Epita:Algo:Cours:Info-Sup:Types abstraits

De EPITACoursAlgo.

Sommaire

- 1 Déclarations
  - 1.1 Signature
  - 1.2 Hiérarchie des types abstraits
  - 1.3 Propriétés d'un type abstrait

Déclarations

Nous définirons Un type abstrait à l'aide d'une signature et d'un ensemble d'axiomes.

Signature

La **signature** d'un type abstrait est composée des **types** et des **opérations**. Les **types** permettent de préciser à l'aide de plusieurs noms d'ensembles de valeurs (entier, arbrebinaire, etc.) le ou les type(s) que nous voulons définir. Par Exemple:

```
pile, liste, entier, graphe, arbre234
```

Les **opérations**, quand à elles, nomment les propriétés propres au(x) **type(s)** que l'on veut définir. Elles sont caractérisées par un identifiant(**nom**) et par la déclaration formelle (**profil**) de leurs arguments ainsi que du **type** de leur résultat. La déclaration des arguments se fait par le biais de leur **type**, ainsi:

```
insérer : liste x entier x élément ⟶ liste
```

déclare l'opération qui consiste à insérer un *élément* à la $i^{ème}$ place (définie dans ce cas par un *entier*) d'une *liste* et qui renvoie une *liste*.

Dans les algorithmes, nous utiliserons les **opérations** comme des fonctions, c'est à dire: l'identifiant accompagné entre parenthèses des arguments effectifs correspondants à ceux définis formellement. Par exemple, si nous voulons utiliser l'opération *insérer* précédente, en prenant des variables *l*, *i* et *e* de **types** respectifs *liste*, *entier* et *élément*, nous aurions :

```
insérer(l,i,e)
```

Pour les identifiants des opérations, tous les caractères et autres glyphes sont possibles exceptions faites de l'espace, des parenthèses ouvrantes, fermantes et du caractère de soulignement _ qui servent respectivement de séparateurs, à forcer les priorités de certaines opérations ou à positionner les arguments de l'opération. Les exemples suivants sont valides:

```
opérations

  factorielle : entier ⟶ entier
    puissance : entier x entier ⟶ entier
discriminent : entier x entier x entier ⟶ entier

ou bien

          _! : entier ⟶ entier
        _^_ : entier x entier ⟶ entier
          Δ : entier x entier x entier ⟶ entier
```

Notons, sur l'exemple précédent, que pour éviter une surcharge de parenthèses et lorsque l'on se réfère à des opérations classiques, comme la factorielle ou la puissance, le nom de l'opération peut en même temps donner la place des arguments à l'aide du caractère de soulignement *(_)*, comme dans *_!* et _^_. Dans ce cas, les paramètres effectifs viennent directement remplacer les caractères de soulignement (dans l'ordre de rencontre). De la même manière on constate que l'on peut utiliser *Δ* à la place de *discriminent*. Ces opérations peuvent alors être utilisées comme ceci:

```
x!
x^y!       (Arf!)
(x^y)!    (re-Arf!)
Δ(x,y,z)
```

Remarque: notez l'utilisation des parenthèses pour lever une éventuelle ambiguïté.

Une opération dont le profil ne demande pas d'argument est une constante, par exemple:

```
   0 : ⟶ entier
Faux : ⟶ booléen
  pi : ⟶ réel
```

Pour finir sur la signature, voici devant vos yeux ébahis un exemple complet, celle du type Booléen:

```
types

   booléen

opérations

   vrai : ⟶ booléen
   faux : ⟶ booléen
    non : booléen ⟶ booléen
     et : booléen x booléen ⟶ booléen
     ou : booléen x booléen ⟶ booléen
```

Hiérarchie des types abstraits

Nous avons la possibilité pour définir un type abstrait de réutiliser ceux précédemment définis. En effet, si un type possède des opérations manipulant des entiers, il est préférable de ne pas devoir redéfinir le type **entier**. Un certain nombre de types de base sont considérés comme définis, parmi eux nous trouvons les types **entier**, **réel**, **booléen**, etc.

Pour définir un type **vecteur**, par exemple, nous allons devoir réutiliser les types **entier** et **élément**. Les

données représentées par le type **élément** peuvent être n'importe quoi, des nombres, des vêtements, des roues de voiture *(et pourquoi pas des roues de voiture? C'est sympa les roues de voiture)* et le type **entier** représentera... Et bien, les entiers *(incroyable!)*. Se profile alors une hiérarchie de ces différents types, celui ou ceux que nous sommes en train de définir et celui ou ceux qui le sont déjà, ceux que nous allons donc réutiliser.

```
types

   vecteur

utilise

   entier, élément

opérations

   modifième :  vecteur x entier x élément ⟶ vecteur
        ième : vecteur x entier ⟶ élément
    borneinf : vecteur ⟶ entier
    bornesup : vecteur ⟶ entier
```

Dans ce cas, la signature du type **vecteur** est l'union des signatures des types **entier** et **élément** à laquelle viennent s'ajouter les nouvelles opérations qui caractérisent le type **vecteur**. Nous pourrons donc utiliser des opérations déjà définies sur les types utilisés comme, par exemple, l'addition sur les entiers. Ce qui pour l'opération **ième** (pour ne citer que celle-là) permettra d'utiliser en 2 ème argument la somme de deux entiers, par exemple:

```
ième(v,i+2)
```

Cette hiérarchie nous permet de dire qu'un type est:

- ***défini*** s'il est nouveau ("en conception", précisé dans **types**),
- ***prédéfini*** s'il existe déjà ("déjà conçu" et précisé dans **utilise**).

De même, nous dirons qu'une opération est:

- ***une opération interne*** si elle renvoie un résultat de type défini,
- ***un observateur*** si elle possède au moins un argument de type défini et si elle renvoie un résultat de type prédéfini.

Nous pourrions les définir autrement et dire qu'en fait les **opérations internes** sont celles-ci qui modifient l'état de la donnée elle-même, alors que les **observateurs** se contentent, comme leur nom l'indique, d'observer et de renvoyer une valeur se trouvant là où on leur demande de regarder.

Dans l'exemple précédent, *vecteur* est un **type défini**, *entier* et *élément* sont des **types prédéfinis** ce qui fait de *modifième* une **opération interne** et de *ième*, *borneinf* et *bornesup* des **observateurs**.

Propriétés d'un type abstrait

L'idée est de donner un sens aux noms de la signature. C'est à dire que lorsque l'on évoquera un type de donnée, on mesurera immédiatement toutes ses possibilités et ses limites (Si je dis pile, je sais que je ne peux pas faire le café avec). Dans ce cas, et si l'on veut se détacher de toute contingence matérielle, on énonce les

propriétés des opérations sous forme d'axiomes. Ce que l'on appelle plus communément **une définition algébrique**.

Le problème est de définir ce que font les **opérations internes**, pour le savoir il suffit de leur appliquer leurs propres **observateurs**. Les valeurs obtenues par ces derniers nous permettront de comprendre ce que fait l'**opération interne**.

Prenons par exemple l'application de l'observateur *ième* à l'opération interne *modifième*, cela donne les deux axiomes suivants :

```
borneinf(v) ≤ i ≤ bornesup(v) ⇒ ième(modifième(v,i,e),i)=e
borneinf(v) ≤ i ≤ bornesup(v) & borneinf(v) ≤ j ≤ bornesup(v) & i≠j
   ⇒ ième(modifième(v,i,e),j)=ième(v,j)
```

Le premier axiome dit que lorsque l'on appelle *modifième* pour un vecteur v, un entier i un élément e, l'élément e se retrouve positionné dans la $i^{ème}$ case du vecteur v. Cela est constaté par l'observateur *ième*, qui appliqué à l'aide du même entier i sur le nouveau vecteur (celui créé par *modifième*) est égal à e.

Le deuxième axiome définit, toujours à l'aide de l'observateur *ième* que seul la $i^{ème}$ case du vecteur v est modifiée par l'élément e et que toutes les autres, référencées par l'entier $j \neq i$, ont conservé les éléments qu'elles contenaient dans le vecteur v (avant changement).

*Note: La définition d'un type algébrique abstrait est donc la composée d'une **signature** et d'un **système d'axiomes** qui la caractérise.*

Une fois l'ensemble des axiomes établi, il faut vérifier deux choses:

- l'absence d'axiomes contradictoires appelée **consistance**. *Le contraire correspond au cas où l'application d'une même opération à des arguments identiques rend des valeurs différentes.*
- le fait d'avoir écrit suffisamment d'axiomes appelé **complétude** et qui correspond au fait de pouvoir déduire une valeur pour toute application d'un observateur à une opération interne.

Il existe des opérations qui ne sont pas décrites partout (personne n'est parfait). On les qualifie de *partielles*. Dans ce cas, et avant de décrire les axiomes utilisant ces opérations, il faut préciser leur domaine de définition. Ce que l'on fait à l'aide de **préconditions**.

Pour finir, reprenons l'exemple du *vecteur* et donnons sa définition complète, soit:

```
types

   vecteur

utilise

   entier, booléen, élément

opérations

            vect : entier x entier → vecteur
       modifième : vecteur x entier x élément → vecteur
            ième : vecteur x entier → élément
   estinitialisé : vecteur x entier → booléen
        bornesup : Vecteur → entier
        borneinf : vecteur → entier
```

```
précondition

   ième(v,i) est-défini-ssi Borneinf(v) ≤ i ≤ bornesup(v) & estinitialisé(v,i) = vrai

axiomes

   borneinf(v) ≤ i ≤ bornesup(v) ⇒ ième(modifième(v,i,e),i)=e
   borneinf(v) ≤ i ≤ bornesup(v) & borneinf(v) ≤ j ≤ bornesup(v) & i≠j
      ⇒ ième(modifième(v,i,e),j) = ième(v,j)

   estinitialisé(vect(i,j),k) = Faux
   borneinf(v) ≤ i ≤ bornesup(v) ⇒ estinitialisé(modifième(v,i,e),i) = Vrai
   borneinf(v) ≤ i ≤ bornesup(v) & borneinf(v) ≤ j ≤ bornesup(v) & i≠j
      ⇒ estinitialisé(modifième(v,i,e),j) = init(v,j)

   borneinf(vect(i,j)) = i
   borneinf(modifième(v,i,e)) = borneinf(v)

   bornesup(vect(i,j)) = j
   bornesup(modifième(v,i,e)) = bornesup(v)

avec
   vecteur v
    entier i,j,k
   élément e
```

La génération spontanée n'existant pas en Algorithmique, nous avons du ajouter l'opération *vect* qui crée un *vecteur* à partir de ses bornes (représentées par deux entiers). D'autre part, l'opération *ième* étant partielle, elle n'est en effet pas définie sur un indice auquel nous n'aurions pas précédemment affecté d'élément (à l'aide de *modifième*), nous avons donc dû rajouter une **opération auxiliaire** *estinitialisé* dont le seul but est de permettre l'écriture d'une **précondition** sur *ième*, autrement dit de préciser le domaine de définition de *ième*.

Pour conclure à l'aide de cet exemple, sur la conception et la compréhension des types algébriques abstraits, la définition du type vecteur est **consistante** et **complète** pour les raisons suivantes :

- Il n'existe aucun axiome en contradiction avec un autre.
- Tout vecteur est le fruit d'une opération *vect* et d'une série d'opérations *modifième*, effets constatés par *estinitialisé*, *bornesup* et *borneinf* dans tous les cas et par *ième* quand la **précondition** est satisfaite.

(Christophe "krisboul" Boullay)

- Dernière modification de cette page le 7 janvier 2013 à 12:03.