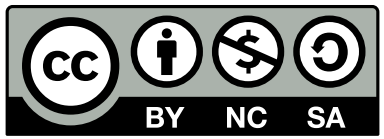
Chapitre 10: Langages à objets

INF7641 Compilation

Jean Privat

Université du Québec à Montréal

INF7641 Compilation
v251



Plan

Langages à objets

Des objets

Les stars des langages à **objets**

- Les meilleures valeurs (les seules?)
- Une **capsule**

Ont une **identité**

- Immuable
- Distinction == et `equals`
- Identité == pointeur ?

Et des propriétés (encapsulées)

- Des attributs (ou champs, ou variables d'instances)
- Des méthodes (ou opérations, ou fonctions)

Du polymorphisme

Libre arbitre ?

- Un objet décide se son propre comportement
- $\rightarrow$ On lui **envoie un message** (nom de méthode, sélecteur)
- $\rightarrow$ Il répond par sa méthode (ou son attribut)

Invocation de méthode, liaison tardive

Qualités de génie logiciel

- Réduit le couplage entre clients et objets

Des classes

- (Pas toujours)

Ensemble d'objets

- Regroupe (factorise) des objets similaires
- $\rightarrow$ Leurs instances

Information statique (dans le code source)

- Décrit (moule?) les objets

Ensemble de propriétés

- Celles décrites
- Pas besoin de traiter individuellement avec chacun des objets
- Permet de raisonner plus facilement sur le code

De la spécialisation

- Regroupe (factorise) les classes similaires

Relation d'ordre

- Spécialisation : relation d'ordre strict
  - Transitive, aréflexive et asymétrique

Guide l'héritage des propriété et la catégorisation des objets

- Les sous-classes héritent les propriétés de leurs super-classes
- Les instances d'une classes sont aussi les instances de ses super-classes

Mondes statique et dynamique

Mondes statique et dynamique

Statique

- À froid
- On regarde le code
- Il n'y a pas de valeur (ni d'objet)
- $\rightarrow$ L'univers du compilateur

Dynamique

- $\rightarrow$ L'univers de l'exécution
- À chaud
- Des objets (des instanciations)
- Qui s'envoient joyeusement des messages
- Et mutent

Typage statique et dynamique

Statique

- L'annotation (parfois implicite) des expressions, attributs, paramètres et valeurs de retour
- Permet au compilateur (ou a l'IDE) de signaler les erreurs de types

Dynamique

- La nature des valeurs
- Détermine les méthodes et attributs possédés

Erreur de type

- Demander à un objet une méthode (ou un attribut) qu'il ne possède pas
- (et parfois appeler une méthode avec un argument d'un mauvais type)

Typage sûr

Définition 1

Le typage statique prévient toute possibilité d'erreur de type à l'exécution

- Le type dynamique est **toujours** sous-type ou égal au type statique
- Est-ce vraiment suffisant ?

Définition 2

- Toute erreur de type à l'exécution est rattrapée

Dans la vrai vie

- Un peu des deux

Effacement des classes

Les classes sont statiques

- Ne servent qu'à définir les objets pour le programmeur

Les objets sont dynamiques

- Si les classes sont superflues
- Le compilateur pourrait (devrait!) s'en débarrasser !

Pourquoi c'est difficile à compiler ?

Polymorphisme

- Le comportement d'un objet dépend de sont **type dynamique**
- Or le type dynamique est dynamique, donc inconnu à la compilation

Généralisation à outrance

- Les valeurs sont des objets opaques
- Les opérations sont (presque?) toutes des méthodes

```
public void caseStmt_Assign(NStmt_Assign node) {
    setVariable(node.get_Id(), visitExp(node.get_Exp());
}
```

Toutes nos optimisation

... sont presque inutiles

Mécanismes objets

Test de sous-typage

- `instanceof`
- `cast` explicites
- `cast` implicites (covariance par exemple)

Accès aux attributs

- `a.foo = a.foo + 1;`

Envoi de message

- `c = a.bar(b);`

Monde ouvert / monde fermé

- Est-ce que l'ensemble du programme est connu à la compilation ?
- Et-ce que des sous-classes, inconnues, existeront ?

Limites

- La définition d'une sous-classe nécessite la connaissance de ses super-classes
- Pas toujours vrai, voir *raffinement de classes*

Sous-typage simple

Sous-typage simple

- Un objet
  - Tableau (structure?) d'attributs
  - Identité == adresse en mémoire
- Attributs
  - Des pointeurs vers d'autres objets
- Méthodes
  - Un tableau de pointeurs
  - $\rightarrow$ Table des fonctions virtuelles (*virtual function table*, VFT)

Très simple, très efficace

Invariants

Invariant 1

- le pointeur sur un objet reste invariant et ne dépend pas de son type statique ou dynamique
- $\rightarrow$ « casting » ascendant gratuit

Invariant 2

- L'index (la positon) d'une méthode (ou d'un attribut) reste invariant par spécialisation
- $\rightarrow$ On peut le déterminer statiquement, même en mode ouvert

Pseudo-asm

Accès aux attributs

```
load attribut ← [recv + #attrOffset]
```

Appel de méthode

```
load table ← [recv + #classOffset]
load method ← [recv + #selectorOffset]
call method
```

Test de sous-typage (et casting descendant)

- Cohen,91
- Attribuer à chaque classe un identifiant et un décalage (index)

```
load table ← [objet + #tableOffset]
load id ← [table + #downTypeOffset]
bneq id, #downTypeId, #fail
```

Détails

- Test de taille de la table
- `#downTypeOffset` = index de la sous-table du sous-type
- `#downTypeId` = ?

Double numérotation (*relative numbering*)

On attribut à chaque classe a deux numéros : x et y

- $i \leftarrow 0$
- Parcours en profondeur de la hiérarchie de classes
- $x_n \leftarrow ++i$ pour le nœud n lors de la descente
- $y_n \leftarrow i$ lors de la remontrée

Test de type

$B \prec A$ si et seulement si $x_A < x_B \leq y_A$

Problème ?

Double numérotation (*relative numbering*)

On attribut à chaque classe a deux numéros : x et y

- $i \leftarrow 0$
- Parcours en profondeur de la hiérarchie de classes
- $x_n \leftarrow ++i$ pour le nœud n lors de la descente
- $y_n \leftarrow i$ lors de la remontrée

Test de type

$B \prec A$ si et seulement si $x_A < x_B \leq y_A$

Problème ?

- Ne fonctionne pas en monde ouvert

Appel à super

- Dépend seulement, de la classe de définition de la méthode
- Liaison tardive superflue

Héritage multiple

Héritage multiple

- Problème : l'invariant 2 ne tient plus

Solution C++

- Héritage non-virtuel
- et/ou briser les invariant 1 et 2

Nouveaux invariants (plus faibles)

- Chaque attribut a un indice non-ambigu et invariant dans le contexte du type statique qui l'introduit
- Chaque méthode a un indice non-ambigu et invariant dans le contexte d'un type statique qui l'introduit ou l'hérite
- Toute entité de type statique s est lié (pointe) sur un sous-objet correspondant à la classe s et muni de sa propre table des méthodes

Bris d'invariant

Il faut se parfois se déplacer entre les sous-objets

Lors d'affectation

- Une affectation vers une *rvalue* d'un super-type
- $\rightarrow$ Pour pointer sur le sous-objet compatible

Lors d'accès aux attributs

- $\rightarrow$ Pour trouver le bon attribut dans le bon sous-objet

Lors de coercition descendante (réussie)

- $\rightarrow$ Pour pointer le sous-objet compatible

Lors d'un appel de méthode

- Pour se trouver sur le « bon » receveur
- $\rightarrow$ Le sous-objet de la classe qui a implémenté la méthode

Informations de décalage

- Doubler la taille de la table des méthodes
- Ajouter le décalage pour le receveur de chaque méthode

```
load table ← [object + #tableOffset]
load delta ← [table + #deltaOffset]
load method ← [table + #selectorOffset]
add recv ← object + delta
call method
```

Technique (alternative) des *thunks*

- Fait le décalage et branche sur la méthode
- Un *thunk* pour **chaque** méthode héritée de **chaque** classe

```
thunk_C_m:
add recv ← recv, #delta
jmp #method
```

(Coercition?) ascendante

- Pour chaque table des méthodes de t
- Un case `#deltaOffset` pour chaque super-type de t

```
load table ← [object, #tableOffset]
load delta ← [table, #deltaOffset]
add objet ← objet, delta
```

Accès aux attributs

```
load attr ← [objet + #attrOffset]
```

- Mais il faut peut-être aussi décaler le sous-objet de l'attribut

Exercice

Implémenter `a.x = b.y`

VBPTR (*virtual base class pointer*)

Optimisation de C++

- Stoque dans chauque sous-objet d'un type t
- Un pointeurs vers le sous-objets de chaque sous-type s

```
load object ← [object + #castOffset]
load attr ← [object + #attrOffset]
```

- Décalages plus faciles
- Accès aux attributs plus faciles
- Mais objets plus gros

Test de sous-typage et coercition descendante (*dyamic cast*)

- Trop compliqué
- $\rightarrow$ Recherche séquentielle
- `strcmp` sur le nom des classe rencontrées

Technique alternative

- Table de hachage

Test d'identité (==)

Pour chaque table des méthodes

- un décalage vers le sous-objet du type dynamique (le plus spécifique)

```
load table1 ← [objet1 + #offsetTable]
load ddelta1 ← [table1 + #dynamicDelta]
load table2 ← [objet2 + #offsetTable]
load ddelta2 ← [table2 + #dynamicDelta]
add dobjet1 ← objet1, ddelta1
add dobjet2 ← objet2, ddelta2
seq result ← dobjet1, dobjet2
```

Appel à super

- Garder un appel statique ?

Appel à super

- Garder un appel statique ?
- Brise la sémantique objet
- Risque d'invocation répétée
- $\rightarrow$ Bonne sémantique: linéarisation

Implémentation

$\rightarrow$ Simulation de méthode polymorphe

- Chaque site d'appel à super introduit une « nouvelle méthode »
- $\rightarrow$ Une case dans les table des méthodes
- L'implémentation (redirection), dépend de chaque sous-classe

Diable

- Brise « un peu » l'encapsulation (mais c'est débattable)

Non-virtualisation

Absence du mot clé `virtual` en C++ (et C#)

Sur les fonctions

- Les appels ne sont pas polymorphes → Le type statique (le client) décide, et non l'objet (son type dynamique)
- Si toutes les fonctions sont non-virtuelles
  → On a quasiment une `struct` C

Sur liens de spécialisation

- La spécialisation devient une **inclusion** structurelle
  → Implémentation de l'héritage simple
- Mais sémantique brisée
  → Héritage répété : duplication des attributs & identité problématique

Combinaison

- Mélange de fonctions virtuelles et non-virtuelles + Mélange d'héritage virtuel et non-virtuel = maux de tête

Dévirtualisation

Supprimer les `virtual` qui ne changent pas la sémantique

- Supprimer les `virtual` des fonctions non redéfinies
- Supprimer les `virtual` sur un arbre couvrant de la hiéarchie de spécialisation
- $\rightarrow$ Approche nécessairement globales et en monde clos

Mise en œuvre

- Ne pas faire à la main (vous allez vous blesser)
- À refaire à « chaque » compilation

Implémentation globale et monde clos

- Arbres de sélection
- Coloration des méthodes et attributs
- Voir hachage parfait

Typage hybride

Typage hybride

- Héritage simple
- Mais sous-typage multiple
- $\rightarrow$ Apparition des **interfaces**

Stratégies

- Complexifier l'implémentation en sous-typage simple
- Simplifier l'implémentation en héritage multiple

Invariants

Entre classes

- Maintient des invariants de sous-typage simple
- Et de l'implémentation facile

Entre interfaces (et classes)

- Une interface jamais un type dynamique
  - $\rightarrow$ Ça simplifie des trucs
- Une interface n'a pas d'attributs
  - $\rightarrow$ Pas de sous-objets
  - $\rightarrow$ Ni de décalages
- Reste à gérer
  - Appel de méthode
  - Test de sous-typage

La vrai vie

- Les objets c'est plus à la mode
- Ni l'implémentation des langages à objets

Monde fermé incrémental

- VM et JIT
- Et autres techniques

Si la perf n'est pas un enjeu

- Tables de hachage depuis le début
- $\rightarrow$ Comme dans les langages à typage dynamique

Analyse inter-procédurale

Graphe d'appel d'un programme

Graphe orienté enraciné

- Les sommets sont des sous-programmes (méthodes, fonctions, etc.)
- La racine est le point d'entrée du programme
- Les arcs sont des appels (potentiels?) entre ces méthodes

Utilité d'un graphe d'appel

- Suppression du code mort
- Base d'autre analyses et optimisations
  - *Inlining*, *taint analysis*, etc.

Problème

- Le construire n'est pas toujours facile
- Polymorphisme, langage fonctionnel, pointeurs sur fonction, etc.

Polymorphisme, le retour

- Un site d'appel est **monomorphe**
  s'il n'invoque qu'une seule méthode
- Un site d'appel est **polymorphe** sinon
  - **oligomorphe** un peu
  - **megamorphe** beaucoup

RE et UN

RA (*reachability analysis*)

Srivastava, 1992

- Un site d'appel appelle toutes les méthodes qui ont le même nom

UN (*unique name*)

Calder&Grunwald, 1994

- S'il n'y a qu'une seule méthode qui s'appelle `n` alors tous les sites d'appel de `n` sont monomorphes

CHA (*class hierarchy analysis*)

Dean&Chambers, 1994

Raffine RA en prenant en compte

- Les types statiques des receveurs
- Le graphe d'héritage

RTA (*rapid type analysis*)

Bacon&al,1996

Approxime deux ensembles: les classes vivantes et les méthodes vivantes

- Le point d'entrée d'un programme est vivant
- Si `new C` apparaît dans une méthode vivante,
  alors C es vivant
- Si `x.foo` apparaît dans une méthode vivante,
  alors toutes les méthodes `foo` des classe vivantes sous-type du type statique de `x` sont vivantes

Aller plus loin ?

- Approximer les types dynamique possibles de chaque receveur
- $\rightarrow$ C'est que que font naïvement CHA et RTA
- Mais comment faire ?

Aller plus loin ?

- Approximer les types dynamique possibles de chaque receveur
- $\rightarrow$ C'est que que font naïvement CHA et RTA
- Mais comment faire ?

Analyse de flux iNtErPrOcÉdUrAlE !

- On connecte les sites d'appels aux CFG des méthodes appelées
- On fait tourner jusqu'au point fixe
- (c'est un peu plus compliqué en vrai)

CFA (*control flow analysis*)

Shivers,1988

- Ensemble: pour chaque variable et chaque attribut, un ensemble de types
- Analyse en avant
- Fusion « union-ish » (voir INF889A)
- Pour `x = new C` met l'ensemble de types de `x` à { C }
- Pour `z = x.foo(y)`
  - Pour chaque méthode `foo` définie dans les types de `x`
  - Connecter `x` et `y` aux arguments de `foo`
  - Connecter le résultat de `foo` à `z`
- Pour les autres instructions,
  mettre une fonction de transfert qui a de l'allure

Problème de CFA

- Lent et lourd
- Pas beaucoup meilleur que RTA en pratique (à sourcer)
- Pourquoi ?

Problème de CFA

- Lent et lourd
- Pas beaucoup meilleur que RTA en pratique (à sourcer)
- Pourquoi ?

Pollution

```
a = new A
b = a.identity()
c = new C
d = c.identity()
```

- Mélange les types à chaque **méthode**
- Or en programmation objet, on définit des méthodes générales qu'on réutilise dans beaucoup de contextes différents

Particularisation (*customisation*)

Chambers&al, 1989

On « recopie » chaque méthode pour chaque classe qui l'hérite

- Les appels avec des receveurs de types différents sont distingués
- Mais augment la complexité algorithmique
- Et c'est pas *si* fantastique en pratique (à sourcer)

Sensibilité au contexte (*context sensitivity*)

(On parle aussi de techniques polyvariantes)

- Généralisation de l'idée :
  distinguer des **contextes** particuliers aux appels de méthodes
- Faire une « version » de chaque méthode pour chaque contexte d'appel

1-CFA

Palsberg&Schwartzbach, 1991

- Un contexte par site d'appel
- Chaque utilisation est distinguée
- (Combinable avec la particularisation pour plus de fun)

Avantage et inconvénients

k-CFA

Vitek&al,1992

Un contexte correspond aux k derniers sites d'appels de la chaîne d'appel.

- Avantages et inconvénients ?

k-CFA

Vitek&al,1992

Un contexte correspond aux k derniers sites d'appels de la chaîne d'appel.

- Avantages et inconvénients ?

k-l-CFA

On distingue les classes par les l derniers sites d'appels de la chaine d'appel

- Permet de distinguer les attributs des classes

Techniques adaptatives

- Ne pas lier les contexte aux chaînes d'appels
- Mais aux types des arguments

$\rightarrow$ C'est ce que fait la particularisation pour le receveur

CPA (*cartesian product analysis*)

Agessen, 1994

Pour `x.foo(y,z)` on associe un contexte pour chaque élément de $[[x]] \times [[y]] \times [[z]]$

- En gros on particularise aussi les types des arguments
- (qui a dit multi-méthodes?)

Avantages et inconvénients

CPA (*cartesian product analysis*)

Agessen, 1994

Pour `x.foo(y,z)` on associe un contexte pour chaque élément de $[[x]] \times [[y]] \times [[z]]$

- En gros on particularise aussi les types des arguments
- (qui a dit multi-méthodes?)

Avantages et inconvénients

- Complexité spatiale théorique en $O(n^{2a})$ où a est l'arité des méthodes
- $\rightarrow$ Ajout d'heuristiques pour borner et élaguer

SCS (*simple class set*)

Grove&al, 1997

Version « condensée » de CPA

- On crée un seul contexte pour $[[x]] \times [[y]] \times [[z]]$

Avantages et inconvénients

SCS (*simple class set*)

Grove&al, 1997

Version « condensée » de CPA

- On crée un seul contexte pour $[[x]] \times [[y]] \times [[z]]$

Avantages et inconvénients

- Complexité spatiale en $O(n^{n^n})$
- Mais en pratique plus rapide que versions bornées de CPA

La vrai vie

- Lambdas
- Éval
- Fonctions primitives, natives et étrangères (monde semi-ouvert?)
- Parallélisme
- $\rightarrow$ INF889A

La prochaine fois

- LR