

FSAB1402: Informatique 2

L'Etat et l'Abstraction de Données



Peter Van Roy
Département d'Ingénierie Informatique, UCL

pvr@info.ucl.ac.be



P. Van Roy

1

Ce qu'on va voir aujourd'hui

- Un exemple de l'utilisation de la sémantique
 - Pourquoi la règle "l'appel récursif doit être terminal" marche
- L'état explicite
 - L'importance pour la modularité
- L'abstraction de données
 - Deux manières: le type abstrait et l'objet

P. Van Roy

2

Résumé du dernier cours



P. Van Roy

3

Sémantique formelle



- Dans la dernière séance, nous avons vu la sémantique formelle de notre langage
 - Un mécanisme qu'on appelle **une machine abstraite**
 - Quasi tous les langages populaires utilisent une machine abstraite semblable (Java, C++, C#, Python, Ruby, etc., etc.)
- Dans le reste du cours, nous allons de temps en temps utiliser la sémantique pour expliquer quelque chose
 - Par exemple, tout de suite on va expliquer pourquoi la règle du dernier appel récursif marche

P. Van Roy

4

Pourquoi l'optimisation de la récursion terminale marche



- Nous allons utiliser la sémantique pour expliquer pourquoi la taille de la pile reste constante si l'appel récursif est la dernière instruction
- Nous allons prendre deux versions de la factorielle, une avec accumulateur (Fact2) et l'autre sans accumulateur (Fact)
- Nous allons les exécuter toutes les deux avec la sémantique
- Ces exemples se généralisent facilement pour toute fonction récursive

P. Van Roy

5

Factorielle avec accumulateur



- Voici une définition (partiellement) en langage noyau

```
proc {Fact2 I A F}
  if I==0 then F=A
  else I1 A1 in
    I1=I-1
    A1=I*A
    {Fact2 I1 A1 F}
  end
end
```

- Nous allons exécuter cette définition avec la sémantique
- Nous allons démontrer que la taille de la pile est la même juste avant chaque appel de Fact2

P. Van Roy

6



Début de l'exécution de Fact2

- Voici l'instruction qu'on va exécuter:

```
local N A F in  
    N=5 A=1  
    {Fact2 N A F}  
end
```

- On suppose que la définition de Fact2 existe déjà
- Pour être complet il faut ajouter la définition de Fact2

P. Van Roy

7



Début de l'exécution (complet)

- Voici la vraie instruction qu'on va exécuter

```
local Fact2 in  
    proc {Fact2 I A F}  
        if I==0 then F=A  
        else I1 A1 in  
            I1=I-1  
            A1=I*A  
            {Fact2 I1 A1 F}  
        end  
    end  
local N A F in  
    N=5 A=1  
    {Fact2 N A F}  
end
```

P. Van Roy

8



Premier appel de Fact2

- Voici l'état juste avant le premier appel:
([({Fact2 N A F}, {Fact2→p, N→n, A→a, F→f}),
{n=5, a=1, f, p=(...)}])
- Un pas plus loin (l'exécution de l'appel commence):
([({if l==0 then F=A else l1 A1 in
l1=l-1 A1=A*I {Fact2 l1 A1 F} end,
{Fact2→p,l→n,A→a,F→f}),
{n=5, a=1, f, p=(...)}])



Deuxième appel de Fact2

- Juste avant le deuxième appel:
([({Fact2 l1 A1 F},
{Fact2→p, l→n, A→a, F→f, l1→i₁, A1→a₁}),
{n=5, a=1, i₁=4, a₁=5, f, p=(...)}])
- On voit que la pile ne contient qu'un seul élément, tout comme le premier appel
- On peut facilement voir que tous les appels de Fact2 seront le seul élément sur la pile
- QED!



Factorielle sans accumulateur

- Voici une définition (partiellement) en langage noyau

```

proc {Fact N F}
  if N==0 then F=1
  else N1 F1 in
    N1=N-1
    {Fact N1 F1}
    F=N*F1
  end
end

```

- Nous allons exécuter cette définition avec la sémantique, avec le premier appel {Fact 5 F}
- Nous allons démontrer que la taille de la pile augmente d'un élément pour chaque nouvel appel récursif

P. Van Roy

11



Début de l'exécution de Fact

- Voici l'exécution juste avant le premier appel:
 $\langle\langle\{\text{Fact } N \ F\}, \{\text{Fact} \rightarrow p, N \rightarrow n, F \rightarrow f\}\rangle\rangle,$
 $\{n=5, f, p=(\dots)\}$
- La partie **else** de l'instruction **if**:
 $\langle\langle\{N1=N-1 \ \{\text{Fact } N1 \ F1\} \ F=N*F1,$
 $\{\text{Fact} \rightarrow p, N \rightarrow n, F \rightarrow f, N1 \rightarrow n_1, F1 \rightarrow f_1\}\rangle\rangle,$
 $\{n=5, f, n_1, f_1, p=(\dots)\}$
- Juste avant le second appel de Fact:
 $\langle\langle\{\text{Fact } N1 \ F1\},$
 $\{\text{Fact} \rightarrow p, N \rightarrow n, F \rightarrow f, N1 \rightarrow n_1, F1 \rightarrow f_1\}\rangle\rangle$ ← Appel récursif
 $\langle\langle F=N*F1,$
 $\{\text{Fact} \rightarrow p, N \rightarrow n, F \rightarrow f, N1 \rightarrow n_1, F1 \rightarrow f_1\}\rangle\rangle,$ ← Après l'appel
 $\{n=5, f, n_1=4, f_1, p=(\dots)\}$

P. Van Roy

12



Plus loin dans l'exécution

- Un des appels suivants de Fact:
 $((\{Fact\ N1\ F1\}, \{...\}),$
 $(F=N*F1, \{F \rightarrow f_2, N \rightarrow n_2, F1 \rightarrow f_3, \dots\}),$
 $(F=N*F1, \{F \rightarrow f_1, N \rightarrow n_1, F1 \rightarrow f_2, \dots\}),$
 $(F=N*F1, \{F \rightarrow f, N \rightarrow n, F1 \rightarrow f_1, \dots\}),$
 $\{n=5, f, n_1=4, f_1, n_2=3, f_2, \dots, p=(\dots)\})$
- A chaque appel successif, une nouvelle instruction "F=N*F1" se met sur la pile
 - Avec un autre environnement bien sûr!
- La pile contient toutes les multiplications qui restent à faire

P. Van Roy

13



Généraliser ce résultat

- Pour que le résultat tienne pour **toutes les fonctions récursives**, il faut définir un **schéma** pour l'exécution d'une fonction récursive
 - Schéma = une représentation de l'ensemble de toutes les exécutions possibles des fonctions récursives
 - On redéfinit la sémantique pour marcher sur le schéma
- Est-ce que l'exécution de l'exemple concret tiendra toujours pour toutes les exécutions du schéma?
 - Oui!
 - La vérification de ce fait est hors de portée pour ce cours, mais si vous avez un esprit mathématique vous pouvez le démontrer rigoureusement

P. Van Roy

14

Conclusion



- Quand l'appel récursif est la dernière instruction, la taille de la pile reste constante
- Quand l'appel récursif n'est pas la dernière instruction, la taille de la pile grandit d'un élément pour chaque appel récursif
 - La pile contient toutes les instructions qui restent à faire
- La sémantique nous montre exactement ce qui se passe!

L'état



La notion de temps



- Dans le modèle déclaratif, **il n'y a pas de temps**
 - Les fonctions sont des fonctions mathématiques, qui ne changent jamais
- Dans le monde réel, **il y a le temps et le changement**
 - Les organismes changent leur comportement avec le temps, ils grandissent et ils apprennent
 - Comment est-ce qu'on peut modéliser ce changement dans un programme?
- On peut ajouter une notion de **temps abstrait** dans les programmes
 - Temps abstrait = une séquence de valeurs
 - Un état = un temps abstrait = une séquence de valeurs

P. Van Roy

17

L'état est une séquence dans le temps



- Un **état** est une séquence de valeurs calculées progressivement, qui contiennent les résultats intermédiaires d'un calcul
- Le modèle déclaratif peut aussi utiliser l'état selon cette définition!
- Regardez bien la définition ci-jointe de Sum

```
fun {Sum Xs A}
  case Xs
  of nil then A
  [] X|Xr then
    {Sum Xr A+X}
  end
end
```

```
{Browse {Sum [1 2 3 4] 0}}
```

P. Van Roy

18



L'état implicite

- Les deux arguments Xs et A représentent un **état implicite**

Xs	A
[1 2 3 4]	0
[2 3 4]	1
[3 4]	3
[4]	6
nil	10

- Implicite** parce qu'on n'a pas changé le langage de programmation
- C'est purement une interprétation de la part du programmeur


```
fun {Sum Xs A}
  case Xs
  of nil then A
  [] X|Xr then
    {Sum Xr A+X}
  end
end
```

```
{Browse {Sum [1 2 3 4] 0}}
```



L'état explicite

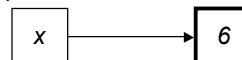
- Un état peut aussi être **explicite**, c'est-à-dire, on fait une extension au langage
- Cette extension nous permettra d'exprimer directement une séquence de valeurs dans le temps
- Notre extension s'appellera une **cellule**
- Une cellule a un contenu qui peut être changé
- La séquence de contenus dans le temps est un état

 Une variable libre

Création d'une cellule avec contenu initiale 5



Changement du contenu qui devient 6

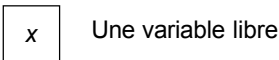


Une cellule



- Une cellule est un conteneur avec **une identité** et **un contenu**
 - L'identité est constante (le "nom" ou l'"adresse" de la cellule)
 - Le contenu est une variable (qui peut être liée)
- Le contenu de la cellule peut être changé

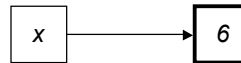
```
X={NewCell 5}  
{Browse @X}  
X:=6  
{Browse @X}
```

 Une variable libre

Création d'une cellule avec contenu initiale 5



Changement du contenu qui est maintenant 6



P. Van Roy

21

Opérations sur une cellule



- On ajoute le concept de la cellule au langage noyau
 - Il y a trois opérations de base
- **X={NewCell I}**
 - Créé une nouvelle cellule avec contenu initial I
 - Lie X à l'identité de la cellule
- **X:=J**
 - Suppose X est lié à l'identité d'une cellule
 - Change le contenu de la cellule pour devenir J
- **Y=@X**
 - Suppose X est lié à l'identité d'une cellule
 - Affecte Y au contenu de la cellule

P. Van Roy

22

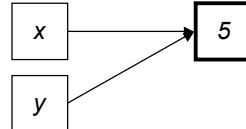


Quelques exemples (1)

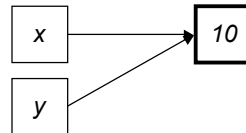
- `X={NewCell 0}`



- `X:=5`
- `Y=X`



- `Y:=10`
- `@X==10 % true`
- `X==Y % true`



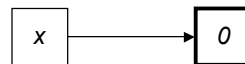
P. Van Roy

23

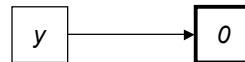


Quelques exemples (2)

- `X={NewCell 0}`
- `Y={NewCell 0}`



- `X==Y % false`
- Parce que X et Y font référence à des cellules différentes, avec des identités différentes



- `@X==@Y % true`

P. Van Roy

24

Egalité de structure et égalité d'identité



- Deux **listes** sont égales si leurs valeurs sont égales (égalité de structure)
 - Même si les structures ont été créées séparément
 - `A=[1 2] B=[1 2]`
`{Browse A==B}` % **true**
- Deux **cellules** sont égales s'il s'agit de la même cellule (égalité d'identité)
 - Deux cellules créées séparément sont toujours différentes
 - `C={NewCell 0} D={NewCell 0}`
`{Browse C==D}` % **false**
`{Browse @C==@D}` % **true**

P. Van Roy

25

Sémantique des cellules (1)



- Extension de la mémoire de la machine abstraite
- La mémoire a maintenant **deux parties**:
 - Mémoire à affectation unique (variables)
 - Mémoire à affectation multiple (cellules)
- Une cellule est **une paire**, un nom et un contenu.
 - Le contenu est une variable!
 - Le nom est une constante (une variable liée)
- Affectation de la cellule
 - Changez la paire: faire en sorte que le nom référence un autre contenu
 - Ni le nom ni le contenu changent!

P. Van Roy

26

Sémantique des cellules (2)



- La mémoire $\sigma = \sigma_1 \cup \sigma_2$ a maintenant deux parties
 - Mémoire à affectation unique
 $\sigma_1 = \{t, u, v, w, x=\zeta, y=\xi, z=10, w=5\}$
 - Mémoire à affectation multiple
 $\sigma_2 = \{x:t, y:w\}$
- Dans σ_2 il y a deux cellules, x et y
 - Le nom de x est la constante ζ
 - L'opération $X:=Z$ transforme $x:t$ en $x:z$
 - L'opération $@Y$ donne le résultat w
 - (dans l'environnement $\{X \rightarrow x, Y \rightarrow y, Z \rightarrow z, W \rightarrow w\}$)

L'état et la modularité



L'état est bénéfique pour la modularité



- On dit qu'un système (ou programme) est **modulaire** si des mises à jour dans une partie du système n'obligent pas de changer le reste
 - Partie = fonction, procédure, composant, ...
- Nous allons vous montrer un exemple comment l'utilisation de l'état explicite nous permet de construire un système modulaire
 - Dans le modèle déclaratif ce n'est pas possible

P. Van Roy

29

Scénario de développement (1)



- Il y a trois personnes, P, U1 et U2
- P a développé le module M qui offre deux fonctions F et G
- U1 et U2 sont des développeurs qui ont besoin du module M

```
fun {MF}
  fun {F ...}
    <Définition de F>
  end
  fun {G ...}
    <Définition de G>
  end
in 'export'(f:F g:G)
end
M = {MF}
```

P. Van Roy

30

Scénario de développement (2)



- Développeur U2 a une application très coûteuse en temps de calcul
- Il veut étendre le module M pour compter le nombre de fois que la fonction F est appelée par son application
- Il va voir P et lui demande de faire cela sans changer l'interface de M

```
fun {MF}
  fun {F ...}
    <Définition de F>
  end
  fun {G ...}
    <Définition de G>
  end
in 'export'(f:F g:G)
end
```

P. Van Roy

31

Dilemme!



- Ceci est **impossible** dans le modèle déclaratif parce que F ne se souvient pas de ses appels précédents!
- La seule solution est de changer l'interface de F en ajoutant deux arguments Fin et Fout:
fun {F ... Fin Fout} Fout=Fin+1 ... end
- Le reste du programme doit assurer que la sortie Fout d'un appel de F soit l'entrée Fin de l'appel suivant de F
- Mais l'interface de M a changé
 - **Tous les utilisateurs de M**, même U1, doivent changer leur programme

P. Van Roy

32

Solution avec l'état explicite



- Créez une cellule quand MF est appelé
- A cause de la portée lexicale, la cellule X est cachée du programme: elle n'est visible que dans le module M
- M.f n'a pas changé
- Une nouvelle fonction M.c est disponible

```
fun {MF}
  X = {NewCell 0}
  fun {F ...}
    X:=@X+1
    <Définition de F>
  end
  fun {G ...} <Définition de G>
  end
  fun {Count} @X end
in 'export'(f:F g:G c:Count)
end
M = {MF}
```

P. Van Roy

33

Conclusion: comparaison des modèles



- **Modèle déclaratif:**
 - + Un composant ne change jamais son comportement
 - - La mise à jour d'un composant implique souvent un changement de son interface et donc des mises à jour de beaucoup d'autres composants
- **Modèle avec état:**
 - + Un composant peut souvent être mis à jour sans changer son interface et donc sans changer le reste du programme
 - - Un composant peut changer son comportement à cause des appels précédents
- On peut parfois combiner les deux avantages
 - Utiliser l'état pour aider la mise à jour, mais quand même faire attention à ne jamais changer le comportement d'un composant

P. Van Roy

34

Un autre exemple: la mémorisation



- La **mémorisation** d'une fonction:
 - La fonction garde un tableau interne qui contient les arguments et les résultats des appels précédents
 - A chaque nouvel appel, si l'argument est dans le tableau, on peut donner le résultat sans faire le calcul de la fonction
- La mémorisation implique un état explicite dans la fonction
 - Mais la fonction garde un comportement déclaratif

Motivation pour l'abstraction



L'importance de l'encapsulation



- Imaginez que votre télévision n'aurait pas de boîtier
 - Tous les circuits de l'intérieur seraient exposés à l'extérieur
- C'est **dangereux pour vous**: si vous touchez aux circuits, vous pouvez vous exposer à des tensions mortelles
- C'est **problématique pour la télévision**: si vous versez une tasse de café dans l'intérieur, vous pouvez provoquer un court-circuit
 - Vous pouvez être tenté de chipoter avec l'intérieur, pour soi-disant "améliorer" les performances de la télévision
- Il y a donc un intérêt à faire une encapsulation
 - Un boîtier qui empêcherait une interaction sauvage, et qui n'autoriserait que les interactions voulues (marche/arrêt, volume)

P. Van Roy

37

L'encapsulation dans l'informatique



- Supposez que votre programme utilise une pile avec l'implémentation suivante:

```
fun {NewStack} nil end
fun {Push S X} X|S end
fun {Pop S X} X=S.1 S.2 end
fun {IsEmpty S} S==nil end
```
- Cette implémentation n'est pas encapsulée!
 - Il y a les mêmes problèmes qu'avec la télévision
 - La pile est implémentée avec une liste et la liste n'est pas protégée
 - Le programmeur peut créer des piles autrement qu'avec les opérations voulues
 - On ne peut pas garantir que la pile marchera toujours!

P. Van Roy

38

Une pile encapsulée



- On utilise un “boîtier” qui isole l’intérieur de la pile (l’implémentation) de l’extérieur
 - On verra comment faire cela dans un programme!
- Le programmeur ne peut pas regarder à l’intérieur
- Le programmeur peut manipuler des piles seulement avec les opérations autorisées
 - On peut donc **garantir** que la pile marchera toujours
 - L’ensemble d’opérations autorisées = **l’interface**
- Le programmeur a la vie plus simple!
 - Toute la complexité de l’implémentation de la pile est cachée

P. Van Roy

39

Avantages de l’encapsulation



- La **garantie** que l’abstraction marchera toujours
 - L’interface est bien définie (les opérations autorisées)
- La **réduction de complexité**
 - L’utilisateur de l’abstraction ne doit pas comprendre comment l’abstraction est réalisée
 - Le programme peut être **partitionné** en beaucoup d’abstractions réalisées de façon indépendante, ce qui simplifie de beaucoup la complexité pour le programmeur
- Le développement de **grands programmes** devient possible
 - Chaque abstraction a un **responsable**: la personne qui l’implémente et qui garantit son comportement
 - On peut donc faire des grands programmes en **équipe**
 - Il suffit que chaque responsable **connait bien les interfaces** des abstractions qu’il utilise

P. Van Roy

40

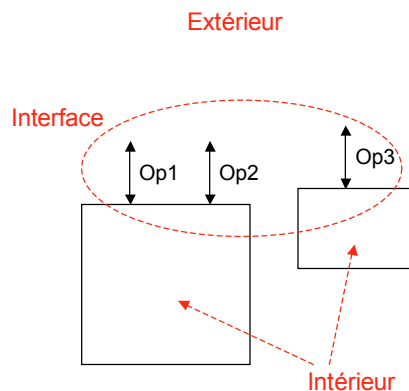
L'abstraction de données



L'abstraction de données



- Une abstraction de données a un **intérieur**, un **extérieur** et une **interface** entre les deux
- L'intérieur est caché de l'extérieur
 - Toute opération sur l'intérieur doit passer par l'interface
- Cette encapsulation peut avoir un soutien du langage
 - Sans soutien est parfois bon pour de petits programmes
 - Nous allons voir comment le langage peut soutenir l'encapsulation, c'est-à-dire faire respecter l'encapsulation



Différentes formes d'abstractions de données



- Il y a plusieurs manières d'organiser une abstraction de données
- Les deux manières principales sont l'objet et le type abstrait
 - Un **type abstrait** a des valeurs et des opérations
 - Un **objet** contient en lui-même la valeur et le jeu d'opérations
- Regardons cela de plus près!

P. Van Roy

43

Une abstraction sans état: le type abstrait



- L'abstraction consiste en un ensemble de valeurs et des opérations sur ces valeurs
- Exemple: les entiers
 - Valeurs: 1, 2, 3, ...
 - Opérations: +, -, *, div, ...
- Il n'y a pas d'état
 - Les valeurs sont des constantes
 - Les opérations n'ont pas de mémoire interne

P. Van Roy

44

Autre exemple d'un type abstrait: une pile



- Il y a des valeurs et des opérations
 - Valeurs: toutes les piles possibles
 - Opérations: NewStack, Push, Pop, IsEmpty
- Les opérations prennent des piles comme arguments et résultats
 - $S = \{\text{NewStack}\}$
 - $S2 = \{\text{Push } S \ X\}$
 - $S2 = \{\text{Pop } S \ X\}$
 - $\{\text{IsEmpty } S\}$
- Attention: ici les piles sont des valeurs, donc des constantes!

P. Van Roy

45

Implémentation d'une pile en type abstrait



- Opérations:
 - `fun {NewStack} nil end`
 - `fun {Push S X} X|S end`
 - `fun {Pop S X} X=S.1 S.2 end`
 - `fun {IsEmpty S} S==nil end`
- La pile est représentée par une liste!
- Mais la liste n'est pas protégée
- Comment est-ce qu'on peut protéger l'intérieur du type abstrait de l'extérieur?

P. Van Roy

46

Utilisation de la pile (en type abstrait)



```
S1={NewStack}
S2={Push S1 a}
S3={Push S2 b}
local X in S4={Pop S3 X} {Browse X} end
```

P. Van Roy

47

Implémentation protégée d'une pile en type abstrait



- Pour protéger l'intérieur, nous utilisons un **'wrapper'** (un emballage sécurisé)
- {NewWrapper ?Wrap ?Unwrap} crée un nouvel emballeur:
 - W={Wrap X} % W contient X mais W est protégé
 - X={Unwrap W} % Retrouver X à partir de W
- Nouvelle implémentation:

```
local Wrap Unwrap in
  {NewWrapper Wrap Unwrap}
fun {NewStack} {Wrap nil} end
fun {Push W X} {Wrap X|{Unwrap W}} end
fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
fun {IsEmpty W} {Unwrap W}==nil end
end
```
- On peut implémenter NewWrapper
 - On verra cela un autre jour

P. Van Roy

48

Implémentation des types abstraits



- Il existe des langages qui permettent au développeur de créer de nouveaux types abstraits
 - Le langage CLU, développé par Barbara Liskov et son équipe dans les années 1970, est l'exemple majeur
- Ces langages soutiennent une notion de protection similaire à Wrap/Unwrap
 - CLU a un soutien syntaxique qui facilite de beaucoup la définition de types abstraits

Une abstraction avec état: l'objet



- L'abstraction consiste en un ensemble d'objets
 - Pas de distinction entre valeurs et opérations
 - Les objets jouent le rôle des deux
- Exemple: une pile
 - $S = \{\text{NewStack}\}$
 - $\{S \text{ push}(X)\}$
 - $\{S \text{ pop}(X)\}$
 - $\{S \text{ isEmpty}(B)\}$
- L'état de la pile est dans l'objet S
 - **Exercice:** comparez la pile en type abstrait et la pile en objet
- S soutient des opérations, on dit que l'on "envoie un message" à l'objet

Implémentation d'une pile en objet



- Les mêmes opérations, mais organisées en objet:

```
fun {NewStack}
  C={NewCell nil}
  proc {Push X} C:=X|@C end
  proc {Pop X} S=@C in C:=S.2 X=S.1 end
  proc {IsEmpty B} B=(@C==nil) end
in
  proc {$ M}
    case M of push(X) then {Push X}
    [] pop(X) then {Pop X}
    [] isEmpty(B) then {IsEmpty B} end
  end
end
```

- L'objet est protégé

P. Van Roy

51

Utilisation de la pile (en objet)



```
S={NewStack}
{S push(a)}
{S push(b)}
local X in {S pop(X)} {Browse X} end
```

P. Van Roy

52

Comparaison entre objets et types abstraits



- Quels sont les avantages et désavantages respectifs des objets et des types abstraits?
- L'avantage majeur des **types abstraits** est qu'on peut faire des opérations qui **regardent à l'intérieur de plusieurs valeurs en même temps**
 - **fun** {Add Int1 Int2} ... **end**
 - On ne peut pas faire cela avec uniquement des objets
- L'avantage majeur des **objets** est **le polymorphisme**
 - La raison principale du succès des objets (à mon avis)
 - C'est aussi possible avec les types abstrait mais moins facile
 - Nous allons voir le polymorphisme la semaine prochaine
- Il y a aussi **l'héritage**
 - La définition incrémentale des objets qui se ressemblent
 - Nous allons voir l'héritage la semaine prochaine

P. Van Roy

53

Utilisation des objets et types abstraits en Java



- Contrairement à l'opinion reçue, Java fait un **savant mélange d'objets et de types abstraits**
- Les entiers en Java sont des types abstraits
 - C'est indispensable, parce que les instructions machine prennent plusieurs entiers comme arguments!
- Si O1 et O2 sont deux objets de la même classe, alors O1 peut regarder les attributs privés de O2!
 - On peut faire des méthodes qui prennent plusieurs objets comme arguments; c'est une propriété "type abstrait" qui est greffée sur les objets
- Smalltalk, un langage "purement" orienté objet, ne permet pas de faire ces choses

P. Van Roy

54

D'autres formes d'abstractions de données



- Il y a deux autres formes possibles d'abstractions de données protégées
- Un “**objet déclaratif**” (objet sans état) est possible
 - Cette forme est plutôt une curiosité!
- Un “**type abstrait avec état**” est possible
 - Cette forme est très utilisée dans le langage C
 - Regardons de plus près cette forme

P. Van Roy

55

Un type abstrait avec état



- Voici la pile en type abstrait avec état:
 - $S = \{\text{NewStack}\}$
 - $\{\text{Push } S \ X\}$
 - $X = \{\text{Pop } S\}$
 - $B = \{\text{IsEmpty } S\}$
- La pile est passée comme argument aux opérations
 - Comme un type abstrait
- La pile elle-même est changée; elle a un état
 - Comme un objet
- Cette forme est intéressante si on veut étendre le jeu d'opérations de la pile
 - On peut définir une nouvelle opération indépendamment des autres; ce n'est pas possible avec un objet classique

P. Van Roy

56

Utilisation de la pile (en type abstrait avec état)



```
S={NewStack}  
{Push S a}  
{Push S b}  
{Browse {Pop S}}
```

Résumé



Résumé



- La sémantique
 - Pourquoi la récursion terminale fait une pile à taille constante
- L'état
 - L'état explicite (la cellule)
 - L'avantage pour la modularité des programmes
 - La sémantique des cellules
- L'abstraction de données
 - Motivation: donner des garanties, la réduction de complexité, faire de grands programmes en équipe
 - Les deux formes principales: le type abstrait et l'objet
 - L'utilité de chaque forme et la mise en oeuvre en Java
 - Le type abstrait avec état: utilisé en C