Université de Liège

INFO0946

Introduction à la Programmation

GameCode : Un exercice dont vous êtes le Héros · l'Héroïne

TAD – Ensemble d'Entiers

Benoit Donnet Simon Liénardy

25 février 2021

Préambule

Exercices

Dans ce GameCode, nous vous proposons de suivre pas à pas la résolution d'un exercice concernant la définition d'un TAD représentant les ensembles d'entiers.

Il est dangereux d'y aller seul [1] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

1. Référence vidéoludique bien connue des Héros.

5.1 Énoncé

On demande de créer un type abstrait pour un **ensemble d'entiers positif** (i.e., nombre naturel). Ce type devra disposer des opérations suivantes :

- créer un ensemble vide ;
- déterminer si un ensemble est vide ;
- déterminer le cardinal (i.e., le nombre d'éléments) de l'ensemble ;
- déterminer si un ensemble est un sous-ensemble d'un autre ;
- déterminer si un élément appartient ou non à un ensemble ;
- déterminer si deux ensembles sont égaux ;
- déterminer l'union de deux ensembles d'entiers ;
- déterminer l'intersection de deux ensembles d'entiers ;
- ajouter un entier à l'ensemble ;
- retirer un entier de l'ensemble ;
- obtenir la différence de deux ensembles, i.e., un ensemble qui contient les éléments qui sont dans l'ensemble A mais pas dans l'ensemble B ;
- obtenir la différence symétrique de deux ensembles, i.e., un ensemble qui contient les éléments qui ne sont soit dans l'ensemble A, soit dans l'ensemble B mais pas dans les deux ;

5.1.1 Méthode de Résolution

Nous allons procéder de la façon suivante :

1. Spécification abstraite (Sec. 5.2) ;

5.2 Spécification Abstraite

Avant de commencer la moindre implémentation, il faut proposer une spécification abstraite pour notre structure de données.

Si vous voyez de quoi on parle, rendez-vous à la Section 5.2.3
Si vous ne savez pas ce qu'est une spécification abstraite, consultez la Section 5.2.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 5.2.2

5.2.1 Structure de la spécification d'un TAD

5.2.1.1 Généralités

Un *Type Abstrait de Données* (TAD), c'est la description d'un ensemble de valeurs ainsi que les opérations permises sur ces valeurs. Passons maintenant aux exercices ! Ou peut-être pas... Précisons qu'on s'intéresse aux propriétés des opérations et non à la manière dont elles sont implémentées. On sépare le *quoi* (spécifications) du *comment* (implémentation).

La spécification d'un TAD contient de nombreux éléments qui sont rappelés ici :

Type C'est le nom du TAD. On utilise le plus souvent un nom anglais qui débute par une lettre capitale. On privilégie un nom en rapport avec ce que l'on est en train de décrire, évidemment. Attention : Un nom ne confère pas *magiquement* des propriétés au type défini : c'est la sémantique qui se charge de décrire les propriétés du TAD ! Par contre, trouver un nom adéquat permet davantage de **lisibilité**.

Utilise On liste ici tous les autres TAD nécessaires à la spécification de celui qui est en train d'être décrit. C'est important : on pourra alors *utiliser* les opérations des TAD mentionnés dans la liste lors de l'écriture des axiomes. Certains types sont considérés comme **bien connus** et peuvent être utilisés *ad libitum* [a]. C'est le cas par exemple de Natural, Integer et Boolean. La généricité d'un TAD s'exprime avec l'utilisation du type Element.

Opérations On liste ici les opérations du TAD, en utilisant une *notation fonctionelle* qui indique le nom de chaque opération ainsi que les objets qu'elle reçoit et qu'elle renvoie. Selon le type de ces objets, on peut classer les opérations en trois catégories : les **constructeurs**, les **observateurs** et les **transformateurs**. Pour savoir exactement comment procéder pour décrire une opération, reportez-vous à la Sec. 5.2.1.2.

Préconditions Parfois, certaines opérations n'ont aucun sens si elles sont appliquées à certaines valeurs particulières du TAD. Dans ce cas, on peut restreindre le domaine de définition de ces opérations à l'aide de préconditions. On obtient alors ce que l'on appelle des **opérations partielles**, parce qu'elles ne sont que partiellement définies sur le domaine de définition.

Axiomes C'est la partie la plus importante du TAD, le plat de résistance ! Grâce aux axiomes, on peut enfin comprendre la signification des différentes opérations. Il faut surtout faire attention à avoir assez d'Axiomes (on parle de *complétude*) et que ceux-ci n'entrent pas en contradiction (on parle de *consistance*). Comment être complet et consistant ? Le sujet est abordé à la Sec. 5.2.1.4.

→ **Type**, **Utilise** et **Operation** forment ce que l'on appelle la *signature* du TAD (ou syntaxe).

→ Les **Préconditions** et les **Axiomes** forment la *sémantique* du TAD.

a. « À volonté », en latin.

Alerte : Rupture d'Abstraction

Aviez-vous remarqué que le « A » de TAD signifie « Abstrait » ? Vous pensez que je vous prends pour un jambon à mentionner cela dans un cadre mis en évidence. Et pourtant...

Il faut faire très attention à **ne pas** mentionner d'informations au sujet d'une quelconque implémentation du TAD (cela arrive *hélas* chaque année !). Cela constitue une *Rupture d'Abstraction* qui est éliminatoire tant dans un projet qu'à un examen !

5.2.1.2 Comment Définir une Opération ?

Voici comment se présente la description d'une opération. Il est impératif de suivre ce schéma :

nom de l'Opération : type $\text{arg}_1 \times$ type $\text{arg}_2 \times \cdots \times$ type $\text{arg}_n \rightarrow$ type

On commence par le nom de l'opération, suivi de deux points. Les type arg_i représentent les types des arguments de l'opération. On ajoute ensuite une flèche et on indique alors le type de l'objet renvoyé par l'opération. Par exemple, pour le TAD Vector, on avait l'opération *set* :

$$\text{set} : \text{Vector} \times \text{Integer} \times \text{Element} \rightarrow \text{Vector}$$

L'opération *set* prend en argument un $Vector$ (qui sera modifié), un entier ($Integer$ – qui représente une position dans le $Vector$), ainsi qu'un élément ($Element$ – qui sera inséré par set à l'endroit spécifié par l'entier). Les précisions entre parenthèses dans la phrase précédente ne sont disponibles qu'une fois les axiomes correctement spécifié : c'est là leur but !

5.2.1.3 Typologie des Opérations

On peut regrouper les opérations en plusieurs catégories :

Les Constructeurs Ce sont les opérations pour lesquelles le TAD n'apparait que du côté des résultats. Comme le nom l'indique, ce sont des opérations qui permettent de créer une valeur du type du TAD.

Les Observateurs Ce sont des opérations pour lesquelles le TAD n'apparait que dans la liste des arguments. Un autre type de valeur est renvoyé. C'est une valeur observée.

Les Transformateurs Pour ces opérations, le TAD apparait tant du côté des arguments que du côté du résultat. La (ou les) valeurs·s de type TAD passée·s en arguments sont transformée·s en la valeur renvoyée en résultat.

Les Constantes Ce sont des opérations sans arguments. Incidemment, ce sont donc des cas particuliers de constructeurs.

On regroupe parfois les constructeurs et les transformateurs en **opérations internes** que l'on distingue donc des observateurs. Cette classification est intéressante lorsque l'on vérifie que les axiomes correspondants aux opérations sont consistants et complets.

Alerte : Conseils pour procéder

Il faut d'abord choisir un nom qui représente correctement l'opération que l'on souhaite décrire. Ensuite, il est sans doute plus facile de réfléchir à la catégorie de l'opération : est-ce un constructeur ? Un observateur ? Un transformateur ? Cela donne déjà quelques indices sur les types des paramètres et des résultats. Ensuite, il faut se demander : de quoi ai-je besoin pour réaliser l'opération ? Il suffit alors de lister les paramètres et de les séparer avec le caractère $\times$. Il faut ensuite bien vérifier que chaque type d'argument, s'il n'est pas le TAD lui-même, est bien listé dans la section **Utilise** de la signature.

5.2.1.4 Description Complète et Consistantes des Axiomes

Écrire des axiomes demande de l'abstraction (en même temps, la signification de TAD devrait vous avoir mis la puce à l'oreille [2]). Voici une série de conseils à mettre en pratique [3] :

2. Déso si c'est pas le cas...
3. Les exemples sur lesquels nous nous appuyons dans la suite sont tirés du cours théorique. Voir Chapitre 5, Slides 23 → 32

1. Réfléchir à la question : mon opération est-elle pertinente pour toutes les valeurs possibles et imaginables de ses arguments ? Si ce n'est pas le cas, il faut écrire des **Préconditions**. Il y a deux manières de les écrire.

   Soit :

   $$\forall i, Cond(i), nomOperation(\text{arg}_1, \text{arg}_2, \ldots, i, \ldots, \text{arg}_\text{n})$$

   On souhaite limiter l'opération pour l'un de ses arguments, on écrit une quantification universelle (cfr. Chapitre 1, Slides 12 $\rightarrow$ 16) introduisant une variable liée (cfr. Chapitre 1, Slides 22 $\rightarrow$ 24) qui sera du même type que l'argument à limiter. $Cond(i)$ est une condition que doit satisfaire toutes les valeurs de l'argument à limiter pour que l'opération soit permise. Exemple :

   $$\forall \text{i}, 0 \leq \text{i} < \text{size(v)}, \text{get(i, v)}$$

   signifie que l'opération *get* n'est permise que si l'entier i est compris entre 0 et $size(v) - 1$. Savoir à quoi exactement servent i et v n'est possible qu'en lisant la suite des axiomes.

   Ou Bien :

   $$nomOperation(\text{arg}_1, \text{arg}_2, \ldots, i, \ldots, \text{arg}_\text{n}) \text{ is defined iff } Cond(i)$$

   Les plus sagaces auront remarqué que cela consiste juste à déplacer la condition $Cond(i)$. La signification est (forcément) la même que ci-dessus. Exemple identique au précédent :

   $$\text{get(i, v) is defined iff } 0 \leq \text{i} < \text{size}$$

   Quelle variante choisir ? La préférence va pour la 2$^\text{e}$ formulation qui a l'avantage d'être explicite, qualité rare quand on parle de TAD.

   Note : La première ligne des préconditions sera constituées de l'information que tous les autres arguments $\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_\text{n}$ peuvent prendre n'importe quelle valeur. Cela se note ainsi :

   $$\forall \text{arg}_1 \in \text{Type arg}_1, \forall \text{arg}_2 \in \text{Type arg}_2, \ldots, \forall \text{arg}_\text{n} \in \text{Type arg}_\text{n}$$

2. Passer aux axiomes proprement-dits. Tous les axiomes présentent la forme suivante :

   Terme de gauche = Terme de droite

   Commençons par regarder le **terme de gauche**. Celui-ci est le plus souvent constitué par la combinaison d'une **opération interne** avec un **observateur**. Ce n'est pas tout le temps le cas (voir plus bas) mais la plupart du temps, on commence par cette combinaison. On sait déjà qu'un observateur prend en argument une valeur de type TAD alors qu'une opération interne produit une valeur de type TAD. Donc, logiquement, c'est l'observateur qui va prendre en argument le résultat de l'opération interne. Attention à ne pas écrire des axiomes trop spécifiques : les arguments de l'opération interne peuvent prendre **n'importe quelle valeur permise par les préconditions**. Et on se limite à cela pour le terme de gauche ! En conséquence, toute la difficulté est d'écrire le terme de droite.
   Exemple, dans :

   $$\text{size(set(v, i, e))} = \ldots$$

On a bien mis un v, un i et un e quelconque. On n'a pas mis, que sais-je ? 0 à la place du i : ce serait trop spécifique. D'ailleurs, on rappelle en début des axiomes que ces arguments peuvent prendre n'importe quelle valeur permise :

$$\forall \text{arg}_1 \in \text{Type arg}_1, \forall \text{arg}_2 \in \text{Type arg}_2, \ldots, \forall \text{arg}_n \in \text{Type arg}_n$$

en l'occurrence :

$$\forall v \in vector, \forall i \in Integer, \forall e \in Element$$

3. Décrire le **terme de droite** des axiomes à l'aide de la **récursion**. Il faut maintenant décrire le résultat de l'opération décrite dans le terme de gauche de manière récursive. Comme d'habitude, il y a un ou plusieurs cas de base ainsi que un ou plusieurs cas récursifs. Si l'opération doit avoir un comportement déterminé pour une valeur particulière d'un de ses argument, c'est dans la définition du terme de droite que l'on discute ce cas précis.

   Reprenons l'exemple du TAD Vector :

   size(set(v, i, e)) = ...

   Il faut exprimer la taille d'un Vector v après modification de l'élément e à la position i. Dans notre esprit, il est clair que v est le $Vector$, que i est un index, que e est une élément de remplacement et que *set* est une opération de transformation. Il faut bien se rappeler qu'un lecteur quelconque ne comprendra tout cela qu'après avoir lu le reste des axiomes ! Puisque l'axiome a toute les chances d'être un cas récursif, on peut déjà recopier l'appel à l'observateur *size*. Par contre, on essaie de simplifier l'argument de cet observateur : c'est l'idée de la récursion, de converger vers un cas de base. Le plus souvent, on essaie de faire disparaitre l'occurrence de l'opération interne présente dans le terme de gauche (ici, c'est *set*). Notez que ce n'est pas toujours possible. En ce qui concerne cet axiome particulier, on veut exprimer que modifier un élément à une position donnée *ne modifie pas* la taille du $Vector$. On aura donc l'axiome :

   size(set(v, i, e)) = size(v)

   On voit bien que $size(v)$ est plus simple que $size(set(v, i, e))$. L'axiome exprime en outre que l'opération est indépendante des valeurs de i et de e, puisqu'ils ne sont pas présents dans le terme de droite.

4. Vérifier que l'on est bien **consistant** et **complet**.

   Le cours mentionne deux méthodes :

   - S'assurer que le comportement du TAD est bien décrit. C'est une méthode intuitive. Il faut avoir l'habitude. *Passez votre chemin en première lecture.*
   - Écrire les axiomes qui définissent le résultat de la composition de tous les observateurs avec toutes les opérations internes. C'est la méthode recommandée. Il est recommandé de travailler avec un tableau à double entrée [4] : les lignes (resp. les colonnes) listent les observateurs (resp. les opérations internes). Dès que **tous les cas** [5] de la combinaison d'un observateur et d'une opération interne sont décrits, vous cochez la case correspondante. Il faut avoir coché toutes les cases du tableau. Simple.

   Voici une variante [6] de la deuxième méthode. Elle se base sur une autre classification des opérations. Elle distingue :

   - Les constructeurs universels : c'est un ensemble minimum d'opérations qui permettent de créer toutes les valeurs possibles de TAD. Exemple pour le TAD *List* (cfr. Chapitre 6, Slides 13 → 17) : *empty_list* et *add_at*. N'importe quelle liste peut être obtenue par la combinaison de ces deux opérations ;

4. C'est d'ailleurs comme cela que les TADs des projets sont corrigés
5. Ne vous contentez pas des cas de base !
6. Les variants sont dans l'air du temps...

- Les autres opérations.

Il faut combiner chacune des autres opérations avec chacun des constructeurs universels. Dans l'exemple des listes, on aurait par exemple plusieurs axiomes qui combinent *remove* et *add_at*. Il se présenteraient comme ceci :

$$\begin{array}{lll} \text{remove}(\text{add\_at}(L, i, e), j) & = L & \text{si } i = j \\ & = \text{add\_at}(\text{remove}(L, j-1), i, e) & \text{si } i < j \\ & = \text{add\_at}(\text{remove}(L, j), i-1, e) & \text{si } i > j \end{array}$$

Attention, dans le terme de gauche, il y a bien deux positions. On utilise alors deux variables distinctes : i et j. Les relations possibles entre i et j sont discutées dans trois axiomes séparés. On ne se contente pas du cas où $(i = j)$ qui est le plus facile à écrire. La première ligne signifie que supprimer l'élément que l'on vient d'ajouter revient à ne rien faire du tout. Les deux autres sont des cas récursifs. Les indices i et j sont modifiés dans l'appel récursif puisque dans le terme de gauche, on retire après un ajout alors qu'on ajoute après un retrait dans le terme de droite. Il faut donc gérer le décalage des positions [7]. Vu les conditions des deux derniers cas et la mise à jour des indices, on voit qu'on converge bien vers le cas de base $i = j$. En quoi est-ce que le cas récursif est « plus simple » ? Dans la mesure où, dans le terme de gauche, *remove* est appliquée au résultat de *add_at* et dans le terme de droite, elle n'est plus appliquée que sur la liste L, on peut dire que c'est plus simple.

5. Réduire le nombre d'axiomes. On peut parfois parvenir à réduire le nombre d'axiome. Par exemple, si une opération C peut s'exprimer en fonction de deux autres opérations A et B (ce sont soit tous des observateurs, soit tous des transformateurs), on peut avoir un axiome du type :

$$C(args) = f(A(args), B(args))$$

Où f est la fonction qui combine le résultat de A et B. Dans le tableau à double entrée présenté précédemment, la ligne (resp. la colonne) de C peut être cochée dès que les lignes (resp. colonnes) de A et de B sont remplies. On peut donc réduire le nombre d'axiomes. Exemple pour le TAD *List* :

add_first(L, e) = add_at(L, 0, e)
add_last(L, e) = add_at(L, length(L), e)

Le TAD *List* possède 4 observateurs. Cex deux axiomes permettent donc de ne pas en écrire 6 (4 *obs* × 2 *op* − 2 *axiomes*).

Alerte : Allergie à la récursivité

Une erreur courante est de ne pas assez être récursif dans l'écriture des axiomes. il faut se dire que la récursivité, c'est plutôt **la norme**. Il faut s'inquiéter si les principales opérations du TAD ne sont pas axiomatisées récursivement !

Un cas particulier consiste à sur-spécifier le terme de gauche de l'axiome, en se focalisant sur des valeurs particulières des arguments de l'opération interne. L'axiome ne documente pas bien l'opération car il se focalise sur un sous-ensemble de valeurs possible. Le risque d'être incomplet est alors très grand !

Suite de l'Exercice

À vous ! Spécifiez de manière abstraite la structure de données et passez à la Sec. 5.2.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 5.2.2

7. Un petit schéma a été réalisé pour ne pas nous tromper...

5.2.2 Indice

Commencez par bien relire l'énoncé. Il contient des informations importantes relatives aux différentes fonctionnalités souhaitées.

Ensuite, il convient de réfléchir à la syntaxe de la spécification abstraite (i.e., la signature du TAD). C'est, normalement, une étape assez simple car tous les éléments sont dictés par l'énoncé.

Enfin, il faut terminer avec la sémantique de la spécification abstraite (i.e., préconditions et axiomes). Soyez à l'aise avec la récursivité. Assurez-vous aussi que vos axiomes sont complets et consistants.

Suite de l'Exercice

À vous ! Spécifiez de manière abstraite la structure de données et passez à la Sec. 5.2.3.

5.2.3 Correction de la Spécification Abstraite

Il faut préciser :

- la syntaxe du TAD ;
- la sémantique du TAD.

5.2.3.1 Syntaxe du TAD

On va appeler notre TAD *IntSet* pour « Integer Set ». On reprend la liste des opérations demandées dans l'énoncé et on détaille leurs arguments ainsi que le type de leurs résultats. Certaines opérations renvoient des *Natural* ou des *Boolean*, on n'oublie donc pas de les ajouter dans la section « Utilise ».

Il vient :

```
Type:
  IntSet
Utilise:
  Natural, Boolean
Opérations:
  empty_set: → IntSet
  is_empty: IntSet → Boolean
  cardinal: IntSet → Natural
  subset: IntSet × IntSet → Boolean
  belong_to: IntSet × Natural → Boolean
  equal: IntSet × IntSet → Boolean
  union: IntSet × IntSet → IntSet
  intersection: IntSet × IntSet → IntSet
  add: IntSet × Natural → IntSet
  remove: IntSet × Natural → IntSet
  difference: IntSet × IntSet → IntSet
  difference_sym: IntSet × IntSet → IntSet
```

Suite de l'exercice

Vous pouvez maintenant

- passer à la sémantique du TAD ;

5.2.3.2 Sémantique du TAD

Préconditions

Les préconditions sur les opérations sont assez limitées. On va juste demander de ne pouvoir retirer un élément que si l'ensemble n'est pas vide et qu'il contient l'élément à retirer. Soit :

```
Préconditions:
   ∀ x ∈ Natural, ∀ s ∈ IntSet
   remove(s, x) is defined iff is_empty(s) = False ∧ belong_to(s, x) = True
```

Axiomes

On a donc 5 observateurs et 7 opérations internes. Ce qui devrait donner :

5 Observateurs × 7 Opérations Internes = 35 Axiomes

Va-t-on écrire 35 Axiomes (au minimum [8]) ? Essayons que non. En fait, il y a deux méthodes pour combiner les axiomes et être complet et consistant (voir la Sec. 5.2.1.4) :

- combiner les observateurs et les opérations internes, que nous appellerons **Première Méthode** ;
- combiner les constructeurs universels et les autres opérations (ici, ce serait *empty_set* et *add*), que nous appellerons **Seconde Méthode**.

En fait, ici, il vaut mieux utiliser la **Seconde méthode** ! Pourquoi ? Ben le meilleur moyen de le voir, c'est de commencer avec la première méthode et voir des axiomes du genre :

```
cardinal(intersection(s1, s2)) = ...
subset(difference(s1, s2), s3) = ...
```

De pas savoir comment développer et de choisir une autre stratégie.
Avec la seconde méthode, on a

2 Constructeurs Universels × 10 Autres Opérations = 20 Axiomes

C'est déjà mieux !

De plus il vaut mieux essayer de tenir compte des propriétés des ensembles d'entiers et d'exprimer certains observateurs (respectivement transformateurs) en fonction des autres observateurs (respectivement transformateurs). Pour cela, on ne peut que vous conseiller de jeter un coup d'œil à l'algèbre des parties d'un ensemble, ainsi que le cours MATH2019 (Chapitre 1, Sec. 1.3 et les slides correspondants). Et à relire l'énoncé (Sec. 5.1) au cas où les opérations ne sont pas nommées de la même manière dans ces deux documents.

Appliquons la seconde méthode :

```
Axiomes:
∀ x, y ∈ Natural, ∀ s, s1, s2 ∈ IntSet

  // Propriété fondamentale des ensembles:
```

8. Il y a plusieurs cas parfois !

```
add(s, x) = s si belong_to(s, x)

// Simplification des observateurs
is_empty(s) = (cardinal(s) = 0)
equal(s1, s2) = subset(s1, s2) ∧  subset(s2, s1)

// Simplification des transformateurs
differrence_sym(s1, s2) = difference(union(s1, s2), inter(s1, s2))

// Observateur cardinal
cardinal(empty_set()) = 0
cardinal(add(s, x)) = cardinal(s1) + 1 si not(belong_to(s, x))
cardinal(add(s, x)) = cardinal(s1) si belong_to(s, x)

// Observateur subset
subset(s, empty_set()) = True
subset(s1, add(s2, x)) = belong_to(s1, x) ∧ subset(s1, s2)

// Observateur belong_to
belong_to(empty_set(), x) = False
belong_to(add(s, x), y) = belong_to(s, y) ∨ (x = y)

// Transformateur union
union(s, empty_set()) = s
union(s1, add(s2, x)) = add(union(s1, s2), x)

// Transformateur intersection
intersection(s, empty_set()) = empty_set()
intersection(s1, add(s2, x)) = intersection(s1, s2) si ¬belong_to(s1, x)
intersection(s1, add(s2, x)) = add(intersection(s1, s2), x)
                               si belong_to(s1, x)

// Transformateur remove
remove(add(s, x), y) = s si x = y ∧ ¬belong_to(s, x)
remove(add(s, x), y) = remove(s,x) si x = y ∧ belong_to(s, x)
remove(add(s, x), y) = add(remove(s, y), x) si x ≠  y ∧ belong_to(s, y)
remove(add(s, x), y) = add(s, x) si x ≠  y ∧ ¬belong_to(s, y)

// Transformateur difference
difference(s, empty_set) = s
difference(s1, add(s2, x)) = difference(remove(s1, x), s2)
                             si belong_to(s1, x)
                             difference(s1, add(s2, x)) = difference(s1, s2)
                             si ¬belong_to(s1, x)
```

Quelques remarques

- Pour $belong\_to(s, x)$, on ne peut pas écrire $x \in s$. Ce serait remplacer un nom par un symbole ! Qui,

certes, correspond à une notation ensembliste mais on devine que $IntSet$ est un ensemble de $Natural$ une fois que l'on a lu tous les axiomes ! Donc « $x \in s$ » ne veut pas dire grand chose avant d'avoir fini d'écrire tous les axiomes.

- $subset(s1, s2)$ représente la notation ensembliste $s2 \subset s1$. L'ordre des opérandes est inversé.
- L'ordre des opérandes des autres opérations d'arité 2 [9] n'est pas inversé. Donc $difference(s1, s2)$ correspond bien à $s1 \setminus s2$. Et en plus, les autres opérations sont symétriques...
- On ne combine pas *remove* and *empty_set* parce que c'est interdit par les préconditions.
- On remarque d'ailleurs que lorsque *remove* apparait dans un terme de droite, une condition teste que sa précondition est bien respectée.
- L'axiome $add(s, x) = s$ si $belong\_to(s, x)$ permet de prendre moins de précautions en écrivant d'autres axiomes (ceux d'*union* par exemple).
- On a bien une vingtaine d'axiomes, comme prévu, bien que les simplifications au début des axiomes nous en ont fait économiser quelques uns.
- On voit que tous les axiomes qui concernent deux ensembles positionnent l'opération *add* uniquement à droite. Jamais à gauche. En gros, le but est de converger avec le terme de droite vers le cas de base.
- Avec cette seconde méthode, on a l'impression qu'on exprime toutes les transformateurs en fonction de *empty_set* et *add*. C'est exactement ce qu'on fait ! Et puisque les observateurs sont correctement définis pour ces deux opérations, le TAD est complet (et consistant).
- Il y a d'autres moyens de procéder. Si vous avez d'autres propositions, discutez-en sur eCampus.

Suite de l'exercice

Vous pouvez maintenant

- revenir à la syntaxe du TAD ;

9. En mathématiques, l'*arité d'une fonction* est le nombre d'arguments ou d'opérandes qu'elle requiert