VÉRIFICATION ET VALIDATION

Types abstraits algébriques

VV032

2012-09-13

Luc LAVOIE
Département d'informatique
Faculté des sciences

UNIVERSITÉ DE SHERBROOKE

Luc.Lavoie@USherbrooke.ca
http://pages.usherbrooke.ca/llavoie

Plan

- Abstraction et signatures
- Complétude suffisante partielle
- Noyau et principe de factorisation
- Types abstraits à plusieurs noyaux
- Quelques types abstraits classiques

Abstraction et signatures Présentation

- Il est nécessaire de définir de manière non ambigüe le comportement d'une routine.
- Il faut distinguer le modèle, l'interface et la représentation.
- Les types abstraits algébriques conviennent particulièrement bien à la description et à la modélisation des modules et des classes.

Abstraction et signatures
Exemple

```
sorte
          Pile;
utilise
          Booléen, Élément;
opérations
    (* manipulateurs : *)
          (* - constructeur : *)
          pile_vide : --> Pile;
          (* - transformateurs : *)
          empiler : Pile x Élément --> Pile;
          dépiler : Pile --> Pile;
    (* observateurs : *)
          sommet : Pile --> Élément;
          est_vide : Pile --> Booléen;
soit
          p : Pile; e : Élément;
antécédents
          dépiler(p) dssi non est_vide(p);
          sommet(p) dssi non est_vide(p);
axiomes
          est_vide(pile_vide) = vrai;
          est_vide(empiler(p,e)) = faux;
          est_vide(dépiler(empiler(p,e))) = est_vide(p);
          sommet(empiler(p,e)) = e;
          non est_vide(p) ==>
                    sommet(dépiler(empiler(p,e)) = sommet(p);
fin Pile.
```

Abstraction et signatures
Nomenclature et règles d'écriture

- Sorte définie :
  une sorte apparaissant à la rubrique «sorte».
- Sorte pré-définie :
  une sorte apparaissant à la rubrique «utilise».
- Profil d'une opération :
  la liste ordonnée des sortes déterminant les paramètres d'une opération et son résultat, notée

  par1 x ... x parn --> res

Abstraction et signatures
Nomenclature et règles d'écriture

- Observateur par rapport à une sorte *s* :
  une opération dont le profil comporte au moins un paramètre de sorte s mais dont le résultat n'est pas de sorte *s*.
- Manipulateur par rapport à une sorte *s* :
  une opération dont le résultat est de sorte *s* (le profil peut aussi comporter, ou non, un paramètre de sorte *s*).
- Constructeur par rapport à une sorte *s* :
  un manipulateur par rapport à *s* dont aucun des paramètres du profil n'est de sorte *s*; un constructeur est dit constant lorsqu'il ne comporte aucun paramètre.
- Transformateur par rapport à une sorte *s* :
  un manipulateur par rapport à *s* dont au moins un des paramètres du profil est de sorte *s*; corollaire : tout manipulateur est soit un constructeur soit un transformateur.

Abstraction et signatures
Nomenclature et règles d'écriture

- R1 :
  toute opération définie au sein d'un TA doit être soit un observateur en regard d'une sorte définie soit un manipulateur en regard d'une sorte définie.
- R2 :
  chaque sorte définie au sein d'un TA, doit être dotée d'au moins un constructeur.

Complétude suffisante partielle Définitions

- Consistance :
  une théorie (un TA) est dite consistante **ssi** pour toute formule P sans variable pouvant être démontrée vrai, il n'est pas possible de la démontrer fausse (la réciproque s'ensuivant).
- Complétude :
  une théorie (un TA) est dite complète ssi elle est consistante et si pour toute formule P sans variable on sait démontrer soit P, soit non P.
- Complétude suffisante :
  un TA est dit suffisamment complet ssi il est consistant et s'il est possible de déduire la valeur de toute application d'un observateur à un manipulateur ne comportant pas de variable.
- Complétude suffisante partielle :
  un TA est dit partiellement suffisamment complet ssi il est consistant et s'il est possible de déduire la valeur de toute application, dans les limites des domaines de définition des opérations en cause, d'un observateur à un manipulateur ne comportant pas de variable.

Complétude suffisante partielle Règle

- R3 :
  il suffit de définir tous les observateurs en regard de chacun des manipulateurs pour obtenir un TA partiellement suffisamment complet.

Complétude suffisante partielle Exemple

```
sorte
          File;
utilise
          Élément, Booléen;
opérations
  (* manipulateurs : *)
          (* - constructeur : *)
          file_vide : --> File;
          (* - transformateurs : *)
          ajouter : File x Élément --> File;
          retirer : File --> File;
  (* observateurs : *)
          est_vide : File --> Booléen;
          premier : File --> Élément;
soit
          f : File; e : Élément;
antécédents
          premier(f) dssi non est_vide(f);
          retirer(f) dssi non est_vide(f);
axiomes
          est_vide(file_vide) = vrai;
          est_vide(ajouter(f,e)) = faux;
          est_vide(retirer(ajouter(f,e)) = est_vide(f);
          premier(ajouter(file_vide,e)) = e;
          non est_vide(f) ==>
                    premier(ajouter(f,e)) = premier(f);
          non est_vide(f) ==>
                    premier(retirer(ajouter(f,e)) =
          premier(ajouter(retirer(f),e);
fin File.
```

Noyau et principe de factorisation
Définition

- Noyau d'une sorte :
  un noyau est un sous-ensemble des manipulateurs d'une sorte permettant d'en engendrer toutes les valeurs. En général on ne considère que les noyaux minimaux, un noyau est minimal si et seulement si on ne peut en retirer aucun élément sans qu'il cesse d'être un noyau. En général, il peut exister plusieurs noyaux pour une même sorte.

Noyau et principe de factorisation Règle

- R4 :
  il suffit de définir toutes les opérations n'appartenant pas au noyau en regard de celles du noyau pour obtenir un TA partiellement suffisamment complet.

Noyau et principe de factorisation
Application à la pile

```
noyau
          { pile_vide, empiler };
axiomes
          dépiler(empiler(p,e)) = p
          est_vide(pile_vide) = vrai
          est_vide(empiler(p,e)) = faux
          sommet(empiler(p,e)) = e
```

Noyau et principe de factorisation
Application à la file

```
noyau
          { file_vide, ajouter };
axiomes
          retirer(ajouter(file_vide,e)) = file_vide
          non est_vide(f) ==>
                    retirer(ajouter(f,e)) = ajouter(retirer(f),e)
          est_vide(file_vide) = vrai
          est_vide(ajouter(f,e)) = faux
          premier(ajouter(file_vide,e)) = e
          non est_vide(f) ==>
                    premier(ajouter(f,e)) = premier(f)
```

Noyau et principe de factorisation
Le vecteur

- Voir notes de cours

Types abstraits à plusieurs noyaux
Règle

- R5 :
  s'il existe plusieurs noyaux, lors de l'application R4, il est suffisant de définir chacune des opérations n'appartenant pas à un noyau en regard de celles d'un seul noyau... dans la mesure où les axiomes définissant l'équivalence entre les noyaux sont fournis.

Types abstraits à plusieurs noyaux
Exemple de la queue (1/2)

```
sorte
          Queue;
utilise
          Élément, Booléen;
opérations
  (* manipulateurs : *)
          (* - constructeur : *)
          queue_vide : --> Queue;
          (* - transformateurs : *)
          ajouterD : Queue x Élément --> Queue;
          ajouterF : Queue x Élément --> Queue;
          retirerD : Queue x Élément --> Queue;
          retirerF : Queue x Élément --> Queue;
  (* observateurs : *)
          est_vide : Queue --> Booléen;
          premier : Queue --> Élément;
          dernier : Queue --> Élément;
          longueur : Queue --> Entier;
noyaux
          {queue_vide, ajouter_D}, {queue_vide, ajouter_F};
```

Types abstraits à plusieurs noyaux
Exemple de la queue (2/2)

```
soit
          q : Queue; e, e1, e2 : Élément;
antécédents
          premier(q) dssi non est_vide(q);
          dernier(q) dssi non est_vide(q);
          retirerD(q) dssi non est_vide(q);
          retirerF(q) dssi non est_vide(q);
axiomes
  (* définition de ajouterD et ajouterF l'une p/r à l'autre *)
  (* d'où il s'ensuit l'équivalence des deux noyaux *)
          ajouterD(queue_vide,e) = ajouterF(queue_vide,e);
          ajouterD(ajouterF(q,e1),e2) = ajouterF(ajouterD(q,e2),e1);
  (* définition des observateurs *)
          est_vide(queue_vide) = vrai;
          est_vide(ajouterF(q,e)) = faux;
          premier(ajouterD(q,e)) = e;
          dernier(ajouterF(q,e)) = e;
          longueur(queue_vide) = 0;
          longueur(ajouterD(q,e)) = longueur(q) + 1;
  (* définition des autres opérations  *)
          retirerD(ajouterD(q,e)) = q;
          retirerF(ajouterF(q,e)) = q;
fin Queue.
```

Quelques types abstraits classiques

- Voir notes de cours