à une fonction. C'est au niveau de la génération du code que la vérification de l'existence d'une fonction correspondant aux paramètres est effectuée. Si aucune fonction ne correspond, une erreur est renvoyée. On ajoute chacun des paramètres de la fonction à la table des symboles du corps afin qu'ils puissent être utilisés par la suite.

1.2.2 Typage

Une fonction est typée comme un type fonction. Ce dernier contient le type de retour et le type des paramètres. On a également ajouté une méthode permettant d'obtenir le type de retour, afin de typer correctement les appels de fonctions.

1.2.3 Code TAM

La génération du code TAM pour les fonctions s'est séparée en plusieurs parties. La première consiste en la création d'une étiquette spécifique. A la déclaration, on alloue un offset unique a chacun des paramètres. Ils sont ainsi traités comme des variables simples. La gestion du **return** nécessite la connaissance de la taille totale des paramètres. On passe donc cette taille pour effectuer le return quand bon nous semble au cours de la fonction. Bien sûr, on empile la valeur de retour avant de placer l'instruction **return**.

2 Classes et Interfaces

Classes et Interfaces sont les concepts au centre de ce projet. Ils nous ont amenés à réfléchir sur la gestion des variables, le cast, et l'appel de fonctions.

2.1 Eléments et droit d'accès

Les éléments sont tous gérés comme des déclarations, avec l'ajout d'une valeur spéciale : NoValue. Dès lors, une variable non initialisée possède cette valeur. La gestion des conflits et de la surcharge d'attributs s'effectue au niveau de la table des symboles. Si un symbole est déjà présent dans le même niveau de la table, on renvoie une erreur. Sinon, on l'ajoute et il masquera la déclaration précédente. La gestion des droits d'accès s'effectue directement à la compilation. Si un élément (attribut ou méthode) est utlisé illégalement, on renvoie une erreur associée.

2.2 Statique et Objet Classe

Au moment de la génération du code associée a la classe, on procède en deux temps. Tout d'abord, on gère les méthodes et attributs statiques. Ceux-ci sont gérés comme des objets normaux dont l'identificateur est le nom de la classe. Ensuite vient la déclaration de la table des symboles, mais nous y reviendrons plus tard.

2.3 Instance de Classe

Si il est un objectif des langages objets incontournables, il s'agit bien sûr de celui-ci : la création et manipulation d'instances de classes. A la création, on alloue tout d'abord la taille nécessaire au stockage de l'objet, puis on appelle un constructeur (processus décrit plus en détail ci-dessous) avec le pointeur fourni. Ce dernier procèdera également à la création du lien vers la table des méthodes. Lors de la manipulation, on peut modifier les attributs qui ont un offset fixe facilement, et on accède au méthode par l'intermédiaire de la table. Le cast sur un type interface se fait par la création d'un pointeur sur l'objet original et vers la table de méthode associée. On peut alors procéder comme précedemment pour manipuler les objets.

2.4 Constructeurs

Les constructeurs sont gérés comme des fonctions d'un type particulier (`ConstructorType`). Lors de la génération de code, on recherche parmi les éléments de classe un constructeur dont les types des paramètres avec lesquels les types des paramètres qu'on passe au constructeur sont compatibles. Si un tel constructeur est trouvé, on l'utilise. Si aucun constructeur convenable n'est trouvé, l'objet est initialisé à l'aide d'un constructeur par défaut, qui alloue l'espace nécessaire pour l'objet (c'est-à-dire sa taille).

2.5 Surcharge

Concernant la surcharge, la majeure partie du travail a consisté à écrire une méthode Java permettant de rechercher une fonction selon son nom et les types des paramètres. Cette méthode prend donc en paramètre une chaîne de caractères correspondant au nom de la fonction, et une liste de type correspondant aux types des paramètres. Elle recherche d'abord

les fonctions ayant le bon nom parmi celles disponibles, et, parmi ces fonctions, en recherche une pour laquelle les types de paramètres passés à la méthode sont compatibles avec les types attendus par la fonction.

Cette méthode est utilisée dans le cadre de la génération de code.

3 Héritage

La notion d'héritage est inévitable lorsque l'on introduit les classes et les interfaces. Son implémentation a demandé une réflexion particulière sur la gestion des éléments hérités, la redéfinition de variable ou de méthode, et a mené a l'introduction d'une table des méthodes virtuelles.

3.1 Construction de la table de méthodes virtuelles

Pour construire la table des méthodes virtuelles, on commence tout d'abord par trier les fonctions à mettre dedans selon l'ordre décrit ci-après. Ensuite, on regroupe les méthodes provenant d'un héritage d'interface sous un même offset, qui pointera vers une sous-table de méthode associée à l'interface. Il ne reste plus qu'à stocker les pointeurs vers les différentes méthodes pour finaliser la création [1].

3.2 Ordre des éléments dans la table des méthodes

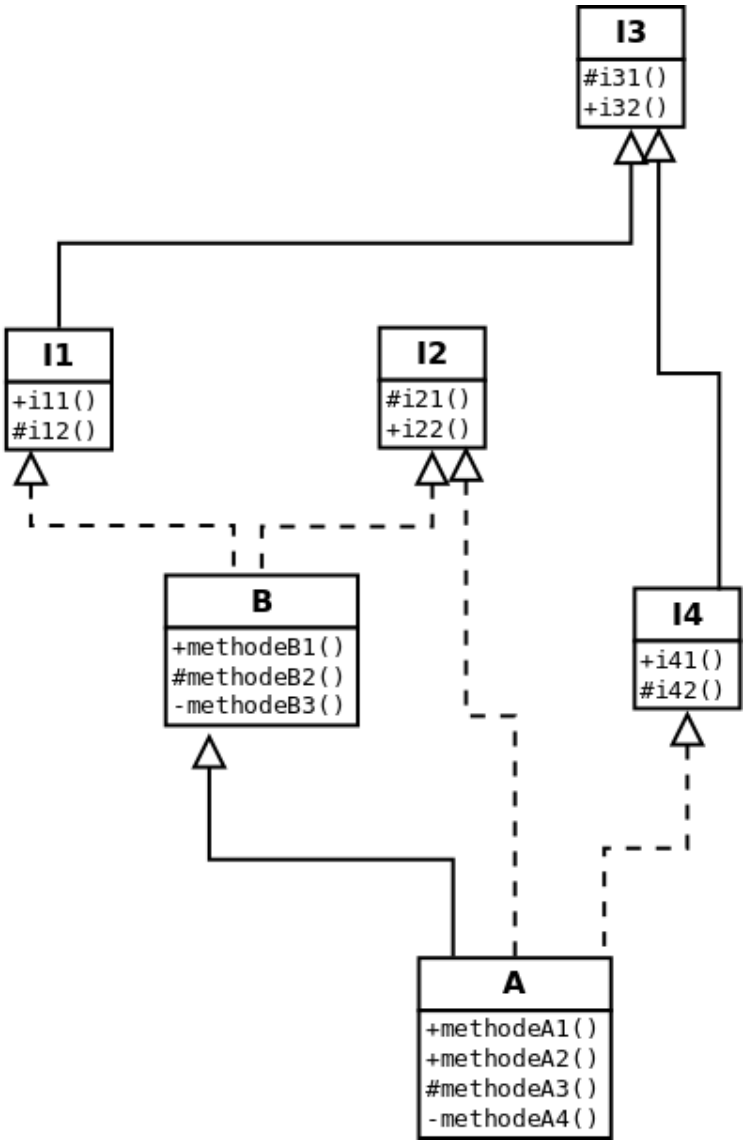
Les fonctions sont triées dans l'ordre suivant :

1. Les fonctions hérités.
2. Les fonctions des interfaces implémentées
3. les fonctions publiques propres à la classe
4. les fonctions protégées propres à la classe
5. les fonctions privées propres à la classe

Si une interface est implémentée 2 fois, alors les pointeurs vers ses fonction apparaîtront 2 fois.

En pratique, utilisons l'exemple suivant pour mieux cerné cet ordre. Il contient deux classes (A et B) et 4 interfaces (I1 à I4). On construit ici la table des méthodes virtuelle de la classe A.

1. Virtual method table - Wikipedia


I3
#i31()
+i32()
I1
+i11()
#i12()
I2
#i21()
+i22()
B
+methodeB1()
#methodeB2()
-methodeB3()
I4
+i41()
#i42()
A
+methodeA1()
+methodeA2()
#methodeA3()
-methodeA4()

La table des méthodes obtenue est la suivante :

		Table des méthodes virtuelles
Elements hérités de B	I1	i11 i12
	I3	i32 i31
	I2	i22 i21
	B	methodeB1 methodeB2
Implémentation des interfaces	I2	i22 i21
	I4	i41 i42
	I3	i32 i31
Fonctions propres à la classe	A	methodeA1 methodeA2 methodeA3 methodeA4

On peut noter ici que les interfaces I2 et I3 sont implémentées deux fois leur méthodes apparaissent donc deux fois. On remarque également que la méthode `methodeB3`, qui est privée n'est pas héritée. Notons également que l'ordre des méthodes dépend d'abord du modificateur, puis de l'ordre de déclaration, ceci est particulièrement visible au niveau des interfaces I2 et I3.

4 Genericite

La généricité consiste à générer du code pouvant travailler avec tous les types de données.Ainsi une classe ou une fonction peut être conçue en faisant abstraction du type de données manipulé.

4.1 Les types

Une classe *GenericType* étendant *Type* a été crée afin de gérer l'instanciation des types. La construction se fait à partir d'un type déclaré et d'une liste de ces même types génériques. L'ensemble des identificateur de type générique sont enregistrés dans une table des symboles associée à *ArgumentGenericite*. Dans le cadre de notre projet la généricité porte sur des types atomiques tels que *bool* ou *int*, l'étendre aux objets aurait permis de factoriser du code et de réaliser la génération de code TAM plus facilement. Ces types se presentent sous la forme *<typeGenerique1, typeGenerique2>*

4.2 Les paramètres

Le paramètre générique possède son type associé *GenericParameterType* permettant de typer les paramètres génériques lors de déclaration de classes, méthodes ou interfaces.Cette classe de typage des paramètre permet une vérification des la compatibilité du type générique. Une table des symboles est associée à la suite de paramètre génericité afin de pouvoir les instancier dans la suite de la classe.

4.3 L'héritage

L'héritage au seins de la génricité est géré au sein même de *GenericParameterImpl*, il s'agit d'une liste de types génériques hérités. Une table des symbole dédiée est crée afin de faire suivre les types générique lors de l'héritage.

Conclusion

Un projet ambitieux durant lequel nous avons su faire preuve d'organisation, être quatre signifie qu'il faut être quatre fois plus rigoureux car la moindre erreur peut se répercuter sur le travail des trois autres. Nous avons pu expérimenter des outils tels que Travis, permettant le lancement de tests automatiques, que nous aurons certainement l'occasion d'utiliser dans nos projets futurs. Quelle satisfaction d'arriver à concevoir un petit compilateur, et qui nous a permis de nous rendre compte qu'il s'agissait d'une tâche longue mais réalisable.