UNIVERSITÉ DE ROUEN

Compilateur LLVM

Langage jouet Kawa

Document Technique - Analyse sémantique

0.1.4

17 avril 2015

Auteur(s): Pierre-Luc BLOT, Alexandre PETRE

Version	Date	Changelog
0.1	29/03/2015	Version initiale.
0.1.1	31/03/2015	Table des symboles : table de hachage + pile
0.1.2	02/04/2015	Interface c++ de la table des symboles
0.1.3	02/04/2015	Objectifs de l'analyseur sémantique
0.1.4	06/04/2015	Exemples table des symboles
0.1.4	06/04/2015	Étapes de l'analyse sémantique

Table des matières

1 Pré-requis

La phase d'analyse sémantique arrive seulement après un retour positif de l'analyse syntaxique. Les fichiers sources écrits en KAWA ont été parsé et aucune erreur de syntaxe n'a été décelée. C'est à ce moment que l'analyseur sémantique est appelé. Afin de vérifier si ce qui a été écrit à du sens. Il n'appartient pas à l'analyseur sémantique de vérifier si des éléments sont syntaxiquement corrects. Pour en savoir plus sur l'analyse syntaxique, vous êtes invitez à consulter le document spécifiant sont fonctionnement.
Afin d'effectuer les analyses durant cette phase, l'analyseur sémantique procède à une décoration de l'arbre syntaxique (voir KawaTree) afin de déterminer le type des expressions qui seront traitées par le backend. Plus généralement, l'analyseur sémantique décore puis vérifie l'arbre syntaxique pour produire l'arbre sémantique. Il s'avère que cet arbre est représenté par la même structure de données nommée KawaTree. Le rôle fondamental de l'analyseur sémantique est, en se basant sur les données du KawaTree fournies par l'analyseur syntaxique, de produire et d'écrire des données supplémentaires dans KawaTree tout en vérifiant leurs sens. KawaTree sera ensuite transmis au backend pour la production de code IR LLVM.

2 Objectifs de l'analyseur sémantique

Afin de bien mettre en évidence les objectifs de l'analyse sémantique, voici quelques exemples qui permettront de comprendre ses enjeux. À ce niveau, dans ces exemples, un **programme correct** est un programme dont la **sémantique est correcte**. Tandis qu'un **programme incorrect** est un programme dont la **sémantique est incorrecte** (seule la syntaxe l'est).

2.1 Déclarer une variable une seule fois par scope

L'analyse lexicale ne fait pas la différence entre un identificateur de variable et de fonction car il s'agit du même token.

Programme correct

```
int a;
int b;
a = 1;
```

La variable **a** et la variable **b** peuvent être utilisées car elles ont été déclarées.

Programme incorrect

```
int a;
int a;
a = 1;
```

La variable **a** ne peut pas être déclarée plusieurs fois (dans le même scope).

2.2 Appeler une variable déjà déclarée

L'analyse syntaxique ne fait pas de lien entre les déclarations et les appels qui les utilisent.

Programme correct

```
int a;
a = 1;
```

La variable **a** peut être utilisée car elle a été déclarée.

Programme incorrect

```
int a;
b = 1;
```

La variable **b** ne peut pas être utilisée car elle n'a pas été déclarée.

Programme incorrect

```
b = 1;
```

La variable **a** ne peut pas être utilisée car elle n'a pas été déclarée.

3 Table des symboles

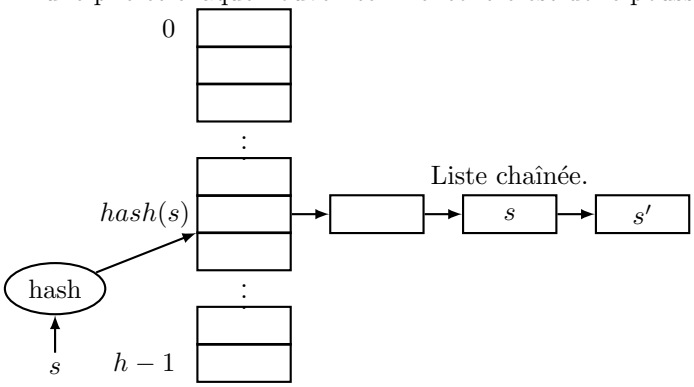
La table des symboles permet de gérer les différents symboles qui seront analysés. Cette partie permet d'établir la logique de cette table des symboles. Afin d'aider à son implémentation, une interface est détaillée en fin de partie.

3.1 Fonctionnement de la table des symboles

Il est nécessaire lors de l'analyse sémantique de lire et d'écrire des symboles dans la table. Cette table permet aussi de répondre aux besoins qui sont soulevés par la notion d'accessibilité de certains symboles au sein d'un même scope.

Deux types de structures de données sont utilisées. Une table de hachage dont les valeurs sont des références vers les instances représentant des symboles. Le figure ci-dessous illustre cette structure de données. Dans cet exemple, on illustre l'insertion d'un symbole dont le nom est stocké dans une variable s. Le hash est calculé en fonction de s afin de déterminer sur quelle liste sera ajouté le symbol.

En ce qui concerne la table de hachage, nous allons nous servir d'une Map¡String, List¡SemantiquePtr*¿¿
— La clé (String) correspond au nom de chacun des symboles rencontrés lors du parcours.
— La liste d'items correspond aux références de ces symboles. On utilise une liste pour pallier au fait que certains éléments puissent être masqués au sein de différents blocs. Cette liste fonctionne comme une pile et chaque nouvel item rencontré est donc poussé en tête de listes.



3.1.1 Comment fonctionne la pile avec la table de hachage ?

Les blocs sont imbriqués les uns dans les autres. Lorsque des déclarations sont faîtes dans des blocs imbriqués il arrive que des symboles masquent des symboles déclararés dans des blocs parents.
Quand dans un bloc un symbol est utilisé il fait référence à la première déclaration qui en a été faite. C'est à ce niveau que la pile est importante.

Les niveaux de la pile représentent les niveaux d'imbrications. Dans cette pile, chacuns des niveaux ont une liste de symboles. C'est symboles sont représentés par leurs noms. Ainsi, grâce à la table de hachage il est aisé de récupérer l'instance correspondant au symbol en question.

Lors de l'analyse, lorsqu'un bloc a été entièrement analysé, les déclarations de symboles qui y ont été réalisés deviennent obsolètes pour la suite de l'analyse. Il est nécessaire de faire du nettoyage dans la pile.

Cette étape consiste à dépiler le bloc courant de la pile. Pour ce faire, le symbol en tête de liste pointée par ce bloc (celui le plus haut dans la pile) est supprimé. Cela doit être répété afin de vider la liste. La suppression est rapide puisque qu'il suffit de supprimer la tete de liste à chaque fois.
Une fois la liste vide, le bloc peut être dépilé.

3.2 Exemple d'analyse sur un programme simple

```
{ // bloc 1
  int a = 2;
  int b = a + 1; // b=3
 { // bloc 2
   int a = b + 2; // a=5
   { // bloc 3
     int b = a + 3; // b=8
     int c = a + b; // c=13
   }
   int c = a + b; // c=8
 }
 int c = a + b; // c=5
}
```

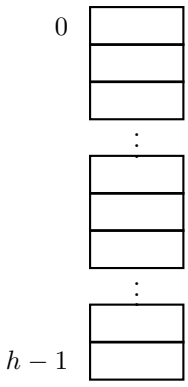
Ce programme réalise différents appels et et différentes déclarations au sein de plusieurs blocs imbriqués. Nous remarquons également que certaines variables sont masquées. Les commentaires nous aides à comprendre l'effet que cela implique sur le programme puisque l'expression a + b n'a pas la même valeur en fonction du bloc dans lequel elle est évaluée. Car ils contiennent des déclarations de a et de b qui masquent les précédentes.

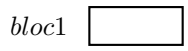
Déroulons pas à pas l'analyse des symboles.

3.2.1 Première étape

```
{ // bloc 1 <-----
  int a = 2; // @ref1
  int b = a + 1; // @ref2
 { // bloc 2
   int a = b + 2; // @ref3
   { // bloc 3
     int b = a + 3; // @ref4
     int c = a + b; // @ref5
   }
   int c = a + b; // @ref6
 }
 int c = a + b; // @ref7
}
```

L'analyse commence par le bloc 1. La table de hachage est vide et la pile également.

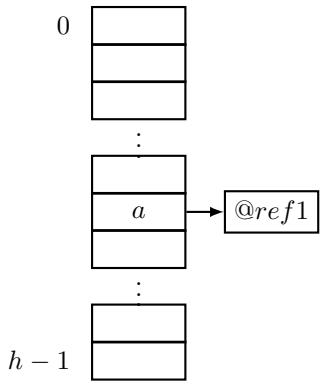


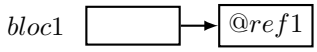


3.2.2 Deuxième étape

```
{ // bloc 1
  int a = 2; // @ref1<-----
  int b = a + 1; // @ref2
 { // bloc 2
   int a = b + 2; // @ref3
   { // bloc 3
     int b = a + 3; // @ref4
     int c = a + b; // @ref5
   }
   int c = a + b; // @ref6
 }
 int c = a + b; // @ref7
}
```

Bloc 1 :
Déclaration de variable : int a = 2 ;

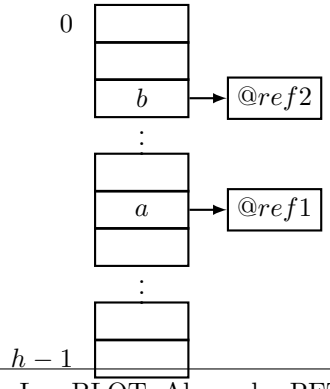


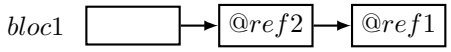


3.2.3 Troisième étape

```
{ // bloc 1
  int a = 2; //@ref1
  int b = a + 1; // @ref2 <-----
 { // bloc 2
   int a = b + 2; // @ref3
   { // bloc 3
     int b = a + 3; // @ref4
     int c = a + b; // @ref5
   }
   int c = a + b; // @ref6
 }
 int c = a + b; // @ref7
}
```

Bloc 1 :
Déclaration de variable : int b = a + 1 ;

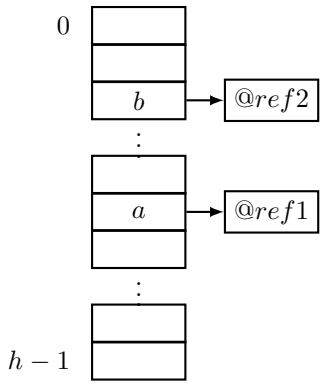


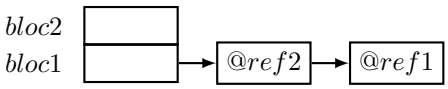


3.2.4 Quatrième étape

```
{ // bloc 1
  int a = 2; // @ref1
  int b = a + 1; // @ref2
 { // bloc 2 <-----
   int a = b + 2; // @ref3
   { // bloc 3
     int b = a + 3; // @ref4
     int c = a + b; // @ref5
   }
   int c = a + b; // @ref6
 }
 int c = a + b; // @ref7
}
```

Bloc 2 : Création d'une liste vide sur la pile

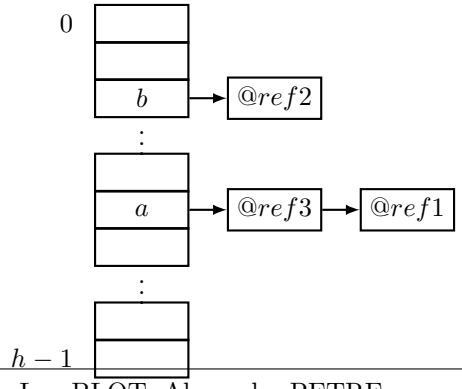


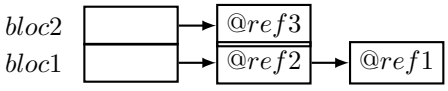


3.2.5 Cinquième étape

```
{ // bloc 1
  int a = 2; // @ref1
  int b = a + 1; // @ref2
 { // bloc 2
   int a = b + 2; // @ref3 <-----
   { // bloc 3
     int b = a + 3; // @ref4
     int c = a + b; // @ref5
   }
   int c = a + b; // @ref6
 }
 int c = a + b; // @ref7
}
```

Déclaration de a qui masque celle précendente.

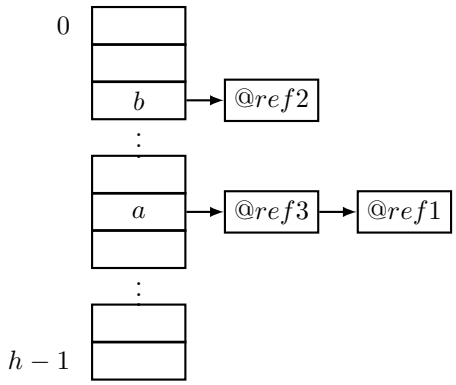


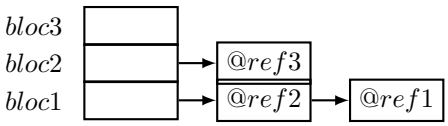


3.2.6 Sixième étape

```
{ // bloc 1
  int a = 2; // @ref1
  int b = a + 1; // @ref2
 { // bloc 2
   int a = b + 2; // @ref3
   { // bloc 3 <-----
     int b = a + 3; // @ref4
     int c = a + b; // @ref5
   }
   int c = a + b; // @ref6
 }
 int c = a + b; // @ref7
}
```

Création d'une liste vide sur la pile pour le bloc3



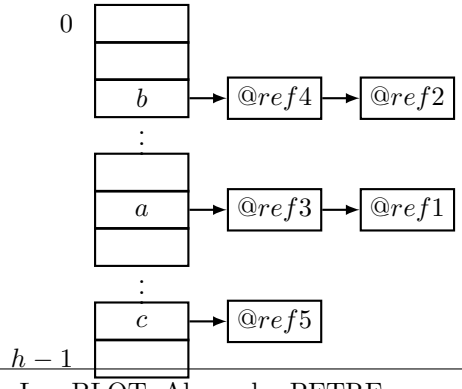


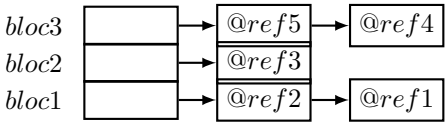
3.2.7 Septième étape

```
{ // bloc 1
  int a = 2; // @ref1
  int b = a + 1; // @ref2
 { // bloc 2
   int a = b + 2; // @ref3
   { // bloc 3
     int b = a + 3; // @ref4 <-----
     int c = a + b; // @ref5 <-----
   }
   int c = a + b; // @ref6
 }
 int c = a + b; // @ref7
}
```

Puisque le fonctionnement est similaire aux étapes précendentes, nous avons condensé les deux étapes une seule sur le schéma.

Déclaration masquante de b et déclaration locale de c.

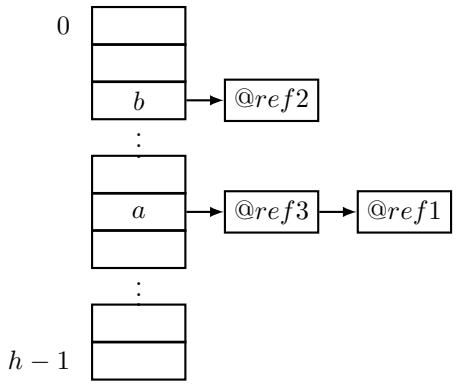


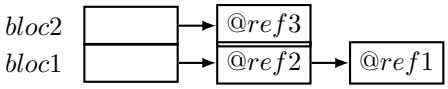


3.2.8 Huitième étape

```
{ // bloc 1
  int a = 2; // @ref1
  int b = a + 1; // @ref2
 { // bloc 2
   int a = b + 2; // @ref3
   { // bloc 3
     int b = a + 3; // @ref4
     int c = a + b; // @ref5
   } <-----
   int c = a + b; // @ref6
 }
 int c = a + b; // @ref7
}
```

Ici on sort du bloc3 donc on va dépiler le bloc3 de la pile.
On récupère la tete de liste du bloc courant (bloc 3) qui est la référence vers un élément de la table de hachage. Ensuite on supprime cet élément de la table de hachage. On supprime également cet élément de la liste du bloc courant.
Quand la liste est vide alors on dépile le bloc3 de la pile.





4 Interface de la table des symboles

```
/**
 * Interface ITableOfSymbol.
 * Specifie les operations sur la table des symboles.
 */
class ITableOfSymbol{public:
  // Requete

  /**
   * Retourne l'instance representant le symbol en fonction de son nom.
   * @param sym Nom du symbol.
   */
  virtual SemanticPtr* getSymbol(String sym) = 0;

  // Commandes

  /**
   * Place le curseur interne sur un nouveau bloc.
   * Toutes les futures operations se feront sur ce bloc.
   */
  virtual void enterBloc() = 0;

  /**
   * Deplace le curseur interne sur le bloc precedent.
   * Toutes les futures operations se feront sur ce bloc.
   */
  virtual void exitBloc() = 0;

  /**
   * Ajoute une definition de symbol dans la table des symboles.
   * @param sym Nom du symbol
   * @param sptr pointeur sur l'instance qui represente le symbol.
   */
  virtual void addSymbol(String sym, SemanticPtr* sptr) = 0;
}
```

4.1 enterBloc

La fonction enterBloc doit être appellée lorsque l'analyseur sémantique entre dans un bloc. Ainsi un nouveau bloc sera empilé sur la pile. Les recherches de symboles se feront en prenant en compte les niveaux d'imbrications des blocs.

4.2 exitBloc

Réciproquement, lorsque l'analyseur sémantique quitte un bloc, la fonction exitBloc doit être appellée. Cela permet de dépiler un bloc de la pile afin de supprimer les déclarations qui sont obsolètes car innaccessibles depuis l'exterieur du bloc.

5 Étapes de l'analyse sémantique

5.1 Liste des vérifications à réaliser

Au cours de l'analyse sémantique il faudra effectuer de nombreuses vérifications. Voici la liste des éléments à vérifier pour compléter une partie de l'analyse sémantique (en effet il y a également une partie décoration) ; dans le désordre :

- Déclaration de variable avec un type existant
- Affectation avec un typage correct
- Comparaison entre type correct (cast implicite autorisé)
- Pas de cycles dans l'héritage
- Appelle de méthode « static » au sein d'une méthode =¿ la méthode doit être « static »
- Présence de méthode « abstract » dans une classe =¿ la classe doit être « abstract »
- Définition de toutes les méthodes lors de l'implémentation d'une interface
- Appelle de méthode correcte (méthode existante + paramètres valide)
- Return présent et correctement typé dans les méthodes avec un type de retour != void
- Appelle de variable correcte (variable déclarée et visible au moment de l'appel)
- Pas de modification sur une variable « final »
- Pas de nommage identique pour les classes

5.2 Les différentes phases de l'analyse sémantiques

Pour réaliser une analyse sémantique complète plusieurs passes dans l'arbre sont nécessaires. Nous allons avoir besoins de parcourir l'arbre 3 (ou 4fois)

5.2.1 Phase 1

- Enregistrer chaque classe comme un type

5.2.2 Phase 2

- Décorer chaque classe avec le type de sa classe mère et de ses interfaces
- Décorer les attributs
- Décorer les méthodes :
  - Associer une référence aux paramètres (et type de retour ?)
  - Calcul et enregistrement de la signature complète

5.2.3 Phase 2 (suite) ou 3

- Création des interfaces publiques pour chaque classe (listes des méthodes et attributs complètes).

5.2.4 Phase 3 ou 4 (selon choix précédent)

- Analyse sémantique des corps de méthodes