

# Chapitre 7

## Test Basé sur la Spécification des Types Abstraites de Données

# Rappel sur les Types en Lotos

- ◆ Description des propriétés requises, indépendante de la future implémentation
- ◆ Signature + Axiomes
- ◆ Permettent des développements (et des évolutions) indépendants
  - du composant logiciel qui implémente le type et
  - des composants qui l'utilisent

# Exemple basique

**type Boolean is**

**sort Bool**

**opns**

true, false :  $\rightarrow$  Bool

not : Bool  $\rightarrow$  Bool

**eqns**

**ofsort Bool**

not(true) = false

not(false) = true

**endtype**

# Exemple plus sophistiqué (1)

```
type FileNat is NaturalNumber
```

```
sorts File
```

```
opns
```

```
vide :      → File
```

```
ajouter :   File, Nat → File
```

```
retirer :   File → File
```

```
premier :   File → Nat
```

# Exemple plus sophistiqué (2)

**eqns**

**forall** x, y : Nat, z : File

**ofsort** Nat

premier (ajouter(vide, x)) = x

premier (ajouter(ajouter(z, x), y) =

premier(ajouter(z, x))

**ofsort** File

retirer (ajouter(vide, x)) = vide

retirer (ajouter(ajouter(z, x), y) =

ajouter(retirer(ajouter(z, x)), y)

**endtype** (\* FileNat \*)

# Particularités du test basé sur les spécifications algébriques

- ◆ On ne peut pas vérifier un couple <données, résultat>
- ◆ On veut vérifier que les opérations implémentées par le programme satisfont les axiomes
- ◆ Exemples :

$$x + y = y + x$$

$$\text{premier}(\text{ajouter}(\text{ajouter}(z, x), y)) = \text{premier}(\text{ajouter}(z, x))$$

# Qu'est-ce qu'un test?

- ◆  $P$  fournit des procédures ou des fonctions pour exécuter les opérations  $op$  de la signature
  - (exemple : classe Java, paquetage Ada, structure ML ...)
- ◆  $op$  est implémentée par  $op_P$ 
  - *Etant donné une expression sans variable,  $t$ , on note  $t_P$  le résultat de son calcul par  $P$*
- ◆ Soit  $\varepsilon$  une équation de la signature
  - **test** de  $\varepsilon$  : toute instantiation close  $t = t'$  de  $\varepsilon$
  - **expérience de test** de  $P$  contre  $t = t'$  : évaluations de  $t_P$  et  $t'_P$  et comparaison des valeurs résultats
  - **NB** : oracle  $\Leftrightarrow$  test de l'égalité
- ◆ Ces définitions se généralisent facilement aux axiomes conditionnels positifs

# Jeu de test exhaustif

- ◆ Soit une spécification  $SP = (\Sigma, Ax)$ , le *jeu de test exhaustif* de  $SP$ , noté  $Exhaust_{SP}$  est l'ensemble de toutes les instances closes bien typées de tous les axiomes de  $SP$ :
  - **NB1** : définition dérivée de la notion classique de satisfaction d'un axiome
  - **NB2** : On n'a pas forcément d'oracle « fini » pour toutes les formes d'axiomes  $(\forall, \exists)$  ;
  - NB3** :  $Exhaust_{SP}$  est exhaustif par rapport à la spécification, pas toujours par rapport au programme...

# Hypothèse de testabilité

- ◆ Inévitable : quand on teste un système, on est obligé de faire des hypothèses sur son comportement et son environnement
- ◆ Ici :  $P$  est  $\Sigma$ -testable si :
  - Les opérations sont déterministes
  - Toutes les valeurs sont spécifiées (pas de junks)
  - Notation :  $H_{\min}$

# Test et correction

- ◆ Sous l'hypothèse de testabilité, le succès du test exhaustif garantit la correction de  $P$
- ◆ Mais comment sélectionner des sous-ensembles finis de  $Exhaust_{SP}$  ?
- ◆ La *Sélection du Jeu de Test* est basée sur le texte de la spécification (test boîte noire)
- ◆ une des solutions: “test par partition”
  - couverture des ensemble de valeurs par un nombre fini de sous-ensembles (sous-domaines)
  - choix d'au moins une donnée de test dans chaque sous-ensemble

# Sélection et Hypothèses

- ◆ On fait des hypothèses PLUS FORTES sur  $P$
- ◆ Exemple : *Hypothèse d'Uniformité*
  - $\Phi(X)$  formule,  $P$  programme,  $D$  sous-domaine
  - $(\forall t_0 \in D)(P \models \Phi(t_0) \Rightarrow (\forall t \in D)(P \models \Phi(t)))$
- ◆ Détermination des sous-domaines ? *guidée par la spécification, à suivre...*
- ◆ Autre exemple : *Hypothèse de Régularité*
  - $((\forall t \in T_\Sigma)(|t| \leq k \Rightarrow P \models \Phi(t))) \Rightarrow (\forall t \in T_\Sigma)(P \models \Phi(t))$
  - Détermination de  $|t|$ ? *cf. spécification*

# Une Méthode

- ◆ Point de départ : couverture des axiomes (un test par axiome)
- ◆  $\Rightarrow$  hypothèses d'uniformité fortes sur les sortes des variables ou sur le domaine de validité des prémisses
- ◆ Exemple : 4 cas de tests pour **FileNat**
  - $\text{premier}(\text{ajouter}(\text{vide}, x)) = x$
  - $\text{premier}(\text{ajouter}(\text{ajouter}(z, x), y)) = \text{premier}(\text{ajouter}(z, x))$
  - $\text{retirer}(\text{ajouter}(\text{vide}, x)) = \text{vide}$
  - $\text{retirer}(\text{ajouter}(\text{ajouter}(z, x), y)) = \text{ajouter}(\text{retirer}(\text{ajouter}(z, x)), y)$
- ◆ 2 sous-domaines d'uniformité pour les files : files à un élément, files à plus de un élément
- ◆ Uniformité sur les entiers

# Affaiblissement des hypothèses

- ◆ Affaiblissements successifs par utilisation des axiomes de la spécification
- ◆ Une manière naturelle pour découvrir des sous-domaines est de faire de *l'analyse de cas sur la spécification*
- ◆ Exemple : on ajoute à FileNat :

est-dans : File, Nat  $\rightarrow$  Bool

**ofsort** Bool

est-dans (vide, x) = false

est-dans(ajouter(vide,x), y) = eq(x,y)

est-dans (ajouter (z, x), y) = or (eq(x,y), est-dans (z, y))

- ◆ Avec la méthode précédente, trois tests, insuffisants
  - Les deux résultats possibles de *est-dans* ne sont pas couverts

# Affaiblissement des hypothèses par composition d'axiomes

- ◆ On sait que :
  - or (true, true) = true, or (true, false) = true
  - or (false, true) = true, or (false, false) = false
- ◆ On compose ces axiomes avec :
  - est-dans (ajouter (z, x), y) = or (eq(x, y), est-dans (z, y))
- ◆ On obtient quatre cas de test :
  - eq(x, y) = true & est-dans (z, y) = true => est-dans (ajouter (z, x), y) = true
  - eq (x, y) = true & est-dans (z, y) = false => est-dans (ajouter (z, x), y) = true
  - eq (x, y) = false & est-dans (z, y) = true => est-dans (ajouter (z, x), y) = true
  - eq (x, y) = false & est-dans (z, y) = false => est-dans (ajouter (z, x), y) = false

# Dépliage

- ◆ Dépliage: une technique classique pour transformer (et comprendre) les définitions récursives
- ◆ Remplacement de  $f(op(x))$  par la définition de  $f$ , avec remplacement adéquat des variables
  - $fact(n) =_{def} \mathbf{if\ n=0\ then\ 1\ else\ n*fact(n-1)}$  devient :
  - $fact(n) =_{def} \mathbf{if\ n=0\ then\ 1\ else\ if\ (n-1)=0\ then\ n*1\ else\ n*(n-1)*fact(n-2)}$ 
    - i.e.  $fact(n) =_{def} \mathbf{if\ n=0\ then\ 1\ else\ if\ n=1\ then\ 1\ else\ n*(n-1)*fact(n-2)}$
  - etc
  - NB : on remplace la définition de la fonction *fact* par son graphe, i.e. son test exhaustif...

# Exemple : tri par insertion

```
type ListeNat is NaturalNumber
```

```
sorts Liste
```

```
opns
```

```
  ∅ :                               → Liste
```

```
  cons :                             Nat, Liste → Liste
```

```
  insérer :                           Nat, Liste → Liste
```

```
eqns
```

```
  forall x, y : Nat, l : Liste
```

```
  ofsort Liste
```

```
    insérer (x,∅) = cons(x,∅)
```

```
    le(x, y) = true => insérer (x, cons(y, l)) = cons (x, cons(y, l))
```

```
    le(x, y) = false => insérer(x, cons(y, l)) = cons(y, insérer(x,l))
```

```
endtype (* ListeNat *)
```

# Couverture des axiomes

- ◆ Trois cas de test

insérer  $(x, \emptyset) = \text{cons}(x, \emptyset)$

$\text{le}(x, y) = \text{true} \Rightarrow \text{insérer}(x, \text{cons}(y, l)) = \text{cons}(x, \text{cons}(y, l))$

$\text{le}(x, y) = \text{false} \Rightarrow \text{insérer}(x, \text{cons}(y, l)) = \text{cons}(y, \text{insérer}(x, l))$

- ◆ *Uniformité sur les entiers*

- ◆ *Uniformité sur*

- *les couples Entier , Liste non vide où l'entier est plus petit ou égal au premier de la liste*
- *les couples Entier , Liste non vide où l'entier est inférieur au premier de la liste*

# On déplie *le*

- ◆ Définition de *le*

- $eq(x, y) = true \Rightarrow le(x, y) = true$
- $lt(x, y) = true \Rightarrow le(x, y) = true$
- $eq(x, y) = false \ \& \ lt(x, y) = false \Rightarrow le(x, y) = false$

- ◆ On obtient quatre cas de test

$ins\acute{e}rer(x, \emptyset) = cons(x, \emptyset)$

$eq(x, y) = true \Rightarrow ins\acute{e}rer(x, cons(y, l)) = cons(x, cons(y, l))$

$lt(x, y) = true \Rightarrow ins\acute{e}rer(x, cons(y, l)) = cons(x, cons(y, l))$

$eq(x, y) = false \ \& \ lt(x, y) = false \Rightarrow ins\acute{e}rer(x, cons(y, l)) = cons(y, ins\acute{e}rer(x, l))$

# On déplie *insérer* dans le dernier cas

- ◆ Dernier cas (récurusif 😊):
  - $eq(x, y) = false \ \& \ lt(x, y) = false \Rightarrow$   
 $insérer(x, cons(y, l)) = cons(y, \underline{insérer}(x, l))$
- ◆ *Insérer* est définie par trois axiomes  $\Rightarrow$  trois sous-cas
  - premier sous-cas :  $insérer(x, \emptyset) = cons(x, \emptyset)$  donc on remplace  $l$  par  $\emptyset$
  - $eq(x, y) = false \ \& \ lt(x, y) = false \Rightarrow$   
 $insérer(x, cons(y, \emptyset)) = cons(y, cons(x, \emptyset))$

# Dépliage de *insérer*, suite

- ◆ Rappel du cas à déplier
  - $eq(x, y) = false \ \& \ lt(x, y) = false \Rightarrow$   
 $insérer(x, cons(y, l)) = cons(y, \underline{insérer(x, l)})$
- ◆ sous-cas 2 :  $le(x', y') = true \Rightarrow \underline{insérer(x', cons(y', l'))} = cons(x', cons(y', l'))$
- ◆ Donc on remplace  $l$  par  $cons(y', l')$ ,  $x'$  par  $x$ , et on compose les pré-conditions des deux cas
  - $eq(x, y) = false \ \& \ lt(x, y) = false \ \& \ le(x, y') = true \Rightarrow$   
 $insérer(x, cons(y, cons(y', l')))) = cons(y, cons(x, cons(y', l')))$
  - NB : pour ceux qui connaissent, c'est de l'unification, puis de la réécriture
  - C'est le sous-cas  $y < x$  et  $x \leq y'$

# Dépliage de *insérer*, fin

## ◆ Rappel du cas à déplier

- $eq(x, y) = false \ \& \ lt(x, y) = false \Rightarrow$

$$insérer(x, cons(y, l)) = cons(y, \underline{insérer(x, l)})$$

## ◆ sous-cas 3 : $le(x', y') = false \Rightarrow \underline{insérer(x', cons(y', l'))} = cons(y', insérer(x', l'))$

## ◆ Donc on remplace $l$ par $cons(y', l')$ , $x'$ par $x$ , et on compose les conditions

- $eq(x, y) = false \ \& \ lt(x, y) = false \ \& \ le(x, y') = false \Rightarrow$

$$insérer(x, cons(y, cons(y', l'))) = cons(y, cons(y', insérer(x, l')))$$

- C'est le cas  $y < x$  et  $y' < x$

# On peut continuer...

## ◆ Dans le sous-cas 2

- $eq(x, y) = false \ \& \ lt(x, y) = false \ \& \ le(x, y') = true \Rightarrow$   
 $insérer(x, cons(y, cons(y', l'))) = cons(y, cons(x, cons(y', l')))$
- On peut déplier *le*

## ◆ Dans le sous-cas 3

- $eq(x, y) = false \ \& \ lt(x, y) = false \ \& \ le(x, y') = false \Rightarrow$   
 $insérer(x, cons(y, cons(y', l'))) = cons(y, cons(y', insérer(x, l')))$
- On peut déplier *le* et *insérer*
- Quand s'arrête-t-on?

# Quand et comment s'arrêter

- ◆ En fonction du contexte (risque, coût, délais), on décide pour chaque spécification :
  - Quels prédicats décomposer
  - Quelles opération déplier et combien de fois (rarement plus d'une fois)
- ◆ Une bonne stratégie standard : composer tous les sous-cas deux à deux
  - NB : on peut avoir des compositions de sous-cas infaisables

# Le problème de l'oracle

- ◆ décider de l'égalité de  $t_P$  et  $t'_P$ , ou décider si un prédicat est valide ou non
- ◆ Le cas simple :
  - la sorte  $s$  de  $t$  et  $t'$  correspond à un type du langage de programmation (sorte observable)
  - ou on teste un prédicat
- ◆ “Hypothèse d'oracle” faible : l'égalité sur les types du langage, et les booléens sont implémentés correctement

# Les autres cas

- ◆ Comment tester que  
 $\text{retirer}(\text{ajouter}(\text{ajouter}(z, x), y) =$   
 $\text{ajouter}(\text{retirer}(\text{ajouter}(z, x)), y) ?$
- ◆ Ou même que :  
 $\text{insérer}(x, \text{cons}(y, \emptyset)) = \text{cons}(y, \text{cons}(x, \emptyset)) ?$
- ◆ Solution : *les contextes observables*
  - On teste que toutes les observations qu'on peut faire sur les deux résultats sont égales
  - **Observation** : composition d'opérations qui donne un résultat observable
  - On peut être amené à ajouter des observateurs, voire une égalité... (dangereux)

# Tous les contextes sont nécessaires pour certaines fautes!

```
proc empty_stack() ;
  stack.h := 0 ; stack.foo := 2 ;
end empty_stack ;
proc push(x: natural) ;
  stack.a[stack.h] := x ; stack.h := stack.h + 1 ; stack.foo := stack.foo+1;
end push ;
proc pop() ;
  if stack.h > 0 then stack.h := stack.h - 1 ; stack.foo := 0 ; end if ;
end pop ;
proc top() ;
  if stack.foo = 1 then return stack.h ;    --!!! 💣
  elseif stack.h > 0 then return stack.a[stack.h] ; end if ;
end top ;
```