CSC4251_4252 : Analyse Sémantique

Pascal Hennequin, J. Paul Gibson, Denis Conan

Novembre 2024

Sommaire

1 (Rappel) Objectifs de l'analyse sémantique

- Valider les règles du langage non gérables au niveau syntaxique
- Établir la signification du programme et sa sémantique d'exécution
- Construire les éléments de contexte
  - Informations nécessaires pour l'exécution mais non explicites dans l'arbre de syntaxe, ou explicites dans une autre partie de l'arbre
- Vérifier ou Inférer des propriétés de l'algorithme
  - Terminaison, validité de l'algorithme, non débordement, ...
  - N.B. : Théorème de Rice $\Longrightarrow$ problèmes indécidables

2 Compilation Classique versus Moderne I

- **Compilation Classique versus Moderne**
  - « *Dragon Book* », Aho, Ullman, 1977 (*green DB*), 1986 (*red* DB, 1ère éd.), 2006 (2ème éd.), 2017 (2ème éd. en français)
  - « *Modern Compiler* », Appel, 1997 (1ère éd.), 2002 (2ème éd.)
- Identique pour l'analyse lexicale et syntaxique
- Différences importantes sur l'analyse sémantique et la génération de la forme intermédiaire
- Différences sur les solutions techniques et aussi sur le problème à résoudre

2 Compilation Classique versus Moderne II

- **Compilation Mono-passe *vs.* Multi-passe ?**
  - Mono-passe = Exécution en séquence dans la même passe
    - Contraignant pour l'algorithmique du compilateur
    - Contrainte sur le langage : tout ce qui est nécessaire pour analyser une portion de programme doit exister « avant » dans le source
  - Multi-passe = Chaînage (pipeline) des phases
    - Plus souple dans la programmation et l'indépendance des phases
- **Historiquement : Mono-passe était un enjeu vital**
  - Économe sur les performances en temps, en espace, en entrée-sortie
  - Algorithmique lourde pour tout résoudre en une fois
- **Aujourd'hui : Multi-passe encouragé**
  - Contraintes plus faibles sur les performances
  - Langages plus « libéraux » (p.ex. JAVA)

3 Approches pour l'analyse sémantique

- **De la théorie :**
  - « Haute » : lambda-calcul, théorèmes du point fixe, preuve de programmes
    - Sémantique formelle (opérationnelle, dénotationnelle, axiomatique), théorie de l'interprétation abstraite
  - « Basse » : grammaires attribuées, traduction dirigée par la syntaxe
    - P.ex., graphe des dépendances pour le calcul des attributs
      = très difficile à calculer
- **Dans ce module, approche pragmatique**
  - Des fonctions sémantiques réalisées de façons modulaires et en passes successives

4 Analyse structurelle et comportementale

- **Structurelle**
  - Les attributs sémantiques sont calculés sur la structure de l'AST
- **Comportementale**
  - On ajoute à l'AST les chemins d'exécution du programme pour calculer des attributs qui vont hériter de nœuds traversés « avant » ou « après »
  - Algorithmique :
    - Flot d'exécution avec « Couture d'arbre »
    - Interprétation symbolique
    - Équations de flots de données
    - Analyse de vivacité

5 Des fonctions sémantiques I

Liaison des identificateurs

- Table des symboles
- Détections : « non défini », « déjà défini », « initialisation »
- Consistance des définitions : détection de boucle (héritage JAVA, etc.), etc.
- Surcharge (*overload*) de fonction ou de méthode
- Paradigmes orientés objet : héritage (simple *vs.* multiple), redéfinition (*override*)

Contrôle de Type

- Typage des expressions, contraintes d'affectation et d'appel
- Surcharge des opérateurs
- Transtypage (*cast*) implicite, équivalence des types
- Héritage de l'orientation objet

5 Des fonctions sémantiques II

- **Propagation de constantes**
  - Évaluation « immediate » de l'assembleur, optimisation d'expressions
  - Calcul d'intervalles : réduction des contrôles à l'exécution, optimisation de boucle, etc.
  - Élimination de code
- **Analyse de vie, Analyse de dernière définition**
  - Optimisation mémoire : registre / sauvegarde de registre / pile / tas
  - Variable *unused*, p.ex. paramètre d'appel
  - *Garbage Collection*
- **Optimisation des appels de fonctions**
  - Fonction *vs.* recopie/insertion de code
  - Détection / Optimisation de récursivité

5 Des fonctions sémantiques III

- **Vivacité, Analyse de flot de données/contrôle**
  - Code mort
  - Existence `return` dans toutes les exécutions d'une méthode
  - ...
- **Signalement de sémantiques douteuses : Compromis entre optimisation silencieuse et détection d'erreurs de programmation**
  - Boucles infinies, code mort, ...
  - Transtypages « étranges »
  - Expressions à valeurs triviales

6 Contrôle de type I

- **Typage**
  - Partie importante en volume de la définition d'un langage
  - Mais beaucoup de règles rapides à vérifier, bien que difficilement gérables au niveau syntaxique
- **Valider les contraintes de typage du langage**
  - Conformité des opérateurs
  - Affectations
  - Appel de fonctions
- **Fixer les contraintes pour l'implantation des variables :**
  - Type $\Longrightarrow$ taille mémoire, registre ou pas, ...

6 Contrôle de type II

- **Valider et réaliser les transtypages implicites ou explicites :**
  - Équivalence de types, héritage
  - Fonctions de conversion : *autoboxing*, transtypage ou *cast*
- **Calcul de la sémantique des expressions**
  - Surcharge des opérateurs
- **Gérer les mécanismes de construction de type du langage**
  - Références, tableaux, enregistrements/structures, énumérations, ensembles, ...
  - Héritage multiple

7 Contrôle de type dans MiniJAVA

7.1 Sous-typage, transtypage, redéfinition
7.2 Liaison entre identificateurs et déclarations
7.3 Visiteur pour le calcul et le contrôle des types

7.1 Sous-typage, transtypage, redéfinition

- Rappel : t est du type $T \equiv t \in T$; S est un sous-type de $T \equiv S \subseteq T$
- Transtypage (implicite) vers le haut (*upcast*)
  - Avec s et t de type S et T respectivement, on peut écrire « `t = s` » :

    Par exemple :

    ```
    S s = new S(...);
    T t = s;
    ```
  - Après « `t = s` », T est appelé le type formel de t et S le type actuel de t
- Liaison dynamique ou tardive = appeler l'opération du type actuel
  - Si S redéfinit l'opération *op* préalablement définie dans T
    Après « `t = s` », « `t.op()` » utilise *op* redéfinie dans S

. Ne pas confondre redéfinition (*overriding*) et surcharge (*overloading*) :
Surcharge (*overriding*) = dans la même classe, des méthodes avec le même nom, mais avec des arguments différents (différences dans le nombre et/ou le type des arguments fournis).

7.2 Liaison entre identificateurs et déclarations

- Liaison = rechercher la déclaration en partant de la portée courante et en remontant l'arbre des portées
  - Dans MiniJAVA, méthodes `Scope::lookupKlass`, `Scope::lookupVarType`
- Fonctionne pour les méthodes de classe (`static`)
- Mais ne fonctionne pas directement pour les méthodes d'instance
  - Pour « `x.get();` », où chercher la déclaration de la méthode « `get` » ?
    - MiniJAVA = limitation à la liaison statique : type « formel » de `x` connu lors de l'analyse sémantique
      - Autrement dit, MiniJAVA ne fait pas de liaison dynamique
    - JAVA = liaison dynamique (tardive) : type « actuel » de x connu à l'instanciation avec `new`
- Pour la liaison dynamique, il faudrait réaliser la liaison :
  - ∨ Dans une phase d'analyse sémantique comportementale (flot d'exécution et coutures d'arbres)
  - ∨ À l'exécution

7.3 Visiteur pour le calcul et le contrôle des types

- Types autorisés dans MiniJAVA : booléen, entier (, tableau), « `void` », classe
- Calculs et vérifications des types primitifs dans les opérateurs
  - Dans MiniJAVA, pas de transtypage pour les types primitifs
    Dans JAVA, *autoboxing*, etc.
- Pour les objets, cf. méthode récursive `compareType` appelée par `checkType`
- Méthodes : redéfinition, mais pas de surchage et pas de liaison dynamique [1]
  - `Method` : vérification sur le type de retour (déclaration *vs.* instruction `return`)
  - `ExprCall` : méthode dans la classe de l'appelé ou dans une des classes parentes
    puis, vérification des types des paramètres formels
    et enfin, calcul du type de retour
- Instructions : les nœuds des tests (booléens)
  - Limitation MiniJAVA, instruction `System.out.println` = que des entiers ;-)

1. Pas plus de méthode abstraite.