Université de Picardie Jules Verne
L@RIA, Bat Curi, 5 rue du moulin neuf
80000 Amiens
E-Mail: cerin@laria.u-picardie.fr
c.cerin@computer.org

– Génie Logiciel –
– Méthodes de programmation –
(fascicule 5 : Spec algébriques)

Licence d'informatique

Christophe Cérin

version 3.6 au 8/8/2002

Spécifications (idées de base)

Langage de spécification : $< Syn, Sem, Sat >$

- Syn, Sem sont des ensembles : domaines syntaxique (notations) et sémantique (univers d'objets) ;
- $Sat \subseteq Syn \times Sem$: relation de satisfaction (règles définissant les objets satisfaisant la spécification) ;

Soit $< Syn, Sem, Sat >$ si $Sat(syn, sem)$ alors syn est une *spécification* de sem et sem est un *spécificande* de syn.

Exemple : Notation BNF avec les règles de grammaire pour syn et l'ensemble des programmes que l'on peut former pour sem

Domaines syntaxique et sémantique

- **Domaine syntaxique** : ensemble de symboles : constantes, variables, connecteurs logiques + ensemble de règles grammaticales pour les combiner ;
  - $\forall x.P(x) \Rightarrow Q(x)$ (formule bien fondée du calcul des prédicats) ;
  - droites, flèches, lignes, icônes peuvent être des éléments d'un domaine syntaxique ;
- **Domaine sémantique** : tout langage de programmation (avec une sémantique bien précise) est un langage de spécification MAIS la réciproque n'est pas vraie (les spécifications ne sont pas toujours (souvent) exécutables) ;

Soit $< Syn, Sem, Sat >$: une implantation $prog \in Sem$ est correcte vis à vis de $spec \in Syn$ ssi $Sat(spec, prog)$.

Propriété requises des spécifications

- Soit $< Syn, Sem, Sat >$, $syn \in Syn$ n'est pas **ambigüe** ssi Sat fait correspondre syn à un et un seul sem de Sem ;
- Soit $< Syn, Sem, Sat >$, $syn \in Syn$ est **consistante** ssi Sat fait correspondre syn a un ensemble non vide (cela veut dire qu'il y a au moins une implantation qui satisfait la spécification).

Si la spécification est un ensemble de faits, la consistance implique que vous ne pourrez pas dériver Vrai et Faux à partir du même fait !

Comment vérifier par programme cette propriété de consistance ?

Spécification complète

Intuitivement, une spécification est complète si, deux termes u et v *quelconques* sont comparables.

En mathématique, une théorie T, et par suite le système d'axiomes qui la définit, est *complète* si elle est consistante et si pour toute formule P sans variable, on sait démonter soit P soit $\neg P$

On peut se contenter d'une propriété de **suffisante complétude**, voire **incomplète** $\rightarrow$ le programmeur a le choix des structures de données. (Ex: d'une liste chaînée formée de couples $< indice, element >$ pour implanter VECTEUR. Les opérations qui insèrent, $\ll$ changer(5,a), puis changer(10,b) $\gg$ ont le même résultat que $\ll$ changer(10,b), puis changer(5,a) $\gg$)

Ce qui est important, c'est que lorsqu'on applique une certaine f à u et v, on rend toujours le même résultat. Le fait qu'ils soient égaux (qu'on peut les comparer) n'est pas essentiel.

Propriétés des Spécifications

Une spécification a une **implantation biaisée** si elle spécifie des propriétés externes inobservables :

Exemple : une spécification du type abstrait << ensemble de >> qui garde la trace de l'ordre d'insertion d'un nouvel élément implanté avec une table de hachage [1]

1. Table de hachage : structure de donn´ees "tableau" ou un ´el´ement n est rang´e en position `n mod 511`

Prouver des Propriétés des $sem \in Sem$

– Méthodes formelles sont définies en terme de langage de spécification muni d'un système d'inférence ;
– À partir d'une spécification (ensemble de faits) on dérive de nouveaux faits en appliquant les règles d'inférence du système de déduction ;
– Quand on *prouve* une expression inférée par des faits, on prouve une propriété d'un élément $sem \in Sem$ satisfaisant la spécification (une propriété pas écrite explicitement)

$\downarrow$

Système d'inférence fournit à l'utilisateur une méthode formelle pour prédire le comportement d'un programme sans avoir à l'exécuter (theorem prouver & proof checkers).

Autres propriétés (temporelles)

Savety : quelque chose de mauvais ne peut jamais arriver (notions de fatalité et de propriété temporelle);

Liveness : quelque chose de bon peut éventuellement arriver (ex : terminaison d'un programme séquentiel) ;

Partial correctness : le programme produit le bon résultat si son exécution termine ; **Total correctness** : partial correctness + terminaison (le programme produit le bon résultat) ;

Les constituants d'une spécification algébrique :

– une **signature** qui décrit la syntaxe du type (nom + type des arguments – sortes) ;

– description des propriétés des opérateurs du type par des **axiomes** – donner une sémantique aux noms de la signature – exprimer des contraintes sur le domaine de définition de la fonction définie ;

La spécification du type Vecteur

SORTE : Vecteur
UTILISE : Element, Entier, Boolean
OPÉRATIONS :

- ieme : Vecteur $\times$ Entier $\rightarrow$ Element
- changer-ieme : Vecteur $\times$Entier $\times$ Element $\rightarrow$ Vecteur
- borne-sup : Vecteur $\rightarrow$ Entier
- borne-inf : Vecteur $\rightarrow$ Entier
- True : $\rightarrow$ Boolean
- $\wedge$: boolean $\times$ boolean $\rightarrow$ boolean

borne-inf(v) $\leq$ i $\leq$ borne-sup(v) $\Rightarrow$ ieme(changer-ieme(v,i,e),i) =e

(Pour tous les i compris entre. . .)

Spécifications Algébriques

Noter que l'écriture précédente n'est pas **purement** équationnelle puisque l'on exprime des contraintes définies par une **pré-condition** sur le domaine de définition de i-eme

$$\downarrow$$

C'est plus constructif $\rightarrow$ rapprochement avec spécifications orientés modèle. (Par contre, on n'a pas de construction if then else dans les axiomes – simulation avec plusieurs axiomes)

borne-inf(v) $\leq$ i $\leq$ borne-sup(v) &
borne-in(v) $\leq$ j $\leq$ borne-sup(v) & i $\neq$ j
$\Rightarrow$ ieme(changer-ieme(v,i,e),j)=ieme(v,j)

(Seul le i^{eme} élément à changé)

Spécifications Algébriques

Quelques définitions très importantes :

opérations internes : celles qui rendent un résultat d'une sorte définie.

observateur : une opération est un observateur si elle a au moins un argument d'une sorte définie et si elle rend un résultat d'une sorte prédéfinie : ieme.

Les générateurs : les opérations qui sont ! Par exemple, on construit les naturels à partir de la constante 0 et du générateur succ.

Propriétés des Spécifications Algébriques

1. N'y a t'il pas d'axiomes contradictoires ? (consistance) ;
2. A t'on donné suffisamment d'axiomes pour décrire toutes les propriétés du type que l'on veut spécifier au départ ? (complétude des axiomes)

$\downarrow$

critère trop fort

$\downarrow$

Complétude suffisante

Les axiomes doivent permettre de décrire une valeur d'une sorte prédéfinie pour toute application d'un observateur à un objet d'une sorte définie.

Complétude d'une Spécification Algébrique

Autrement dit : le membre droit d'une équation quelconque ne peut pas se réduire au membre gauche d'une autre équation (chaque équation caractérise des objets différents).

Propriété à vérifier : puisque les objets sont obtenus par les opérations internes, il faut écrire des axiomes qui définissent le résultat de la composition des observateurs avec toutes les opérations internes.

Comme les opérations ne sont pas définies partout $\rightarrow$ on doit pouvoir déduire une valeur pour tous les observateurs sur tout objet de sorte définie appartenant au domaine de définition de cet observateur.

Implantation de ieme

```
Vecteur
changer-ieme(v,i,e)
   v : Vecteur;
   i : int;
   e : Element;
{
   int j

   for(j=borne-inf(v);j<=borne-sup(v);j++)
      if(i==j) v[i] = e ;

   return(v);
}
```

Seule la i^{eme} case est changée. v est implanté au moyen d'un tableau (par contigüité $\neq$ par chaînage)

Deux cas particuliers des listes jouent un rôle important en informatique :

1. les **piles** où les données sont ajoutées et supprimées à une même extrémité (le sommet) ;
2. les **files** où les données sont ajoutées à une extrémité (la tête) et supprimées à une autre (la queue) ;

Les listes

- liste d'objets que l'on peut indexer avec les entiers (rang dans la liste) : $\lambda =< e_1, \cdots, e_n >$
- l'ordre des éléments est fondamental : ce n'est pas un ordre sur les éléments mais sur les places des éléments.
  $\Rightarrow$ une unique fonction succ telle que chaque place est accessible en l'appliquant de manière répétée à partir de la première place.
- chaque place a un contenu qui est un élément de la sorte considérée ;
- le nombre n d'éléments (donc de places) est appelé la *longueur* de λ ; $n = 0$, liste vide ; $succ^n(tete(\lambda))$ n'est pas défini ;

Les listes << itératives >> : signature & axiomes

Voici une définition << itérative >> pour le type de donnée liste.

SORTE : Liste, Place
UTILISE : Entier, Élément
OPÉRATIONS :

- liste-vide : $\rightarrow$ Liste
- accès : Liste $\times$ entier $\rightarrow$ Place
- contenu : Place $\rightarrow$ Élément
- longueur : Liste $\rightarrow$ Entier
- supprimer : Liste $\times$ Entier $\rightarrow$ Liste
- insérer : Liste $\times$ Entier $\times$ Élément $\rightarrow$ Liste
- succ : Place $\rightarrow$ Place

Les listes << itératives >> : signature & axiomes

- Liste et Place sont les sortes définies ;
- Les opérations accès et succ sont les opérations internes de Place ;
- Les opérations liste-vide, supprimer et insérer sont les opérations internes de Liste ;
- accès et longueur sont des observateurs de Liste ;

Les opérations de la signature ne sont pas définies partout : on exprime donc les pré-conditions suivantes :

VARIABLES

λ : Liste, k : Entier, e : Élément

PRÉ-CONDITIONS :

accès(λ, k) **est-défini-ssi** $1 \leq k \leq longueur(\lambda)$;

supprimer(λ,k) **est-défini-ssi** $1 \leq k \leq longueur(\lambda)$;

insérer(λ,k,e) **est-défini-ssi** $1 \leq k \leq longueur(\lambda) + 1$;

Les listes << itératives >> : axiomes pour "longueur"

AXIOMES :

longueur(liste-vide) $= 0$

$\lambda \neq$ liste-vide & $1 \leq k \leq longueur(\lambda) \Rightarrow$

longueur(supprimer(λ,k)) $=$ longueur(λ) $-$ 1

$1 \leq k \leq longueur(\lambda) + 1 \Rightarrow$

longueur(insérer(λ,k,e)) $=$ longueur(λ) $+$ 1

Ces axiomes définissent suffisamment et de manière consistante l'opération longueur (noter que l'on a 3 axiomes).

Les listes << itératives >> : axiomes

Plutôt que de décrire les axiomes de l'observateur acces, on se donne maintenant un nouvel observateur ieme tel que :

ieme : Liste $\times$ Entier $\rightarrow$ Élément
ieme(λ,k) = contenu(acces(λ,k))

Cette opération a les mêmes pré-conditions que l'opération acces.

AXIOMES :

$\lambda \neq$ liste-vide & $1 \leq k \leq longueur(\lambda) \& 1 \leq i < k \Rightarrow$
ieme(supprimer(λ,k),i) = ieme(λ, i)

$1 \leq k \leq longueur(\lambda) \& k \leq i \leq longueur(\lambda) - 1 \Rightarrow$
ieme(supprimer(λ,k),i)) = ieme($\lambda, i-1$)

$1 \leq k \leq longueur(\lambda) + 1 \& 1 \leq i < k \Rightarrow$
ieme(insérer(λ,k,e),i) = ieme(λ,i)

$1 \leq k \leq longueur(\lambda) + 1 \& k = i \Rightarrow$
ieme(insérer(λ,k,e),i) = e

$1 \leq k \leq longueur(\lambda) + 1 \& k < i \leq longueur(\lambda) + 1 \Rightarrow$
ieme(insérer(λ,k,e),i) = ieme(λ,i+1)

Les listes << récursives >> : signature & axiomes

Les opérations de base ne sont plus l'accès, l'insertion à la position k : opérations "tete" et "reste".

SORTE Liste, Place
UTILISE Élément, Entier
OPÉRATIONS
- liste-vide : $\rightarrow$ Liste
- tete : Liste $\rightarrow$ Place
- reste : Liste $\rightarrow$ Liste
- cons : Élément $\times$ Liste $\rightarrow$ Liste
- premier : Liste $\rightarrow$ Élément
- contenu : Place $\rightarrow$ Élément
- succ : Place $\rightarrow$ Place

Les listes << récursives >> : signature & axiomes

PRÉ-CONDITIONS

tete(λ) **est-défini-ssi** $\lambda \neq$ liste-vide

reste(λ) **est-défini-ssi** $\lambda \neq$ liste-vide

premier(λ) **est-défini-ssi** $\lambda \neq$ liste-vide

AXIOMES

premier(λ) = contenu(tete(λ))

reste(cons(e,λ)) = λ

premier(cons(e,λ)) = e

succ(tete(λ)) = tete(reste(λ))

Extensions du type Liste

On a souvent besoin d'effectuer sur les listes des opérations plus complexes, par exemple la concaténation, << retourner >> une liste, rechercher un << motif >> (suite d'éléments consécutifs). . .

Ces opérations se définissent à partir des opérations de base données ci-dessus (briques de base – noyau) $\rightarrow$ les **extensions** du type.

opération de concaténation

OPÉRATION

concatener : Liste $\times$ Liste $\rightarrow$ Liste

AXIOMES

longueur(concatener(l,l'))=longueur(l)+longueur(l')

$1 \leq i \leq longueur(l) \Rightarrow$ ieme(concatener(l,l'),i)=ieme(l,i)

$longueur(l) + 1 \leq i \leq longueur(l) + longueur(l') \Rightarrow$
ieme(concatener(l,l'),i)=ieme(l',i - longueur(l))

Concaténation de deux listes

On vérifie que les axiomes précédents sont suffisamment complets puisque la valeur des observateurs longueur et ieme peut être déduite pour concatener(l,l'), si on connaît la valeur de ces observateurs pour l et l'.

Opération de concaténation récursive

A partir des opérations liste-vide et cons :

AXIOMES

concatener(liste-vide,l)=l

concatener(cons(e,l'),l)=cons(e,concatener(l',l))

On vérifie, pour le deuxième axiome, que l'on << tend >> vers la forme syntaxique du premier axiome $\rightarrow$ terminaison est garantie.

Recherche d'un élément dans une liste

But : décrire une opération qui recherche si un élément est présent dans une liste et, dans ce cas retourne la place de cet élément : opération rechercher.

Convention : cette opération n'est pas définie si l'élément n'est pas présent dans la liste : opération est-present.

OPÉRATIONS

rechercher : Liste $\times$ Element $\rightarrow$ Place

est-present : Liste $\times$ Element : Booleen

PRÉ-CONDITION(S)

rechercher(l,e) **est-défini-ssi** est-present(l,e) = vrai

AXIOMES

est-present(liste-vide,e) = faux

$e = e' \Rightarrow$ est-present(cons(e,l),e') = vrai

$e \neq e' \Rightarrow$ est-present(cons(e,l),e')= est-present(l,e')

est-present(l,e)=vrai $\Rightarrow$ contenu(rechercher(l,e)) = e

Recherche d'un élément dans une liste

Il faut noter qu'en cas de répétition de e dans l, cette spécification ne précise pas de quelle occurrence de e on retourne la place ! On dit seulement que sous la condition où e est bien présent dans l alors le contenu de rechercher(l,e) (qui est bien un objet de type Place) c'est bien e !

Prendre un exemple avec plusieurs occurrences

Si on veut préciser qu'on recherche la première occurrence, on peut écrire, en utilisant le type liste récursive, les axiomes suivants :

AXIOMES

$e = e' \Rightarrow$ rechercher(cons(e,l),e')=tete(cons(e,l))

$e \neq e' \Rightarrow$ rechercher(cons(e,l),e') = rechercher(l,e')

 On a supposé que e était bien présent dans l.

Les files

On fait des adjonctions à une extrémité, les accès et les suppressions à l'autre extrémité.

SORTE File
UTILISE Boolean, Element
OPÉRATIONS
- file-vide : $\rightarrow$ File
- ajouter : File $\times$ Element $\rightarrow$ File
- retirer : File $\rightarrow$ File
- premier : File $\rightarrow$ Element
- est-vide : File $\rightarrow$ Boolean

PRÉ-CONDITIONS
- premier(f) **est-défini-ssi** est-vide(f) = faux
- retirer(f) **est-défini-ssi** est-vide(f) = faux

Les files

f est de sorte File et e de sorte Élément.

L'opération ajoute ajoute en << fin >>, l'opération retire retire en << tête >>.

AXIOMES

est-vide(f)=vrai $\Rightarrow$ premier(ajouter(f,e)) = e
est-vide(f)=faux $\Rightarrow$ premier(ajouter(f,e)) = premier(f)
est-vide(f)= vrai $\Rightarrow$ retirer(ajouter(f,e))=file-vide
est-vide(f)=faux $\Rightarrow$ retirer(ajouter(f,e)) = ajouter(retirer(f),e)

est-vide(file-vide) = vrai
est-vide(ajouter(f,e)) = faux

Complétude

On vérifie que la spécification est complète (reportez vous à la définition) vis à vis des opérations (il y a ici 2 observateurs : premier et est-vide et 2 opérations internes : ajouter et retirer)

$\downarrow$

Pour retirer et ajouter, remarquer que l'on n'a pas d'axiomes pour ajouter (ne pas définir ajouter en fonction de retirer n'importe comment $\rightarrow$ définition croisée ! – il faudrait exprimer ajouter avec un constructeur de file comme le cons des listes).

Pour premier et est-vide : la valeur de ces observateurs peut être déduite pour les opérations internes retirer(f) et ajouter(f,e) si l'on connaît la valeur de ces observateurs pour f.

Le type abstrait Ensemble

- collection clairement définie d'objets appelés éléments. $x \in A$
- description en *extension* $\{1, 2, 3\}$, $\emptyset = \{\}$ (l'ordre n'a pas d'importance)
- description en *compréhension* : $A = \{x | p(x)\}$, $\emptyset = \{x | (x \in N)$ et $2x + 1 = 1\}$ par exemple !
- **Le type de donnée** x-dont-le-carre : x-dont-le-carre $= \{x \in N \bullet 1 \leq x^2 \leq 9\}$
- $A \subseteq B$ – A est inclus dans B, A est une partie de B, A est un sous ensemble de B
- $P(A) = \{x | x \subseteq A\}$
- $A \cup B = \{x | (x \in A)\ ou\ (x \in B)\}$
- $A \cap B = \{x | (x \in A)\ et\ (x \in B)\}$
- commutativité, associativité, distributivité
- théorème : $\mid A \cup B \mid = \mid A \mid + \mid B \mid - \mid A \cap B \mid$

Ensemble

SORTE Ensemble
UITLISE Element, Boolean
OPÉRATIONS
ens-vide : $\rightarrow$ Ensemble
ajouter : Element $\times$ Ensemble $\rightarrow$ Ensemble
supprimer : Element $\times$ Ensemble $\rightarrow$ Ensemble
$\in$: Element $\times$ Ensemble $\rightarrow$ Boolean

Pour les axiomes, on a plusieurs choix pour l'opération ajouter dans le cas d'un élément déjà présent (idem pour supprimer)

$\downarrow$

Émettre un message ce qui signifie que les 2 opérations sont partiellement définies – les opérations sont sans effet – réaliser les deux cas précédents.

Axiomes pour Ensemble

Dans ce qui suit, ajouter un élément présent ou supprimer un élément absent laissent l'ensemble inchangé.

x,y sont des Éléments et e est de sorte Ensemble

AXIOMES

$x \in$ ens-vide = faux

$x = y \Rightarrow (x \in ajouter(y, e)) = vrai$

$x \neq y \Rightarrow (x \in ajouter(y, e)) = (x \in e)$

$x = y \Rightarrow (x \in supprimer(y, e)) = faux$

$x \neq y \Rightarrow (x \in supprimer(y, e)) = (x \in e)$

Enrichissement du type Ensemble

Enrichissons Ensemble par l'observateur card : Ensemble $\rightarrow$ Entier défini comme suit :

AXIOMES

card(ens-vide) = 0

$(x \in e) = vrai \Rightarrow card(ajouter(x, e)) = card(e)$

$(x \in e) = faux \Rightarrow card(ajouter(x, e)) = card(e) + 1$

$(x \in e) = vrai \Rightarrow card(supprimer(x, e)) = card(e) - 1$

$(x \in e) = faux \Rightarrow card(supprimer(x, e)) = card(e)$

Le type Multi-ensemble

Définition : des ensembles avec répétition(s). Pour la suite, x et y sont de sorte Element et m de sorte Multi-ensemble

SORTE Multi-ensemble
UTILISE Element, Boolean
OPÉRATIONS
- M-ens-vide : $\rightarrow$ Multi-Ensemble
- ajout : Multi-ensemble $\times$ Element $\rightarrow$ Multi-ensemble
- sup : Multi-ensemble $\times$ Element $\rightarrow$ Multi-ensemble
- app : Element $\times$ Multi-ensemble $\rightarrow$ Boolean

AXIOMES
- app(x,M-ens-vide) = faux
- $x = y \Rightarrow app(x, ajout(m, y)) = vrai$
- $x \neq y \Rightarrow app(x, ajout(m, y)) = app(x, m)$
- sup(M-ens-vide,x) = M-ens-vide
- $x = y \Rightarrow sup(ajout(m, y), x) = m$
- $x \neq y \Rightarrow sup(ajout(m, y), x) = sup(m, x)$

Les Multi-ensemble

Questions pour les TD : est-ce que l'on supprime toutes les occurrences d'un élément ? la spécification est-elle consistante ?

Que se passe t'il si nous ajoutons les axiomes :

$$x = y \Rightarrow app(x, sup(m, y)) = faux$$
$$x \neq y \Rightarrow app(x, sup(m, y)) = app(x, m)$$

Qu'en est'il de la complétude ?