Introduction aux Types Abstraits Algébriques Définitions et Résultats

Gilles Bernot

LIENS, CNRS URA 1327
45 rue d'Ulm
75230 PARIS Cedex 05

Avertissement : ce poly n'est pas « self-contained ». Il n'a aucun sens sans le cours correspondant. Il n'est qu'une suite de définitions/énoncés/preuves qui ne signifie rien sans l'intuition informatique, issue du génie logiciel, qui va avec. Les résultats donnés ici sont trop triviaux, et insuffisamment généraux, pour avoir un quelconque intérêt mathématique en soi.

Les exercices donnés dans ce poly ne sont généralement pas les mêmes que ceux donnés en cours ; ils peuvent aider à bien assimiler le cours.

Chapitre A :
Présentation d'un Type Abstrait Algébrique

1 Introduction

Un « Type Abstrait Algébrique » (TAA) peut être considéré en première approche comme la donnée d'une *syntaxe* décrivant une structure de données munie d'opérations sur cette structure et d'une *sémantique* déterminant (généralement via une notion de « validation ») une classe de modèles associée à cette syntaxe. Tout ceci est défini en utilisant des techniques algébriques et peut mettre en jeu des aspects hiérarchiques combinant des « modules de spécification » et/ou diverses autres « primitives de structuration ».

2 Signature d'un TAA

La signature d'un TAA est la partie de la syntaxe qui déclare les divers éléments mis en jeu dans la spécification du TAA. Dans ce qui suit, on n'expose que la théorie la plus simple de spécification algébrique, telle que décrite pour la première fois dans les travaux du groupe ADJ en 1978. Il faut bien comprendre que les TAA ne sont pas réduits à une seule théorie. Il s'agit plutôt d'un ensemble de théories ayant de nombreux traits communs (l'usage systématique de *spécifications formelles*, avec une sémantique *algébrique* parfaitement définie, etc.). La théorie présentée ici en est l'exemple le plus simple (mais aussi l'un des moins puissants).

On décrit ici des structures *typées* munies d'*opérations* travaillant sur ces structures. On trouve donc en premier lieu une notion de « sorte » (i.e. nom de type), puis une notion de nom d'opération muni d'une « arité » typée.

2.1 Les ensembles de sortes

Définition 1 : Un *ensemble de sortes* est un ensemble fini S. Un élément de S est appelé une *sorte*, c'est en fait un nom de type.

Une sorte est un objet syntaxique, tandis qu'un *type* est à la fois un ensemble de données et un ensemble de fonctionnalités travaillant sur ces données. Notons qu'à un même ensemble de données on peut associer plusieurs types différents selon les fonctionnalités considérées.

Exemple 2 : Si l'on envisage de spécifier des piles d'entiers naturels, on peut par exemple choisir $S = \{ Nat , Stack \}$.
Pour spécifier des ensembles finis d'entiers, avec test booléen d'appartenance, on peut par exemple choisir $S = \{ Bool , Nat , Set \}$, etc.

Pour chaque élément de syntaxe, on prendra toujours soin de définir les modèles qui lui sont associés :

Définition 3 : Un *S-ensemble*, ou encore *ensemble hétérogène au-dessus de S*, est un ensemble A muni d'une partition indexée par S :

$$A = \coprod_{s \in S} A_s$$

(où $\coprod$ désigne l'union disjointe)[1].

Exemple 4 : (Cf. ceux donnés en cours ...)

Variante : Certaines théories des TAA n'imposent pas que les A_s soient disjoints 2 à 2. On a alors une famille couvrante

$$A = \bigcup_{s \in S} A_s$$

Ceci revient à dire qu'une « donnée » peut posséder plusieurs types à la fois. De nombreux résultats de la suite de ce poly restent valides, mais pas tous...

Dans toute la suite, pour chaque classe de modèles nouvellement définie, on définira ce qu'est un *morphisme* entre deux tels modèles. Ce sont les morphismes qui premettent de « comparer » deux modèles entre eux. On s'empresse donc de définir ici les morphismes de S-ensembles :

Définition 5 : Etant donné un ensemble de sortes S, un *S-morphisme* entre deux S-ensembles A et B est une application μ de A vers B telle que :

$$(\forall s \in S) (\mu(A_s) \subseteq B_s)$$

Commentaires : contrairement aux fonctions, les applications sont toujours totalement définies ; chaque élément $a \in A$ possède une image $\mu(a)$ dans B. Un S-morphisme est simplement une application qui préserve les sortes (l'image d'une « donnée de type s » par μ est encore de type s). On n'impose nullement que μ soit surjective ou injective.

Exemple 6 : (Cf. ceux donnés en cours ...)

2.2 les signatures

Définition 7 : Une *arité* sur un ensemble de sortes S est un mot fini non vide sur S.

Définition 8 : Une *signature* est définie par un ensemble de sortes S et par un ensemble Σ de noms d'opération munis (chacun) d'une arité sur S. Les noms d'opération doivent être distincts 2 à 2.

Il n'est pas requis que l'ensemble Σ soit fini, bien qu'en pratique il le soit toujours (ne serait-ce que pour pouvoir écrire une spécification en un nombre fini de caractères).

[1] Ce n'est pas par hasard que $\coprod$ est symétrique du produit cartésien ($\prod$) ; nous verrons que ce sont des opérations duales dans la catégorie des ensembles.

Exemple 9 : Si l'on envisage de spécifier des piles d'entiers naturels, on peut par exemple choisir la signature constituée de $S = \{ Nat, Stack \}$ et de Σ contenant les noms d'opération et les arités suivants :

$0 :\rightarrow Nat$
$succ : Nat \rightarrow Nat$
$+ : NatNat \rightarrow Nat$
$EmptyStack :\rightarrow Stack$
$Push : NatStack \rightarrow Stack$
$Pop : Stack \rightarrow Stack$
$Top : Stack \rightarrow Nat$

Par convention, on place une flèche juste avant la dernière sorte de l'arité (qui existe puisque l'arité est un mot non vide). Intuitivement, les sortes à gauche de la flèche sont les types des arguments de l'opération, et l'unique sorte à droite de la flèche est le type du résultat retourné par l'opération. Lorsque l'arité d'une opération est réduite à une seule sorte (cf. 0 ou *EmptyStack*) il s'agit en fait d'une constante.
Le fait de n'avoir toujours qu'une seule sorte « cible » des opérations peut paraître limitatif : par exemple, dans la pratique, Pop et Top sont généralement « fondues » en une seule opération retournant la valeur du Top et effectuant le Pop par « effet de bord » sur la pile. Plusieurs sortes à droite de le flèche pourraient faciliter la spécification de telles opérations. En fait, on peut déclarer explicitement une sorte produit cartésien si nécessaire. De plus, et surtout, l'unicité du type cible permet de définir facilement la notion de *terme* (voir plus loin).

Une signature est donc en fait un couple (S, Σ). Par abus de notation, on notera souvent simplement Σ une signature.

Définition 10 : Etant donnée une signature Σ, une Σ-*algèbre* est un S-ensemble A muni, pour chaque nom d'opération de Σ

$$op : s_1\, s_2 \cdots s_n \rightarrow s$$

d'une application op_A (la sémantique de op dans A) de la forme :

$$op_A : A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n} \rightarrow A_s$$

(où « $\times$ » désigne le produit cartésien ensembliste usuel). Dans le cas où $n = 0$, op_A est simplement un élément de A_s.
On note $Alg(\Sigma)$ l'ensemble (ou plutôt la classe) des Σ-algèbres.

Par abus de notation, étant donnée une Σ-algèbre $A \in Alg(\Sigma)$, on note encore A le S-ensemble sous-jacent. Toutefois, lorsque cette ambiguïté peut être gênante, on le note plutôt $|A|$. Le S-ensemble $|A|$ est aussi appelé le *support* de A. Il faut remarquer qu'un même S-ensemble peut donner lieu à plusieurs Σ-algèbres différentes : il suffit de le munir de différentes sémantiques d'applications op_A (c'est ce qui motive la distinction entre « sorte » et « type »).

Exemple 11 : (Cf. ceux donnés en cours ...)

Définition 12 : Etant donnée une signature Σ, un Σ-morphisme entre deux Σ-algèbres A et B est un S-morphisme μ de A vers B tel que pour chaque nom d'opération de la signature

$$op : s_1 \cdots s_n \rightarrow s$$

et pour tout n-uplet $(a_1, \cdots, a_n) \in A_{s_1} \times \cdots \times A_{s_n}$ on a :

$$\mu(op_A(a_1, \cdots, a_n)) = op_B(\mu(a_1), \cdots, \mu(a_n))$$

(compatibilité de μ avec les opérations de la signature).
Dans le cas où $n = 0$, on impose simplement que $\mu(op_A) = op_B$.

Il existe bien sûr de nombreux S-morphismes qui ne sont pas des Σ-morphismes.
Exemple 13 : (Cf. ceux donnés en cours ...)

2.3 L'algèbre des termes fermés

L'ensemble des termes fermés est l'ensemble des « compositions licites d'opérations » que l'on peut faire à partir de la signature. Par exemple on peut considérer, sur la signature des piles d'entiers naturel, les termes :

$$(0 + 0)$$
$$(Top(Push(0, Push(succ(0), Push(succ(0), EmptyStack)))) + succ(0))$$
$$Pop(Pop(Push(0, EmptyStack))).$$

Par contre des compositions telles que $Pop(0)$ ou $succ(EmptyStack + 0)$ induisent des « erreurs de typage » et ne sont pas des termes licites.

Définition 14 : On définit ci-dessous l'algèbre des *termes fermés* sur une signature Σ, notée T_Σ.
Sa structure de S-ensemble est définie comme suit : T_Σ est le plus petit S-ensemble (au sens de l'inclusion) tel que :
- chaque constante ($cste :\rightarrow s$) de Σ est un terme fermé de sorte s, c'est-à-dire $cste \in (T_\Sigma)_s$
- pour toute opération non constante de Σ, $op : s_1 \cdots s_n \rightarrow s$, et pour tout n-uplet de termes fermés de sortes compatibles

$$(t_1, \cdots, t_n) \in (T_\Sigma)_{s_1} \times \cdots \times (T_\Sigma)_{s_n}$$

$op(t_1, \cdots, t_n)$ est encore un terme fermé, de sorte s (élément de $(T_\Sigma)_s$)
- deux termes fermés qui s'écrivent différemment sont différents dans T_Σ.

On munit T_Σ d'une structure de Σ algèbre très simplement : pour toute opération $op : s_1 \cdots s_n \rightarrow s$, sa sémantique op_{T_Σ} est l'application de $(T_\Sigma)_{s_1} \times \cdots \times (T_\Sigma)_{s_n}$ dans $(T_\Sigma)_s$ qui à $(t_1, \cdots, t_n)$ associe le terme fermé $op(t_1, \cdots, t_n)$.

Remarquons que, par exemple sur la signature des entiers naturels, les termes $succ(0)$, $succ(0+0)$, $(0+succ(0))$, etc. sont tous distincts puisque deux termes fermés qui s'écrivent différemment sont différents dans T_Σ. Il ne faut pas confondre un *terme* et sa « valeur »calculée dans une autre algèbre (dans $\mathbb{N}$ tous ces termes donnent le même résultat : 1).

Parmi toutes les Σ-algèbres, T_Σ joue un role particulier car il possède la propriété suivante, dite « d'initialité » :

Théorème 15 : Pour toute Σ-algèbre A, il existe un unique Σ-morphisme de T_Σ dans A.

Preuve : Commençons par démontrer l'unicité ; nous allons démontrer la propriété suivante :
Pour tout terme fermé t il existe un élément a de A tel que tout morphisme μ de T_Σ dans A vérifie $\mu(t) = a$. C'est-à-dire :

$$(\forall\, t \in T_\Sigma\,) \, (\exists\, a \in A\,) \, (\forall\, \mu :\, T_\Sigma \to A\,) \, (\, \mu(t) = a\,)$$

Nous faisons une récurrence sur le nombre d'occurrences d'opérations contenues dans le terme t.
Le cas de base : si t est réduit à une opération (donc une constante $cste$), alors on choisit $a = cste_A$. Puisque tous les Σ-morphismes μ sont compatibles avec les opérations de Σ on a nécessairement $\mu(cste) = cste_A$.
L'étape de récurrence : on considère un terme t constitué de n opérations (avec $n \geq 2$) et on suppose que tout terme contenant au plus $n-1$ opérations vérifie cette propriété. Puisque t contient au moins 2 opérations, il est nécessairement de la forme $t = op(t_1, \cdots, t_m)$. Donc pour tout Σ-morphisme μ, on a $\mu(t) = op_A(\mu(t_1), \cdots, \mu(t_m))$; ceci à cause de la compatibilité des morphismes avec les opérations de Σ. Or tous les termes t_i contiennent strictement moins d'opérations que t (ils ne contiennent pas l'opération de tête de t : op), donc il existe $a_1 \cdots a_m$ tels que tout Σ-morphisme μ vérifie $\mu(t_i) = a_i$. Il en résulte que pour tout μ on a $\mu(t) = op_A(a_1, \cdots, a_m)$. Il suffit donc de choisir $a = op_A(a_1, \cdots, a_m)$.

Pour démontrer l'existence, on remarque tout d'abord que les conditions dégagées précédemment définissent complètement une application de T_Σ dans A : $\mu(cste) = cste_A$ et $\mu(op(t_1, \cdots, t_m)) = op_A(\mu(t_1), \cdots, \mu(t_m))$. En effet, ceci définit μ récursivement sur tous les termes fermés (cf. la définition de T_Σ). Cette application est trivialement compatible avec les sortes et avec les opérations ; c'est donc un Σ-morphisme. Ceci clos la preuve. □

Intuitivement, ce Σ-morphisme dont l'existence et l'unicité est affirmée par le théorème précédent est justement le morphisme qui « calcule » (qui « evalue ») un terme de T_Σ au sein d'une algèbre (A). C'est lui qui relie une « composition licite d'opérations » à sa valeur calculée dans une algèbre A. Pour cette raison on le note souvent $eval_A :\, T_\Sigma \to A$.

2.4 Les algèbres finiment engendrées

Etant donnée une Σ-algèbre A quelconque, le morphisme $eval_A$ n'est pas necessairement surjectif. S'il ne l'est pas, ceci signifie qu'il existe des « valeurs » a de A qui ne sont jamais « résultat d'un calcul » issu des opérations de la signature. Ces valeurs « supplémentaires » sont donc difficiles à manipuler puisqu'il n'existe aucune écriture syntaxique pour les exprimer. En particulier on pourra démontrer beaucoup moins de propriétés les concernant. On comprend bien dès lors que les algèbres qui ne contiennent pas de telles valeurs ont un rôle particulier à jouer puisqu'on peut mieux les décrire. C'est pourquoi on pose la définition suivante.

Définition 16 : Une Σ-algèbre A est dite *finiment engendrée* (ou encore « finiment générée ») si le Σ-morphisme $eval_A$ est surjectif. On note $Gen(\Sigma)$ la classe des Σ-algèbres finiment engendrées.

On a donc : $Gen(\Sigma) \subseteq Alg(\Sigma)$.

Exemple 17 : L'algèbre des entiers naturels $\mathbb{N}$ est finiment engendrée sur la signature de NAT (celle avec les opérations 0 *succ* et +). L'ensemble des entiers relatifs $\mathbb{Z}$ est muni de manière naturelle d'une structure de Σ-algèbre sur cette même signature, mais elle n'est pas finiment engendrée car les entiers négatifs ne peuvent être atteints par aucune composition des opérations 0 *succ* et + .

Notons que T_Σ est lui-même finiment engendré car l'identité sur T_Σ est un Σ-morphisme surjectif.
Lorsque l'on veut démontrer un résultat concernant les algèbres de $Gen(\Sigma)$, on peut utiliser un raisonnement par *induction* similaire à celui développé dans la preuve d'initialité (le théorème d'existence et unicité de $eval_A$). Il n'y a en général aucune raison pour qu'un résultat démontré par cette méthode sur $Gen(\Sigma)$ puisse s'étendre à $Alg(\Sigma)$.

2.5 L'algèbre triviale

Définition 18 : L'*algèbre triviale*, notée $Triv_\Sigma$ ou simplement $Triv$ si la signature considérée est clairement établie par le contexte, et définie comme suit :
- le S-ensemble $Triv$ est défini par $Triv_s = \{s\}$ pour toute sorte $s \in S$ (ceci revient à dire que $Triv = S$ en tant qu'ensemble, et est muni de sa structure canonique de S-ensemble)
- pour chaque opération $op : s_1 \cdots s_n \to s$ de Σ, sa sémantique est l'application op_{Triv} de $Triv_{s_1} \times \cdots \times Triv_{s_n} = \{s_1\} \times \cdots \times \{s_n\}$ dans $Triv_s = \{s\}$ qui à l'unique élément $(s_1, \cdots, s_n)$ associe s

Remarquons que $Triv$ possède la particularité suivante : une fois sa structure de S-ensemble établie (par le premier point de la définition), il n'existe qu'une unique façon d'en faire une Σ-algèbre (c'est-à-dire que l'on ne dispose d'aucun choix pour la sémantique de chaque opération *op* de Σ). Ceci est dû au fait que chaque $Triv_s$ ne contient qu'un seul élément, c'est donc nécessairement lui le résultat sémantique de toute opération dont la cible est de sorte s.
Cette remarque peut s'étendre aux Σ-morphismes, fournissant une propriété similaire à la propriété d'initialité concernant T_Σ. Il s'agit en fait de la propriété dite *duale*.

Théorème 19 : Pour toute Σ-algèbre A (de $Alg(\Sigma)$) il existe un unique Σ-morphisme de A dans $Triv$.

Preuve : C'est l'application qui à tout élément a de A associe sa sorte s. On vérifie trivialement que c'est un S-morphisme, et un Σ-morphisme (i.e. compatible avec les opérations). L'unicité résulte encore du fait que pour toute sorte s, $Triv_s$ ne contient que s pour élément. □

Exercice : Considérant la variante qui définit les Σ-algèbres via une famille couvrante (i.e. $A = \bigcup_{s \in S} A_s$) au lieu d'une partition (i.e. $A = \coprod_{s \in S} A_s$), comment définir l'algèbre $Triv_\Sigma$ pour que le théorème précédent soit encore vrai ?

Définition 20 : Une signature Σ est dite *raisonnable* (« sensible » en anglais) si $Triv_\Sigma$ est finiment engendrée.

Commentaire : puisque nous avons muni $Triv$ d'une structure de Σ-algèbre, on a $Triv \in Alg(\Sigma)$, mais rien ne prouve a priori que $Triv \in Gen(\Sigma)$. Examinons de plus près le cas pathologique où $Triv$ n'est pas finiment engendrée. Ceci signifie qu'il existe un élément de $Triv$, c'est-à-dire une sorte s, telle qu'aucun terme fermé n'est de cette sorte. Autrement dit, il existe des « types qui ne sont jamais atteints » par une composition licite d'opérations.

Exemple 21 : La signature suivante n'est pas raisonnable car il n'existe aucun terme fermé de sorte *Stack* :
- $S = \{ Nat , Stack \}$
- $\Sigma = \{0 , succ , +, Push , Pop , Top\}$ avec les arités déjà données pour les piles (ici, on a simplement omis la constante $EmptyStack$)

Clairement, pour qu'une signature soit raisonnable, il *suffit* qu'elle possède une constante dans chaque sorte. Mais la réciproque est fausse :

Exemple 22 : La signature suivante est raisonnable, bien que sans constante de sorte PC :
- $S = \{ Nat , PC \}$ (PC pour *Produit Cartésien*)
- Σ contenant les opérations 0 *succ* et + habituelles et l'operation

$$\_ \cdot \_ \ : \ Nat\ Nat \rightarrow PC$$

(on constate que le terme $(0 \cdot 0)$ est de sorte PC).

Exercice : Ainsi, alors que T_Σ est toujours finiment engendré, ce n'est pas toujours le cas de $Triv_\Sigma$. Il en résulte que $Gen(\Sigma)$ possède toujours une algèbre ayant la propriété d'initialité (T_Σ elle même), et l'on peut se demander ce qui se passe concernant la propriété duale :
Dans le cas d'une signature non raisonnable trouver une algèbre *finiment engendrée* $TrvGen$ telle que pour toute Σ-algèbre *finiment engendrée* A, il existe un unique Σ-morphisme de A dans $TrvGen$.

Dans toute la suite de ce cours, sauf mention du contraire (en particulier pour la paramétrisation), on ne considèrera plus que des signatures raisonnables.

3 Equations et axiomes

Les équations ou axiomes servent à énoncer des propriétés liant les opérations de la signature. Par exemple que $0+0 = 0$. Enoncer ces propriétés exhaustivement sur toutes les compositions licites d'opérations (i.e. sur T_Σ) conduirait à des axiomes trop peu puissants. On énonce des propriétés plus générales grâce à des « variables ».

3.1 L'algèbre des termes avec variables

Définition 23 : Etant donnée une signature Σ, un *ensemble de variables* est un S-ensemble X ne contenant aucun des noms d'opération de Σ. Les éléments de X sont appelés des *variables*.

Remarquons que chaque variable est donc « typée » (i.e. possède une et une seule sorte). Dans ce qui suit, on considère une signature Σ et un ensemble de variables X.

Définition 24 : La Σ-algèbre des *termes avec variables*, notée $T_\Sigma(X)$, est définie ci-dessous.
Sa structure de S-ensemble est définie comme suit : $T_\Sigma(X)$ est le plus petit S-ensemble (au sens de l'inclusion) tel que :
- chaque variable x de sorte s est un terme avec variables de sorte s, c'est-à-dire $x \in (T_\Sigma(X))_s$
- chaque constante ($cste :\rightarrow s$) de Σ est un terme avec variables de sorte s, c'est-à-dire $cste \in (T_\Sigma(X))_s$
- pour toute opération non constante de Σ, $op : s_1 \cdots s_n \rightarrow s$, et pour tout n-uplet de termes avec variables de sortes compatibles

$$(t_1, \cdots, t_n) \in (T_\Sigma(X))_{s_1} \times \cdots \times (T_\Sigma(X))_{s_n}$$

$op(t_1, \cdots, t_n)$ est encore un terme avec variables, de sorte s (élément de $(T_\Sigma(X))_s$)
- deux termes avec variables qui s'écrivent différemment sont différents dans $T_\Sigma(X)$.

On munit $T_\Sigma(X)$ d'une structure de Σ algèbre très simplement : pour toute opération $op : s_1 \cdots s_n \rightarrow s$, sa sémantique $op_{T_\Sigma(X)}$ est l'application de $(T_\Sigma(X))_{s_1} \times \cdots \times (T_\Sigma(X))_{s_n}$ dans $(T_\Sigma(X))_s$ qui à $(t_1, \cdots, t_n)$ associe le terme avec variables $op(t_1, \cdots, t_n)$.

On peut définir plus rapidement $T_\Sigma(X)$ comme étant l'algèbre des termes fermés sur la signature $\Sigma \cup X$ (même si cette nouvelle signature n'est pas finie), munie de la structure de Σ-algèbre obtenue en ne considérant que les sémantiques des opérations de Σ (puisque $\Sigma \subseteq \Sigma \cup X$). Ceci revient à *oublier*, dans $T_{\Sigma \cup X}$, les sémantiques associées aux opérations constantes que sont les éléments de X pour la signature $\Sigma \cup X$.

Exemple 25 : (cf. les exemples de termes avec variables donnés en cours).

Remarquons que $T_\Sigma(X)$ contient T_Σ de manière canonique (les termes « avec variables » qui ne contiennent aucune variable dans leur expression sont en fait des termes fermés). Cette inclusion est clairement un Σ-morphisme (c'est donc le morphisme $eval_{T_\Sigma(X)}$ par unicité de ce morphisme).
Remarquons de même que $T_\Sigma(X)$ contient X de manière canonique (puisque chaque variable est un terme avec variable de même sorte). Cette inclusion est un S-morphisme de X dans $T_\Sigma(X)$.
Notons enfin que $T_\Sigma = T_\Sigma(\emptyset)$.

3.2 Les équations

Définition 26 : Une *équation* est une paire de termes avec variables de même sorte. On la note : $t = t'$ (où t et t' sont les termes avec variables de cette paire).

Rappelons qu'une « paire » se différencie d'un couple en ce que ses composantes ne sont pas ordonnées : la paire $t = t'$ est égale à la paire $t' = t$.

Exemple 27 : On peut par exemple énoncer les propriétés des entiers naturels et piles d'entiers naturels, sur la signature des piles déjà donnée en exemple, par :

$n + 0 = n$
$n + succ(m) = succ(n + m)$
$Pop(EmptyStack) = EmptyStack$

$Pop(Push(n, X)) = X$
$Top(EmptyStack) = 0$
$Top(Push(n, X)) = n$

(ici les variables de sorte Nat sont les lettres minuscules, et celles de sorte $Stack$ sont les lettres majuscules).
Naturellement, on constate que les troisième et cinquième équations ne sont que des conventions. Il existe des théories plus puissantes des TAA autorisant la spécification de traitement d'exceptions, avec messages d'erreurs et autres traitements spécifiques. Dans la théorie très simple que nous exposons ici, nous sommes cependant contraints à de telles conventions peu réalistes.

Dans toutes les théories de TAA utilisables en pratique, on définit rigoureusement une notion de « formule bien formée » (ou plus simplement « formule »). Ici nous avons défini les équations. Il faut ensuite définir leur sémantique, c'est-à-dire caractériser parmi les Σ-algèbres celles qui *valident* (i.e. satisfont) une formule donnée. Dans le cas des équations, la notion de validation passe par celle de *substitution*, qui utilise elle même la notion d'*interprétation*.

Définition 28 : Etant donnée une Σ-algèbre A, une *interprétation* est un S-morphisme I de l'ensemble de variables X dans A.

Ceci revient simplement à associer à chaque variable une « valeur » dans l'algèbre A, de même sorte que la variable.

Théorème 29 : Etant donnée une interprétation I de X dans A, il existe un unique Σ-morphisme σ de $T_\Sigma(X)$ dans A qui prolonge I.

Preuve : Laissée en exercice. La notion de « prolongement » est prise ici au sens de l'inclusion de X dans $T_\Sigma(X)$ mentionnée plus haut ; ceci signifie simplement que pour tout terme réduit à une variable x, on requiert que $\sigma(x) = I(x)$. En utilisant cette contrainte de prolongement, la démonstration du théorème est exactement calquée sur celle du théorème d'initialité de T_Σ. □

Remarquons que le théorème précédent est une généralisation du théorème d'initialité concernant T_Σ : le théorème 15 est un cas particulier du théorème 29 (dans le cas où $X = \emptyset$).
Ce que dit le théorème 29, c'est simplement qu'une fois que l'on a fixé la valeur de chaque variable dans A, chaque terme avec variable peut « se calculer » (s'évaluer) en une et une seule valeur dans A.

Définition 30 : Le morphisme σ défini par le théorème 29 précédent est appelé une *substitution à valeur dans A*.

Définition 31 : Soient $t = t'$ une équation et A une Σ-algèbre. L'algèbre A *valide* $t = t'$ si pour toute substitution σ à valeur dans A on a : $\sigma(t) = \sigma(t')$ (dans A).

Ceci signifie simplement que l'équation doit être satisfaite dans A quelles que soient les valeurs données aux variables. On constate donc que toutes les variables apparaissant dans une équation sont implicitement quantifiées universellement.

Remarque 32 : L'algèbre T_Σ ne valide aucune équation non triviale (une équation triviale est une équation de la forme $t = t$). Par contre $Triv$ valide toutes les équations.

3.3 Les équations conditionnelles positives

Définition 33 : Une *équation conditionnelle positive* (ou parfois *axiome conditionnel* ou simplement *axiome* par abus de langage) est une formule de la forme :

$$(t_1 = t'_1) \ \wedge \cdots \wedge \ (t_n = t'_n) \Longrightarrow t_{n+1} = t'_{n+1}$$

où les $(t_i = t'_i)$ sont des équations.

Une équation est en particulier une équation conditionnelle positive (cas où $n = 0$).

Exemple 34 : On peut par exemple spécifier les ensembles finis d'entiers naturels avec l'ensemble de sorte suivant

$S = \{ \, Bool \, , \, Nat \, , \, Set \, \}$

l'ensemble d'opérations Σ contenant les opérations suivantes :

$True :\rightarrow Bool$
$False :\rightarrow Bool$
$or : BoolBool \rightarrow Bool$
$0 :\rightarrow Nat$
$succ : Nat \rightarrow Nat$
$eq? : NatNat \rightarrow Bool$
$\emptyset :\rightarrow Set$
$insert : NatSet \rightarrow Set$
$\in : NatSet \rightarrow Bool$
$delete : NatSet \rightarrow Set$

et les axiomes suivants :

$b \ or \ True = True$
$b \ or \ False = b$
$eq?(0, 0) = True$
$eq?(0, succ(n)) = False$
$eq?(succ(n), 0) = False$
$eq?(succ(n), succ(m)) = eq?(n, m)$
$insert(x, insert(y, X)) = insert(y, insert(x, X))$
$insert(x, insert(x, X)) = insert(x, X)$
$x \in \emptyset = False$
$x \in insert(y, X) = eq?(x, y) \ or \ x \in X$
$delete(x, \emptyset) = \emptyset$
$eq?(x, y) = False \Longrightarrow delete(x, insert(y, X)) = insert(y, delete(x, X))$
$eq?(x, y) = True \Longrightarrow delete(x, insert(y, X)) = delete(x, X)$

Définition 35 : Une Σ-algèbre A *valide* un axiome de la forme

$$(t_1 = t'_1) \ \wedge \cdots \wedge \ (t_n = t'_n) \Longrightarrow t = t'$$

si et seulement si pour toute substitution σ à valeur dans A on a : *si* $\sigma(t_i) = \sigma(t'_i)$ pour tout $i = 1..n$ *alors* $\sigma(t) = \sigma(t')$ (dans A).

Exercice : L'énoncé suivant est-il correct ?
A valide $[(t_1 = t'_1) \wedge \cdots \wedge (t_n = t'_n) \Longrightarrow t = t']$ si et seulement si : *si* A valide chacune des équations $t_i = t'_i$ *alors* A valide $t = t'$.

Exercice : Que peut-on dire de T_Σ et de $Triv$ concernant la validation d'une équation conditionnelle positive ?

3.4 Présentation d'un types abstraits algébriques

Nous sommes maintenant à même de définir ce qu'est une présentation dans le cadre de la théorie « classique » des TAA (celle définie par ADJ).

Définition 36 : Une *présentation d'un TAA* (ou plus simplement une *spécification*) est un triplet $PRES = (S, \Sigma, Ax)$ tel que (S, Σ) est une signature et Ax est un ensemble d'équations conditionnelles positives sur cette signature.

Ax peut éventuellement être infini, bien qu'en pratique on n'accepte qu'un nombre fini d'axiomes (pour pouvoir écrire une spécification en un nombre fini de caractères !).

Définition 37 : Soit $PRES = (S, \Sigma, Ax)$ une présentation. Une Σ-algèbre A *valide* $PRES$ signifie que A valide tous les axiomes de Ax.
On note $Alg(PRES)$ la sous-classe de $Alg(\Sigma)$ contenant les Σ-algèbres qui valident $PRES$. On note $Gen(PRES)$ la sous-classe des Σ-algèbres finiment engendrées qui valident $PRES$.

Chapitre B :
Les congruences

1 Définitions

Rappelons qu'une *relation* (binaire) $\mathcal{R}$ sur un ensemble A est caractérisée par l'ensemble des couples (x, y) d'éléments de A tels que x est en relation avec y pour $\mathcal{R}$, noté $x\mathcal{R}y$. Une relation binaire est donc caractérisée par son *graphe* qui n'est autre qu'un sous-ensemble de $A^2 = A \times A$. Dans toute la suite, une relation $\mathcal{R}$ dénotera sans distinction la relation $\mathcal{R}$ elle même ou son graphe. En particulier l'ensemble des relations sur A est muni d'un ordre partiel défini par l'inclusion des graphes. Dire que $\mathcal{R}$ est *plus petite* que $\mathcal{R}'$ signifie donc simplement :

$$(\forall (x, y) \in A^2) (x\mathcal{R}y \Longrightarrow x\mathcal{R}'y)$$

Définition 1 : Soient Σ une signature et A une Σ-algèbre. Une *congruence* sur A est une relation d'équivalence « $\equiv$ » sur A (i.e. réflexive, symétrique et transitive) telle que :

***compatibilité avec les sortes* :** deux éléments de A ne peuvent être en relation que s'ils sont de même sorte, c'est-à-dire

$$(\forall (x, y) \in A^2) (x \equiv y \Longrightarrow (\exists\, s \in S)\, (x \in A_s \text{ et } y \in A_s))$$

ou encore : $\equiv \subseteq (\coprod_{s \in S} A_s)$

***compatibilité avec les opérations* :** pour toute opération de la signature op : $s_1 \cdots s_n \to s$, et pour tout n-uplets d'éléments de A $(a_1, \cdots, a_n)$ et $(b_1, \cdots, b_n)$ de sortes compatibles on a : *si* $a_1 \equiv b_1$ *et ... et* $a_n \equiv b_n$ *alors* $op_A(a_1, \cdots, a_n) \equiv op_A(b_1, \cdots, b_n)$

Rappelons qu'étant donnée une relation d'équivalence $\mathcal{R}$ sur un ensemble A, on peut considérer l'ensemble quotient $A/_{\mathcal{R}}$ dont les éléments sont les classes d'équivalence de $\mathcal{R}$ sur A. On note $\overline{a}$ la classe d'équivalence d'un élément a de A.

Proposition 2 : Soit A une Σ-algèbre. Si $\equiv$ est une congruence sur A, alors l'ensemble quotient $A/_{\equiv}$ est canoniquement muni d'une structure de Σ-algèbre.

Preuve : On munit d'abord $A/_{\equiv}$ d'une structure de S-ensemble en associant à chaque classe d'équivalence la sorte d'un de ses éléments. Ceci a un sens car d'une part, par définition des ensemble quotient, aucune classe d'équivalence n'est vide, et d'autre part tous les éléments d'une même classe d'équivalence on la même sorte à cause du premier point de la définition précédente. La sorte d'un élément de $A/_{\equiv}$ (i.e. d'une classe d'équivalence) ne dépend donc pas du représentant choisi dans cette classe.
On munit ensuite $A/_{\equiv}$ d'une structure de Σ-algèbre comme suit. Pour chaque opération constante $cste$ de Σ, on pose $cste_{A/_{\equiv}} = \overline{cste_A}$. Pour chaque opération non constante $op : s_1 \cdots s_n \to s$ de Σ, et pour chaque n-uplet de classes d'équivalence $(\overline{a_1}, \cdots, \overline{a_n})$ de sortes compatibles, on pose :

$$op_{A/_{\equiv}}(\overline{a_1}, \cdots, \overline{a_n}) = \overline{op_A(a_1, \cdots, a_n)}$$

Ceci a un sens car $\overline{op_A(a_1, \cdots, a_n)}$ ne dépend pas des représentants $a_1 \cdots a_n$ choisis, à cause du second point de la définition. □

Définition 3 : Par abus de notation, on notera toujours $A/_{\equiv}$ la Σ-algèbre canonique dont le support est l'ensemble quotient $A/_{\equiv}$. C'est l'*algèbre quotient* de A par la congruence $\equiv$.

Proposition 4 : Avec les notations précédentes, l'application quotient de A sur $A/_{\equiv}$ (qui à tout élément a de A associe sa classe $\overline{a}$) est un Σ-morphisme.

Preuve : En exercice, il suffit de se référer aux définitions. □

Exercice : Soit $\mu : A \to B$ un Σ-morphisme. Soit $\equiv_\mu$ la relation binaire sur A définie par $(a \equiv_\mu a') \Longleftrightarrow \mu(a) = \mu(a')$. Démontrer que $\equiv_\mu$ est une congruence sur A.

2 Résultats fondamentaux

Les résultats énoncés ici sont au centre de la théorie des TAA suivant l'approche « ADJ ». Ils sont en fait cruciaux pour toute théorie des TAA qui veut dégager une sémantique plus ou moins monomorphique (c'est-à-dire une sémantique qui privilégie une unique classe d'isomorphisme d'algèbres associée à une syntaxe donnée).

Théorème 5 : Soit A une Σ-algèbre. Soit $\mathcal{R}$ une relation binaire sur A compatible avec les sortes (i.e. $\mathcal{R} \subseteq (\coprod_{s \in S} A_s)$).

Il existe une plus petite congruence sur A contenant $\mathcal{R}$. Elle est notée $\equiv_{\mathcal{R}}$.

Preuve : Soit $\mathcal{F}$ l'ensemble des toutes les congruences sur A qui contiennent $\mathcal{R}$. Cet ensemble est non vide car il contient la congruence $\equiv_{Triv}$ définie par

$$(a \equiv_{Triv} b) \Longleftrightarrow (a \textit{ et } b \textit{ de même sorte}) \ .$$

Soit $\equiv_{\mathcal{R}}$ la relation binaire sur A définie par $\equiv_{\mathcal{R}} = (\bigcap_{\equiv \in \mathcal{F}} \equiv)$.

Si $\equiv_{\mathcal{R}}$ est une congruence, alors elle répond clairement au théorème. Prouvons donc que c'est une congruence ; notons que par définition, pour tous éléments a et b de A :

$$a \equiv_{\mathcal{R}} b \Longleftrightarrow (\ \forall\ \equiv \in \mathcal{F}\ )\ (\ a \equiv b\ ) \ .$$

- puisque toutes les congruences $\equiv$ de $\mathcal{F}$ sont réflexives, symétriques et transitives, il en est de même de $\equiv_{\mathcal{R}}$ (immédiat)
- puisque toutes les congruences de $\mathcal{F}$ sont compatibles avec les sortes (en particulier $\equiv_{Triv}$), $\equiv_{\mathcal{R}}$ est aussi compatible avec les sortes
- soit $op : s_1 \cdots s_n \to s$ une opération quelconque de la signature et supposons que $a_1 \equiv_{\mathcal{R}} b_1$ et ... et $a_n \equiv_{\mathcal{R}} b_n$ (de sortes compatibles). Il en résulte que, par définition de $\equiv_{\mathcal{R}}$, on a $a_1 \equiv b_1$ et ... et $a_n \equiv b_n$ pour toute congruence $\equiv$ de $\mathcal{F}$. Par conséquent, les $\equiv$ étant toutes compatibles avec op, on a $op_A(a_1, \cdots, a_n) \equiv op_A(b_1, \cdots, b_n)$ pour toute congruence $\equiv$ de $\mathcal{F}$. D'où la compatibilité de $\equiv_{\mathcal{R}}$ avec op : $op_A(a_1, \cdots, a_n) \equiv_{\mathcal{R}} op_A(b_1, \cdots, b_n)$.

□

Définition 6 : Avec les notations du théorème précédent, on dit que $\equiv_{\mathcal{R}}$ est la congruence *engendrée par* $\mathcal{R}$ *sur* A.

Notation 7 : Soit $PRES = (S, \Sigma, Ax)$ une présentation. Soit A une Σ-algèbre. On note $\mathcal{R}_{Ax}$ la relation binaire sur A telle que $a\mathcal{R}_{Ax}b$ si et seulement si :
il existe un axiome de Ax

$$(t_1 = t'_1) \ \wedge \cdots \wedge \ (t_n = t'_n) \Longrightarrow t = t'$$

et il existe une substitution σ à valeur dans A telle que $\sigma(t_i) = \sigma(t'_i)$ pour tout $i = 1..n$ et $\sigma(t) = a$ et $\sigma(t') = b$.

Commentaire : $\mathcal{R}_{Ax}$ est simplement l'ensemble des *égalités qui manquent* à A pour qu'elle valide les axiomes de Ax. On remarque en particulier que si A valide déjà $PRES$ alors $\mathcal{R}_{Ax}$ est incluse dans la diagonale de A^2 (la « diagonale »est l'ensemble des couples de la forme (a, a)).
Remarquons de plus que, puisque les égalités sont des paires de termes de même sorte, $\mathcal{R}_{Ax}$ est compatible avec les sortes. On peut donc lui appliquer le théorème fondamental précédent et ainsi poser la définition suivante.

Définition 8 : Avec les notations précédentes, la congruence engendrée par $\mathcal{R}_{Ax}$ est notée $\equiv_{Ax}$, et est appelée la congruence engendrée par Ax sur A.
De même la Σ-algèbre quotient $A/_{\equiv_{Ax}}$ est plus simplement notée $A/_{Ax}$, et est appelé l'algèbre quotient de A par Ax.
On notera $quotient_{A,Ax}$ le Σ-morphisme quotient de A sur $A/_{Ax}$.

Proposition 9 : Avec les notations précédentes, pour toute Σ-algèbre A, $A/_{Ax}$ est une $PRES$-algèbre (i.e. elle valide Ax). De plus, si A est finiment engendrée alors il en est de même de $A/_{Ax}$.

Preuve : En exercice. □

On remarque facilement que si A valide déjà Ax, alors $A/_{Ax}$ est isomorphe à A via $quotient_{A,Ax}$ (les classes d'équivalences de $\equiv_{Ax}$ sont réduites à des singletons).

3 Minimalité et Initialité

Nous disposons maintenant de deux façons, que l'on peut qualifier de « duales », pour lier $Alg(\Sigma)$ et $Gen(\Sigma)$ avec $Alg(PRES)$ et $Gen(PRES)$. Nous avons tout d'abord défini $Alg(PRES)$ (resp. $Gen(PRES)$) comme une *sous-classe* de $Alg(\Sigma)$ (resp. $Gen(\Sigma)$). Nous avons maintenant, grâce aux congruences, un moyen de « forcer » toute Σ-algèbre A à valider $PRES$ en la quotientant par $\equiv_{Ax}$. Bien sûr, l'application de $Alg(\Sigma)$ dans $Alg(PRES)$ qui fait correspondre $A/_{Ax}$ à toute Σ-algèbre A est surjective, tandis que l'inclusion de $Alg(PRES)$ dans $Alg(\Sigma)$ est injective. Intuitivement, pour les mordus de géométrie élémentaire, la « dualité » *inclusion* vs. *quotient* entre $Alg(\Sigma)$ et $Alg(PRES)$ peut être comparée avec l'*inclusion* d'une droite dans le plan vs. la *projection* (par exemple orthogonale) du plan sur cette droite.

Nous allons maintenant étudier d'un peu plus près la vision « quotient ».

Proposition 10 : Soient A une Σ-algèbre et B une $PRES$-algèbre. S'il existe un morphisme μ de A dans B, alors il existe un unique morphisme $\overline{\mu}$ de $A/_{Ax}$ dans B tel que $\mu = \overline{\mu} \circ quotient_{A,Ax}$.

Preuve : Tout d'abord l'unicité est immédiate car $quotient_{A,Ax}$ est surjectif : pour toute classe d'équivalence $\overline{a}$ de $A/_{Ax}$, on doit nécessairement avoir $\overline{\mu}(\overline{a}) = \mu(a)$. Ensuite, pour que $\overline{\mu}$ existe, il suffit que $\overline{\mu}(\overline{a})$ ne dépende pas du représentant a de $\overline{a}$.
Considérons la congruence $\equiv_\mu$ définie sur A par $(a \equiv_\mu a') \Longleftrightarrow \mu(a) = \mu(a')$. Il suffit de démontrer que $\equiv_\mu$ contient $\equiv_{Ax}$. Mais puisque B valide Ax, $\equiv_\mu$ contient la relation $\mathcal{R}_{Ax}$ (exercice : s'en convaincre), c'est-à-dire que $\equiv_\mu$ est élément de l'ensemble $\mathcal{F}$ décrit dans la preuve du théorème fondamental de la section précédente. Puisque $\equiv_{Ax}$ est la plus petite congruence de $\mathcal{F}$, elle est en particulier incluse dans $\equiv_\mu$. □

Ce qu'affirme le résultat précédent, c'est simplement que $quotient_{A,Ax}$ est le morphisme minimal qui quotiente A en une $PRES$-algèbre. Par conséquent, si B est un autre quotient de A (via μ) qui valide $PRES$ alors on peut factoriser ce quotient « plus grand » en quotientant d'abord de manière minimale (i.e. en construisant $A/_{Ax}$), puis en effectuant les quotients « qui manquent » pour atteindre B (via $\overline{\mu}$). On comprend alors que cette proposition est simplement une autre manière de dire que $\equiv_{Ax}$ est une congruence minimale ; ici on a *traduit* cette propriété en terme de Σ-morphismes. On comprendra mieux l'intérêt de telles traductions par un usage systématique de la théorie des catégories.

Appliquons ce quotient minimal à l'algèbre T_Σ. On peut prévoir que ce quotient est d'un intérêt majeur car nous avons vu que T_Σ est déjà minimale parmi les Σ-algèbres (propriéte d'initialité) ; on conçoit bien que son quotient sera quant-à lui minimal parmi les $PRES$-algèbres.

Définition 11 : Soit $PRES = (S, \Sigma, Ax)$ une présentation. Le quotient de T_Σ par la congruence engendrée par Ax est noté T_{PRES}.

C'est donc une Σ-algèbre qui valide Ax : $T_{PRES} \in Alg(PRES)$. Mieux : puisque $quotient_{T_\Sigma,Ax}$ est surjectif, on a $T_{PRES} \in Gen(PRES)$.

Théorème 12 : Pour toute $PRES$-algèbre B, il existe un unique Σ-morphisme de T_{PRES} dans B.

Preuve : *Existence* : on applique la proposition précédente avec $A = T_\Sigma$ et μ étant l'unique morphisme de T_Σ dans B (propriété d'initialité de T_Σ) ; le morphisme $\overline{\mu}$ de $T_{PRES} = A/_{Ax}$ dans B convient.
Unicité : supposons qu'il existe 2 morphismes distincts α et β de T_{PRES} dans B. Puisque $quotient_{T_\Sigma,Ax}$ est surjectif, $\alpha \circ quotient_{T_\Sigma,Ax}$ et $\beta \circ quotient_{T_\Sigma,Ax}$ seraient 2 morphismes distincts de T_Σ dans B ; ce qui contredit la propriété d'initialité de T_Σ. □

Par conséquent, de même que T_Σ est initial vis-à-vis de $Alg(\Sigma)$ et $Gen(\Sigma)$, T_{PRES} est initial vis-à-vis de $Alg(PRES)$ et $Gen(PRES)$. Pour la théorie (monomorphique) la plus simple des TAA, c'est T_{PRES} qui est l'algèbre privilégiée représentant la sémantique de la présentation $PRES$.

Chapitre C : Exemples de spécifications équationnelles

Source: thèse de troisième cycle, G. Bernot, Université de Paris-Sud, février 1986.

1. Les booléens

Il s'agit de la spécification usuelle des booléens avec les constantes *True* et *False*, et les opérations usuelles : $\neg$ (négation), $\wedge$ (conjonction), $\vee$ (*ou* inclusif), *if_ then_ else_* .

S = { *BOOL* }

Σ = { *True* , *False* , $\neg\_$, $\_ \wedge \_$, $\_ \vee \_$, *if_ then_ else_* } avec les arités évidentes (et nécessaires ici, puisqu'il n'y a qu'une seule sorte).

E :

$$\begin{aligned}
\neg\ True &= False \\
\neg\ False &= True \\
True \wedge b &= b \\
False \wedge b &= False \\
a \vee b &= \neg\,(\neg a \wedge \neg b) \\
\textit{if True then a else b} &= a \\
\textit{if False then a else b} &= b
\end{aligned}$$

2. Les entiers naturels

Dans le cadre très simple des axiomes équationnels définis dans ce cours, les opérations "prédecesseur" (*pred*), "soustraction" (−), "division" (*div*) et "modulo" (*mod*) posent quelques problèmes, car elles présentent de manière intrinsèque des cas d'application exceptionnels, or le traitement d'exception n'a pas été abordé. Pour ne pas nuire à la complétude de cette spécification, nous poserons par convention que :

- *pred(0) = 0*
- $n - m = 0$ lorsque *m* est plus grand que *n*
- *n div 0* et *n mod 0* sont toujours égaux à 0

Une théorie des types abstraits algébriques avec traitement d'exception peut être trouvée dans [Ber86] ou [BBC86a].

Afin de pouvoir spécifier l'opération d'ordre naturel sur les entiers naturels ("<", qui est utile pour spécifier la division), nous considérons *NAT* comme une présentation au-dessus de *BOOL*. Evidemment, si l'on s'intéresse seulement aux opérations 0, *succ* (successeur), *pred* (prédécesseur), + (addition), − (soustraction) et × (multiplication), alors on peut considérer *NAT* comme une spécification à part entière, en otant les autres opérations et les axiomes les concernant.

S = { *NAT* }

Σ = { 0 , *succ_* , *pred_* , *_+_* , *_−_* , *_×_* , *_<_* , *eq(_,_)* , *if_ then_ else_* , *_div_* , *_mod_* }
avec les arités évidentes.

E :

pred(succ(n))	=	*n*
n + 0	=	*n*
n + succ(m)	=	*succ(n) + m*
n − 0	=	*n*
n − succ(m)	=	*pred(n) − m*
n × 0	=	*0*
n × succ(m)	=	*(n×m) + n*
n < 0	=	*False*
0 < succ(m)	=	*True*
succ(n) < succ(m)	=	*n < m*
eq(0,0)	=	*True*
eq(0,succ(m))	=	*False*
eq(succ(n),0)	=	*False*
eq(succ(n),succ(m))	=	*eq(n,m)*
if True then n else m	=	*n*
if False then n else m	=	*m*
n div succ(m)	=	*if n<succ(m) then n else succ((n−succ(m)) div succ(m))*
n mod succ(m)	=	*n − (n div succ(m))*

et par convention :

pred(0)	=	*0*
n div 0	=	*0*
n mod 0	=	*0*

Remarquons qu'en fait, il faudrait lever la surcharge sur l'opération "*if_ then_ else_* ", qui était déjà une opération booléenne ; mais le contexte (ici le nom des variables) nous permet aisément de différencier ces deux opérations, ce qui nous évite d'alourdir les notations.

Remarquons aussi que nos équations "par convention" n'induisent aucune inconsistance, car les axiomes tels que

n div succ(m) = if n<succ(m) then n else succ((n−succ(m)) div succ(m))

imposent que le second membre (*succ(m)*) soit différent de 0.

Remarquons enfin qu'il est par contre inutile de spécifier que *(n−m)* égale 0 lorsque *m* est plus grand que *n*, car ceci résulte de la convention "*pred(0)=0*" : le calcul de la soustraction aboutit alors à *(0−0)*.

3. Les entiers relatifs

Pour ne pas nuire à la complétude de cette spécification, nous poserons par convention que *n div 0* et *n mod 0* sont toujours égaux à 0 .

Afin de pouvoir spécifier l'opération d'ordre naturel sur les entiers relatifs ("<", qui est utile pour spécifier la division), nous considérons *INT* comme une présentation au-dessus de *BOOL*. Evidemment, si l'on s'intéresse seulement aux opérations 0, *succ* (successeur), *pred* (prédécesseur), *op* (opposé), + (addition), − (soustraction) et × (multiplication), alors on peut considérer *INT* comme une spécification à part entière, en otant les autres opérations et les axiomes les concernant.

S = { *INT* }

Σ = {0, *succ_, pred_, op_, _+_, _−_, _×_, _<_, eq(_,_), if_ then_ else_, _div_, _mod_*}
avec les arités évidentes.

E :

pred(succ(x))	=	*x*
op(0)	=	*0*
op(succ(x))	=	*pred(op(x))*
op(pred(x))	=	*succ(op(x))*
x + 0	=	*x*
x + succ(y)	=	*succ(x) + y*
x + pred(y)	=	*pred(x) + y*
x − 0	=	*x*
x − succ(y)	=	*pred(x) − y*
x − pred(y)	=	*succ(x) − y*
x × 0	=	*0*
x × succ(y)	=	*(x×y) + x*
x × pred(y)	=	*(x×y) − x*
0 < 0	=	*False*
0 < succ(0)	=	*True*
0 < succ(succ(y))	=	*if 0<succ(y) then True else 0<succ(succ(y))*
0 < pred(y)	=	*if 0<y then 0<pred(y) else False*
succ(x) < y	=	*x < pred(y)*
pred(x) < y	=	*x < succ(y)*
eq(x,y)	=	*¬ (x<y ⌄ y<x)*
if True then x else y	=	*x*
if False then x else y	=	*y*
x div y	=	*if y×d ≤ x < y×succ(d) then d else x div y*
x mod y	=	*if eq(y,0) then 0 else x − (x div y)*
x div 0	=	*0*

Les équations relatives à "<" et "*div*" méritent quelques explications :

- Concernant l'opération "<", il est bien connu que l'on ne peut pas spécifier

  0 < succ(n) = True

  comme dans les entiers naturels, car on créerait des inconsistances dans les booléens :

  0 < succ(pred(0)) = True = 0 < 0 = False

  On remarque que la spécification que nous avons donnée ici ne fournit pas un système de réécriture qui termine. Nous sommes dans un cas où l'on aurait besoin d'une opération "*if_ then_* ", mais on ne dispose ici que de "*if_ then_ else_* ". Le moyen algébrique d'obtenir un "*if_ then_* " est le suivant : chaque fois que l'on voudrait écrire

  t = si b alors t'

  on écrit :

  t = if b then t' else t

  même si cette équation ne fournit pas une système de réécriture qui termine, elle n'en est pas moins valide sur le plan purement algébrique (congruences).

- De la même façon, l'equation que nous avons donnée pour définir la division ne modélise en aucune façon un algorithme de calcul de la division euclidienne ; néanmoins elle définit

parfaitement le résultat *d* de la division de *x* par *y*, sur le plan algébrique.
Bien sûr, un moyen plus constructif de spécifier la division euclidienne est l'équation suivante, qui est cependant plus complexe :

```
x div y   =   if eq(y,0) then 0     /* par convention */
              else     /* cas général */
              (if y<0 then op(x) div op(y)
                  else     /* le diviseur y est positif */
                  (if 0≤x<y then 0    /* fin de récursion */
                      else (if y≤x
                          then succ( (x–y) div y )    /* dividende positif */
                          else pred( (x+y) div y )    /* dividende negatif */
```

(dans cet axiome, "$u \leq v$" est une abbréviation pour "$(u<v \vee eq(u,v))$" ; et "$0 \leq x<y$" est une abbréviation pour "$(0 \leq x \wedge x<y)$").
Prouver que ces deux manières de spécifier la division sont équivalentes relève du formalisme d'implémentation abstraite.

4. Les tableaux

Nous spécifions les tableaux comme une présentation *ARRAY* au-dessus de deux spécifications prédéfinies :

- Une spécification *RANG* contenant une sorte *RANG* et la sorte *BOOL*, et munie d'un prédicat d'égalité sur la sorte *RANG* (noté *eq*). Cette sorte *RANG* sera celle des indices des tableaux (le plus souvent *NAT* dans les exemples).
  Prédicat d'égalité : dans l'algèbre initiale de *RANG*, *eq(x,y)* est égal à *True* si $x=y$, et à *False* sinon. En particulier, la propriété *eq(x,y)=True* est réflexive, symétrique et transitive dans toute algèbre finiment générée validant *RANG*.

- Une spécification *ELEM* contenant une sorte *ELEM*, et munie d'une constante "distinguée" de sorte *ELEM*, notée *e* . Cette constante est nécessaire du fait qu'en l'absence de traitement d'exception, on est contraint de supposer, pour conserver la suffisante complétude de notre spécification, que le tableau "vide" (*create*) contient un élément en chacun de ses rangs. Le plus simple est de poser qu'il contient cette constante *e* uniformément.

S = { *ARRAY* }

Σ =	*create :*		→	*ARRAY*	(tableau initial)
	_ [_]:=_ :	*ARRAY RANG ELEM*	→	*ARRAY*	(affectation)
	_ [_] :	*ARRAY RANG*	→	*ELEM*	(accès)
	if_ then_ else_ :	*BOOL ARRAY ARRAY*	→	*ARRAY*	

Par convention, on notera *t* ou *t'* les variables de sorte *ARRAY*, *i* ou *j* celles de sorte *RANG*, et *x* ou *y* celles de sorte *ELEM*.

E : *if True then t else t'* = *t*
if False then t else t' = *t'*
create = *create[i]:=e*
(t[i]:=x)[j]:=y = *if eq(i,j) then t[j]:=y else (t[j]:=y)[i]:=x*
(t[i]:=x)[i] = *x*

Ces axiomes traduisent simplement que : le tableau vide contient la constante *e* partout, l'affectation dans un tableau est commutative sauf si elle concerne deux fois le même indice auquel cas la dernière affectation écrase la précédente, et l'accès à un tableau fournit le dernier élément affecté en l'indice en question (la commutativité de l'affectation permet toujours de faire remonter la "bonne" affectation à la racine du terme représentant le tableau auquel on accède).

5. Les piles

Nous spécifions les piles (non bornées) comme une présentation *STACK* au-dessus de *NAT+BOOL* (pour pouvoir spécifier l'opération "hauteur"), et d'une spécification prédéfinie *ELEM* contenant une sorte *ELEM*, et munie d'une constante "distinguée" de sorte *ELEM*, notée *e* (les éléments placés dans les piles seront ceux de sorte *ELEM*). Cette constante *e* est nécessaire du fait qu'en l'absence de traitement d'exception, on est contraint de supposer, pour conserver la suffisante complétude de notre spécification, que l'accès à la pile vide fournit un élément prédéfini. Cet élément sera *e* .
Naturellement, il est possible que *ELEM* soit en fait *NAT* ou *BOOL* (avec par exemple *e=0*).

S = { *STACK* }

Σ =	*empty :*		→ *STACK*		(pile vide)
	push :	*ELEM STACK*	→ *STACK*		(empile un élément)
	pop :	*STACK*	→ *STACK*		(depile d'un cran)
	top :	*STACK*	→ *ELEM*		(élément en haut de pile)
	height :	*STACK*	→ *NAT*		(hauteur)
	is-empty :	*STACK*	→ *BOOL*		(la pile est-elle vide ?)

E : *pop(push(x,X))* = *X*
top(push(x,X)) = *x*
height(empty) = *0*
height(push(x,X)) = *succ(height(X))*
is-empty(X) = *eq(height(X),0)*
et par convention :
pop(empty) = *empty*
top(empty) = *e*

6. Les chaînes

Nous spécifions les chaînes (non bornées) comme une présentation *STRING* au-dessus de *NAT+BOOL* (pour pouvoir spécifier l'opération "longueur"), et d'une spécification prédéfinie *ELEM*

contenant une sorte *ELEM* (les “caractères” formant les chaînes). Naturellement, il est possible que *ELEM* soit en fait *NAT* ou *BOOL*.

S = { *STRING* }

Σ =	λ :		→	*STRING*	(chaîne vide)
	add :	*ELEM STRING*	→	*STRING*	(ajoute un élément)
	concat :	*STRING STRING*	→	*STRING*	(concaténation)
	length :	*STRING*	→	*NAT*	(longueur)
	is-empty :	*STRING*	→	*BOOL*	(la chaîne est-elle vide ?)

E :	*concat*(λ,*u*)	=	*u*
	concat(*add*(*x*,*u*),*v*)	=	*add*(*x*,*concat*(*u*,*v*))
	length(λ)	=	*0*
	length(*add*(*x*,*u*))	=	*succ*(*length*(*u*))
	is-empty(*u*)	=	*eq*(*length*(*u*),*0*)

7. Les ensembles

Nous spécifions les ensembles (finis) comme une présentation *SET* au-dessus de *NAT+BOOL* (pour pouvoir spécifier l’opération “cardinal”), et d’une spécification prédéfinie *ELEM* contenant une sorte *ELEM* (les éléments des ensembles), et munie d’un prédicat d’égalité, noté *eq*, sur la sorte *ELEM*.
Naturellement, il est possible que *ELEM* soit en fait *NAT* (muni de *eq*), ou *BOOL* (le prédicat d’égalité est alors défini par : eq(a,b) = (a$\wedge$b)$\vee\neg$(a$\vee$b))

S = { *SET* }

Σ =	*if_ then_ else_* :	*BOOL SET SET*	→	*SET*	
	∅ :		→	*SET*	(ensemble vide)
	ins :	*ELEM SET*	→	*SET*	(insertion)
	del :	*ELEM SET*	→	*SET*	(enlever un élément)
	∈ :	*ELEM SET*	→	*BOOL*	(élément de)
	∪ :	*SET SET*	→	*SET*	(union)
	∩ :	*SET SET*	→	*SET*	(intersection)
	− :	*SET SET*	→	*SET*	(différence)
	Δ :	*SET SET*	→	*SET*	(différence symétrique)
	card :	*SET*	→	*NAT*	(cardinal)
	is-empty :	*SET*	→	*BOOL*	(l’ensemble est-il vide ?)

E =

if True then X else Y	=	*X*
if False then X else Y	=	*Y*
ins(x,ins(y,X))	=	*if eq(x,y) then ins(y,X) else ins(y,ins(x,X))*
del(x,ins(y,X))	=	*if eq(x,y) then del(x,X) else ins(y,del(x,X))*
$x \in \varnothing$	=	*False*
$x \in ins(y,X)$	=	*if eq(x,y) then True else* $x \in X$
$X \cup \varnothing$	=	*X*
$X \cup ins(x,Y)$	=	$ins(x, X \cup Y)$
$X \cap \varnothing$	=	$\varnothing$
$X \cap ins(x,Y)$	=	*if* $x \in X$ *then* $ins(x, X \cap Y)$ *else* $X \cap Y$
$X - \varnothing$	=	*X*
$X - ins(x,Y)$	=	$del(x,X) - Y$
$X \Delta Y$	=	$X \cup Y - X \cap Y$
$card(\varnothing)$	=	*0*
card(ins(x,X))	=	*if* $x \in X$ *then card(X) else succ(card(X))*
is-empty(X)	=	*eq(card(X),0)*

et par convention :

$del(x,\varnothing)$	=	$\varnothing$

8. Les files

Nous spécifions les files (“FIFO”, non bornées) comme une présentation *QUEUE* au-dessus de *NAT+BOOL* (pour pouvoir spécifier l’opération “longueur”), et d’une spécification prédéfinie *ELEM* contenant une sorte *ELEM*, et munie d’une constante “distinguée” de sorte *ELEM*, notée *e* (les éléments placés dans les files seront ceux de sorte *ELEM*). Cette constante *e* est nécessaire du fait qu’en l’absence de traitement d’exception, on est contraint de supposer, pour conserver la suffisante complétude de notre spécification, que l’accès à la file vide fournit un élément prédéfini. Cet élément sera *e* .
Naturellement, il est possible que *ELEM* soit en fait *NAT* ou *BOOL* (avec par exemple *e=0*).

S = { *QUEUE* }

Σ =

new :		→ *QUEUE*		(file vide)
add :	*ELEM QUEUE*	→ *QUEUE*		(ajoute un élément)
remove :	*QUEUE*	→ *QUEUE*		(detruit l’élément en tête)
first :	*QUEUE*	→ *ELEM*		(premier élément de la file)
length :	*QUEUE*	→ *NAT*		(longueur)
is-empty :	*QUEUE*	→ *BOOL*		(la file est-elle vide ?)

E : $remove(add(x,new)) = new$

$remove(add(x,add(y,X))) = add(x,remove(add(y,X)))$

$first(add(x,new)) = x$

$first(add(x,add(y,X))) = first(add(y,X))$

$length(new) = 0$

$length(add(x,X)) = succ(length(X))$

$is\text{-}empty(X) = eq(length(X),0)$

et par convention :

$remove(new) = new$

$first(new) = e$

Chapitre D : Quelques résultats complémentaires
Good functors ... are those preserving philosophy !

Gilles BERNOT

LIENS, CNRS URA 1327
Ecole Normale Supérieure,
45 Rue d'Ulm,
F-75230 PARIS Cédex 05,
FRANCE

bitnet: bernot@frulm63
uucp: bernot@ens.ens.fr

(Appeared in Proc. of "Category Theory and Computer Science", LNCS 283, 1987.)

Résumé

Le but de ce chapitre est de mettre en garde le chercheur en types abstraits algébriques contre une utilisation abusive de la théorie des catégories. Quelques propriétés peu souhaitables du (pourtant classique) *foncteur de synthèse* sont décrites, spécialement si l'on s'intéresse aux sémantiques dites "loose". Tous les résultats énoncés ici sont particulièrement simples, sinon triviaux ; néanmoins, ils illustrent des faits donnant lieu à de nombreuses erreurs dans le cadre des types abstraits algébriques. Ces erreurs résultent souvent d'une inadéquation entre certains outils catégoriques bien connus et le concept informatique que l'on souhaite modéliser. Enfin, une approche hiérarchique fondée sur la catégorie des modèles "protégeant les sortes prédéfinies" est proposée, et les premières propriétés en sont dégagées.

Mots clés : complétude, consistance, modèle initial, spécifications abstraites, spécifications structurées, théorie des catégories, types abstraits algébriques.

Abstract

The aim of this paper is to prevent the abstract data type researcher from an improper, naive use of category theory. We mainly emphasize some unpleasant properties of the *synthesis functor* when dealing with so-called *loose semantics* in a hierarchical approach. All our results and counter-examples are very simple, nevertheless they shed light on many common errors in the abstract specification field.
We also summarize some properties of the category of models "protecting predefined sorts."

Keywords : abstract data types, abstract specifications, category theory, completeness, consistency, initial model, structured specifications.

1 Introduction

In the following pages, we focus our attention on results which seem to be "trivially ensured" in the basic abstract data type framework. We sometimes give proofs... often counter-examples of such results. In order to get striking counter-examples, we provide very simple ones, if not trivial (mainly based on elementary algebraic properties of natural numbers). Nevertheless, many common errors, or misinterpretations found in the abstract data type litterature result from similar mechanisms. This emphasizes the fact that category theory should be carefully used in the abstract data type field, including for (very) low level concepts.
More provocatively: this paper mainly points out the fact that the synthesis functor $\mathcal{F}$ of abstract data types "does not preserve philosophy." However, since about teen years [ADJ76], it is well known that this functor is crucial for defining a hierarchical, modular approach of abstract specifications!

Some elementary reminders about abstract data types are given in the next section (Section 2). Section 3 discusses about the well known *forgetful* and *synthesis functors*, $\mathcal{U}$ and $\mathcal{F}$, associated with a hierarchical approach. Section 4 shows the difficulty of properly defining sufficient completeness and hierarchical consistency with loose semantics. In Section 5, we show what happens when combining enrichments. Lastly, Section 6 discusses about a loose semantics obtained by "protecting" predefined sorts.

The following discussions are mainly centered on pairs [positive fact / proof] (respectively: [negative fact / counter-example]).

2 Elementary reminders

Let us begin with basic definitions and properties [ADJ76]:
Given a *signature* Σ (i.e. a finite set S of *sorts* and a finite set Σ of *operation-names* with arity in S), a Σ-algebra, A, is a heterogeneous set partitioned as $\{ A_s \}_{s \in S}$, and for each operation-name $op : s_1 \cdots s_{n-1} \rightarrow s_n$ of Σ there is an operation $op_A : A_{s_1} \times \cdots \times A_{s_{n-1}} \rightarrow A_{s_n}$.
A Σ-*morphism* from A to B is a sort-preserving, operation-preserving application from A to B. This defines a category, denoted by $Alg(\Sigma)$; it has an initial object: the ground-term algebra T_Σ .
In the following, a *specification* $SPEC$ will be defined by a signature Σ and a finite set E of *positive conditional equations* of the form:

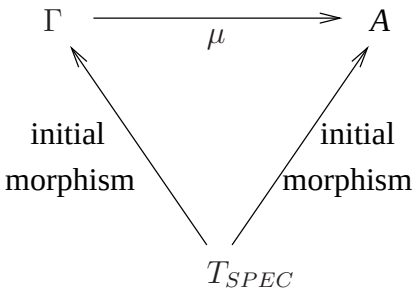
$$v_1 = w_1 \ \wedge \cdots \wedge \ v_{n-1} = w_{n-1} \Longrightarrow v_n = w_n$$

where v_i and w_i are Σ-terms with variables.
Given a specification $SPEC$, $Alg(SPEC)$ is the full sub-category of $Alg(\Sigma)$ whose objects are the Σ-algebras which validate each axiom of E. The category $Alg(SPEC)$ has an initial object, denoted by T_{SPEC} [BPW82].
Since T_{SPEC} exists, $Gen(SPEC)$ can be defined as the full sub-category of $Alg(SPEC)$ such that the initial morphism is an epimorphism (i.e. is *surjective*, in our framework). $Gen(SPEC)$ is the category of the *finitely generated algebras*. Our first "fact" will be devoted to the following remark:
It is well known that $Gen(SPEC)$ is a particularly interesting category for the abstract

data type computer scientist; nevertheless, this is not exactly due to its large spectrum of morphisms, as reminded below.

fact 1 : (Morphisms from a finitely generated algebra)
Let Γ be an object of $Gen(SPEC)$ and A an object of $Alg(SPEC)$. The set $Hom_{Alg(SPEC)}(\Gamma, A)$ contains *at most one element*. Consequently, for all objects X and Y of $Gen(SPEC)$, $Hom_{Gen(SPEC)}(X, Y)$ contains at most one morphism.

Proof: By initiality properties, if there exists a morphism μ, then the following triangle commutes:



Thus, the unicity of μ results from the surjectivity of the initial morphism associated with Γ. □

One of the most important aspect of abstract data types is its structured, hierarchical, modular approach. This is obtained by means of *presentations*. A presentation $PRES$ over $SPEC$ is a new "part of specification" $PRES =< S', \Sigma', E' >$ such that the disjoint union $SPEC + PRES =< S \cup S',\ \Sigma \cup \Sigma',\ E \cup E' >$ is a specification. Sorts and operations of $SPEC$ are often called the *predefined* sorts and operations. Relations between the categories $Alg(SPEC)$ and $Alg(SPEC + PRES)$ are handled by the well known *forgetful functor* and *synthesis functor*:

$$(\mathcal{U} \ :\ Alg(SPEC + PRES) \rightarrow Alg(SPEC)) \text{ and}$$
$$(\mathcal{F} \ :\ Alg(SPEC) \rightarrow Alg(SPEC + PRES)).$$

The functor $\mathcal{F}$ is a left adjoint for the functor $\mathcal{U}$. Consequently, for each $SPEC$-algebra A, there is a particular morphism from A to $\mathcal{U}(\mathcal{F}(A))$: the morphism deduced from the adjunction unit (or *adjunction morphism*). This morphism is absolutely crucial for the hierarchical approach: it allows to evaluate the modifications performed on A under the action of $PRES$.

Example 2 : If A is equal to $\mathbb{N}$ over the signature $\{0,\ succ\_\}$ (without axioms) and if $PRES$ adds $pred\_$ with the axioms $[pred(succ(n)) = succ(pred(n)) = n]$, then $\mathcal{U}(\mathcal{F}(\mathbb{N}))$ is isomorphic to $\mathbb{Z}$. The unit of adjunction leads to the natural inclusion; and this morphism permits to show that $\mathbb{N}$ has been modified by adding negative values.
If the axioms were $[pred(succ(n)) = n$ and $pred(0) = 0]$, then the unit of adjunction leads to the identity over $\mathbb{N}$ showing that this second specification of $pred$ does not change $\mathbb{N}$.

3 Forgetful and synthesis functors

We first present a rather obvious reminder about the *forgetful functor*. Let B be a $SPEC+PRES$-algebra. The forgetful functor removes all subsets B_s where $s \in S'$, and all operations of Σ' are forgotten (including those with arity in S only), but *it does not remove any value of predefined sort*: $\mathcal{U}(B_s) = B_s$ for each $s \in S$. For instance, in Example 2, $\mathcal{U}(\mathbb{Z}) = \mathbb{Z} \neq \mathbb{N}$.

Let us remind the classical definition of the *synthesis functor* (although classical, this definition is the starting point of some misinterpretations!): Let A be a $SPEC$-algebra and let $T_{\Sigma+\Sigma'}(A)$ be the algebra of $\Sigma + \Sigma'$-terms with variables in A; we denote by $eval : \mathcal{U}(T_{\Sigma+\Sigma'}(A)) \rightarrow A$ the canonical evaluation morphism. $\mathcal{F}(A)$ is the quotient of $T_{\Sigma+\Sigma'}(A)$ by the smallest congruence containing both the fibers of $eval$ and the close instanciations of $E + E'$.
Because $E + E'$ is required in the definition of $\mathcal{F}$ (instead of E' alone), $\mathcal{F}(A)$ does not only depend on A and $PRES$; it also depends on $SPEC$.

fact 3 : Given a presentation $PRES$, the action of the synthesis functor $\mathcal{F}$ over a given, fixed algebra A is highly dependent of the predefined specification.
As outlined in the following example, this fact considerably restricts the possibility of writing "implementation independent" specifications (see for instance [EKMP80], [SW82], or [BBC86a] about abstract implementations).

Example 4 : Let $SPEC$ be a classical specification of NAT with operations 0, $succ\_$ and $\_ + \_$:

$$x + 0 = x$$
$$x + succ(y) = succ(x + y)$$

Let $SPEC'$ be the specification obtained by adding the following axiom to $SPEC$:

$$x + y = x + z \Longrightarrow y = z$$

The specifications $SPEC$ and $SPEC'$ have clearly the same initial object: $\mathbb{N}$.
Let $PRES$ be the presentation adding no sort, adding the operation $\_ \times \_$, and adding the axioms:

$$x \times succ(0) = x > \text{(1 is neutral)}$$
$$x \times succ(y) = x + (x \times y) > \text{(recursive definition)}$$

When $PRES$ is shown as a presentation over $SPEC$, $\mathcal{F}(\mathbb{N})$ is a model where all terms containing a multiplication by 0 cannot be evaluated. When $PRES$ is shown as a presentation over $SPEC'$, $\mathcal{F}(\mathbb{N})$ is isomorphic to $\mathbb{N}$, because:

$$x + 0 = x = x \times succ(0) = x + (x \times 0)$$

and the simplification axiom of $SPEC'$ leads to $0 = x \times 0$.
Notice that, in spite of the fact that $SPEC$ and $SPEC'$ have the same initial semantics, the presentation $PRES$ is not completely specified over the first specification, but is completely specified over the second one.

4 Consistency and completeness

The subject of this section is an examination of some *a priori* possible definitions of the notions of *sufficient completeness* and *hierarchical consistency* with loose semantics. We start with the most loose semantics: the entire category $Alg(SPEC)$. We will show that the simplest definitions are unacceptable for abstract specification purposes.

All the counter-examples provided in this section are based on the following specification+presentation example. Hopefully, we believe that this counter-example cannot be suspected to be too much unusual, complicated or *ad hoc*.

Example 5 : Let $SPEC$ be a specification of natural numbers (for instance the specification given in Example 4) together with a sort $BOOL$ and boolean operations $True$ and $False$. We consider the presentation $PRES$ enriching $SPEC$ by an equality predicate $eq?$:

$$eq?(0,0) = True$$
$$eq?(0, succ(n)) = False$$
$$eq?(succ(m), 0) = False$$
$$eq?(succ(m), succ(n)) = eq?(m,n)$$

Looking at this presentation $PRES$, we can affirm that a "good notion" of sufficient completeness (resp. hierarchical consistency) should be satisfied by $PRES$. This example is simply written by taking into account each possible value for the arguments of $eq?$, with respect to the constructors of $SPEC$, moreover there are no axioms between constructors (fair presentation [Bid82]).

We may of course imagine more sophisticated presentation examples, in particular examples which add new sorts to $SPEC$. But our goal is simply to prevent the abstract data type researcher from using a naive, rather unrealistic definition of sufficient completeness or hierarchical consistency.

4.1 Sufficient completeness

In the initial approach, sufficient completeness is defined as follows [Gau78].

"*The adjunction morphism associated with the initial algebra is surjective:*"

$$T\_SPEC \rightarrow \mathcal{U}(\mathcal{F}(T\_SPEC))$$

This condition exactly means that $PRES$ does not add new values to T_SPEC . Remind that $\mathcal{F}(T\_SPEC) = T_{SPEC+PRES}$, due to adjunction properties.

fact 6 : The following definition of sufficient completeness is not suitable in the general case:
"$PRES$ is sufficiently complete if and only if for all algebras in $Alg(SPEC)$ the adjunction morphism is surjective".

Using Example 5, we convince ourselves of this fact by considering the $SPEC$-algebra obtained by two copies of $\mathbb{N}$. This algebra, ($\mathbb{N} \times \{0,1\}$ and $\{True, False\}$) , is not finitely generated, but is an object of $Alg(SPEC)$ by sending the operation-name 0 over the element $(0,0)$, and $succ((n,a)) = (succ(n), a)$. Terms of the form $eq?((n,0),(m,1))$ cannot be evaluated using the $PRES$ axioms of Example 5. Consequently, they add new boolean values, and the adjunction morphism is not surjective.

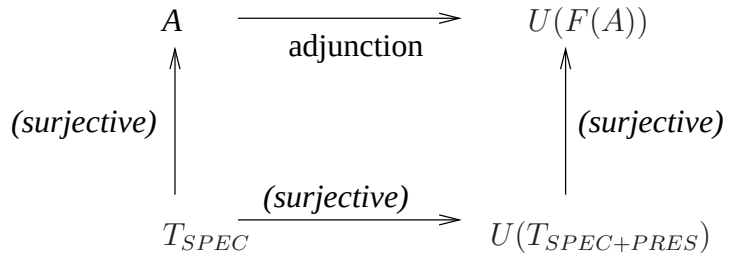
fact 7 : The following two definitions of sufficient completeness are logically equivalent:

1. the adjunction morphism associated with the initial algebra T_SPEC is surjective
2. for all algebras in $Gen(SPEC)$ the adjunction morphism is surjective.

Proof: $[2 \Longrightarrow 1]$ is trivial because the initial algebra is finitely generated.
$[1 \Longrightarrow 2]$: let A be a finitely generated $SPEC$-algebra. By construction of $\mathcal{F}$, $\mathcal{F}(A)$ is finitely generated over the signature of $SPEC + PRES$. Consequently, the image of the initial morphism via the forgetful functor is surjective:

$$\mathcal{U}(init_A) : \mathcal{U}(\mathcal{F}(T\_SPEC)) = \mathcal{U}(T_{SPEC+PRES}) \rightarrow \mathcal{U}(\mathcal{F}(A))$$

Our conclusion results from the commutativity of the following diagram:



□

Restricting ourselves to finitely generated algebras has several disadvantages. For instance, parameterized presentations require a non finitely generated semantics [ADJ80].

4.2 Hierarchical consistency

In the initial approach, hierarchical consistency is defined as follows:

"*the adjunction morphism associated with the initial algebra is a monomorphism*"

(i.e. is *injective* in our framework).

fact 8 : The following definition of hierarchical consistency is not suitable in the general case:
"$PRES$ is hierarchically consistent if and only if for all algebras in $Alg(SPEC)$ the adjunction morphism is injective".

Let us return to Example 5. If we consider the $SPEC$-algebra $\mathbb{Z}$ (which is a non finitely generated algebra), we get the following inconsistency:

$$True = eq?(0,0) = eq?(0, succ(-1)) = False$$

Restricting hierarchical consistency checks to finitely generated algebras does not yield better results:

fact 9 : The following definition of hierarchical consistency is not suitable in the general case:
"$PRES$ is hierarchically consistent if and only if for all algebras in $Gen(SPEC)$ the adjunction morphism is injective".

Using Example 5 again, we consider a finitely generated algebra of the form $\frac{\mathbb{Z}}{n\mathbb{Z}}$, and we get the following inconsistency:

$$True = eq?(0,0) = eq?(0,n) = eq?(0, succ(n-1)) = False$$

These facts prove that "defining sufficient completeness on $Alg(SPEC)$" , "defining hierarchical consistency on $Alg(SPEC)$" or "defining hierarchical consistency on $Gen(SPEC)$" are too strong requirements. Extension from the purely initial semantics to a loose semantics must be done more carefully.

5 Combining presentations

In the remainder of this paper, we simply follow the definitions of sufficient completeness and hierarchical consistency given at the beginning of sections 4.1 and 4.2 (i.e. the initial approach). Given a specification $SPEC$, we consider two presentations $PRES_1$ and $PRES_2$ *with disjoint signatures.*
Let $PRES$ be the union of $PRES_1$ and $PRES_2$, we care about the sufficient completeness and hierarchical consistency of $PRES$. In spite of the strong hypothesis described here, we have sometimes to be careful, as detailed in the following two subsections.

5.1 Sufficient completeness

fact 10 : If $PRES_1$ and $PRES_2$ are both sufficiently complete over $SPEC$, then $PRES = PRES_1 + PRES_2$ remain sufficiently complete. Moreover, under the same hypothesis, $PRES_2$ is sufficiently complete over $SPEC + PRES_1$.

Proof: *(using elementary tools)*
$T_{SPEC+PRES_1+PRES_2}$ is the quotient of $T_{\Sigma+\Sigma_1+\Sigma_2}$ by the smallest congruence containing the close instanciations of the $SPEC + PRES_1 + PRES_2$ axioms [BPW82]. Consequently, it suffices to prove that each $\Sigma + \Sigma_1 + \Sigma_2$-ground-term of sort in S (resp. in $S + S_1$) belongs to the equivalence class of a Σ-term (resp. $\Sigma + \Sigma_1$-term). This can be trivially proved via structural induction. □

Obviously, the converse is false: the sufficient completeness of $PRES$ does not imply the sufficient completeness of $PRES_1$ or $PRES_2$.

5.2 Hierarchical consistency

fact 11 : The hierarchical consistency of $PRES_1$ and $PRES_2$ over $SPEC$ *does not imply* the hierarchical consistency of $PRES = PRES_1 + PRES_2$ over $SPEC$.

Example 12 : Let $SPEC$ be a specification of natural numbers. Let $PRES_1$ be the presentation simply containing the following axiom:

$$succ(n) = 0 \Longrightarrow n = 0$$

$PRES_1$ is clearly consistent (in fact, the premise cannot be satisfied in the initial object, thus this axiom is never applied). Let $PRES_2$ be the presentation adding the operation $pred\_$ with $[pred(succ(n)) = succ(pred(n)) = n]$. $PRES_2$ is clearly hierarchically consistent over natural numbers (even though it is not sufficiently complete). The union $PRES = PRES_1 + PRES_2$ is not hierarchically consistent because from $succ(pred(0)) = 0$ we get:

$$0 = pred(0) \text{ , which leads to } succ(0) = succ(pred(0)) = 0$$

Another example of the same fact is the following:

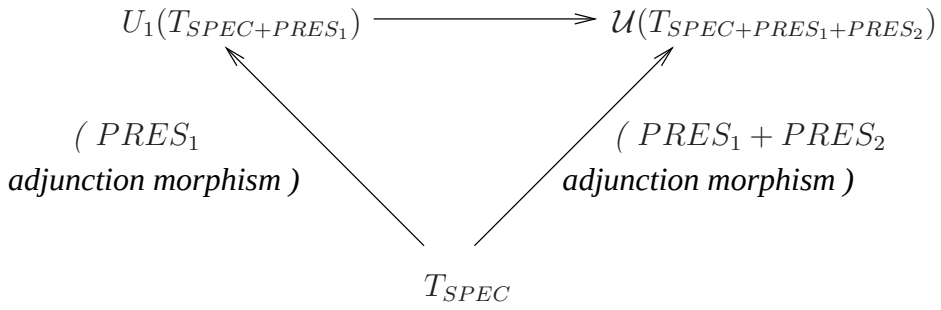
Example 13 : Let $PRES_1$ be the presentation described in Example 5 (adding equality predicate to natural numbers), and let $PRES_2$ be the same presentation as Example 12 before (adding $pred$). $PRES_1$ and $PRES_2$ are clearly hierarchically consistent over natural numbers, but the union $PRES = PRES_1 + PRES_2$ is not hierarchically consistent because:

$$True = eq?(0, 0) = eq?(0, succ(pred(0))) = False$$

(a similar example was first presented in [EKP80], for abstract implementation purposes).

fact 14 : If $PRES = PRES_1 + PRES_2$ is hierarchically consistent over $SPEC$ then $PRES_1$ and $PRES_2$ are hierarchically consistent over $SPEC$.

Proof: Assume that $PRES_1$ is not consistent: the morphism from T_{SPEC} to $\mathcal{U}(T_{SPEC+PRES_1})$ is not injective. Since the following diagram commutes, the adjunction morphism from T_{SPEC} to $T_{SPEC+PRES_1+PRES_2}$ is not injective:



(the horizontal arrow is the forgetful of the adjunction morphism for $PRES_2$ over $SPEC+PRES_1$).
It results that $PRES$ is not hierarchically consistent over $SPEC$. □

fact 15 : If $PRES_1$ and $PRES_2$ are both hierarchically consistent and sufficiently complete over $SPEC$, then $PRES = PRES_1 + PRES_2$ too. Moreover, under the same hypothesis, $PRES_2$ is hierarchically consistent and sufficiently complete over $SPEC + PRES_1$.

(This fact is well known; a demonstration with conditional axioms, including exception handling, can be found in [Ber86]).

6 Loose semantics with "Protect"

Clearly, abstract specifications do not necessarily directly lead to executable specifications. It is often convenient to specify some operations via "universal properties." For instance the subtraction can be specified via:

$$z - y = x \Longleftrightarrow x + y = z$$

Sometimes, such axioms may lead to uncompletely specified presentations, as in the following example.

Example 16 : Let $SPEC$ be an initial specification of integers with operations 0, $succ\_$, $pred\_$, $\_ + \_$, $\_ - \_$ and $\_ \times \_$. Let us specify a presentation $PRES$ adding the operation $\_div\_$ as follows:

$$0 \Longleftarrow (a - (b \times (a div b))) = True$$

$$(a - (b \times (a div b))) < b = True$$

These axioms characterize $(a\,div\,b)$ among all integers *finitely generated* with respect to *succ* and *pred*. However, in the initial model $T_{SPEC+PRES}$, the term $(a\,div\,b)$ is not reached by *succ* and *pred* . Its value is only a unreachable value such that the (unreachable) remainder $(a - (b \times (a\ div\ b)))$ returns the specified boolean values when compared with 0 and b .
Consequently, this presentation is uncompletely specified according to the usual definition of sufficient completeness.

In such examples, the only interesting models are those which do not modify the predefined initial model ($\mathbb{Z}$). This leads to a (loose) semantics where models are those *protecting* predefined sorts [Kam80]. Indeed, when writing relatively large specifications, this semantics seems to be highly suitable (ASL [Wir82] [SW83], PLUSS [Gau84], OBJ [FGJM85], LARCH [GH83]...).

Let us define the associated category:

Definition 17 : (The "Protect" category)
Let $SPEC$ be a specification and let $PRES$ be a presentation over $SPEC$. The category of $PRES$-models protecting $SPEC$ is the full subcategory of $Alg(SPEC + PRES)$ whose objects are the $SPEC + PRES$-algebras A such that $\mathcal{U}(A)$ is isomorphic to the initial predefined algebra T_{SPEC} . We denote this category by $Prot(SPEC, PRES)$.

Notice that the object class of $Prot(SPEC, PRES)$ can be empty.

fact 18 : If $Prot(SPEC, PRES)$ is not an empty category, then $PRES$ is hierarchically consistent over $SPEC$.

(Here, consistency is defined with respect to the initial algebra T_{SPEC} only)

Proof: If $T_{SPEC_{+PRES}}$ is inconsistent over T_{SPEC}, then a fortiori all $SPEC + PRES$-algebras are inconsistent over T_{SPEC} (because $T_{SPEC_{+PRES}}$ is minimal). □

fact 19 : Even if $PRES$ is consistent over $SPEC$, $Prot(SPEC, PRES)$ may be empty.

Example 20 : Let $SPEC$ be the boolean specification with $True$ and $False$. Let $PRES$ be a specification of $SET(BOOL)$ with $\emptyset$, $insert$, $\in$ and $choose$:

$True \in \emptyset = False$
$False \in \emptyset = False$
$b \in insert(b', X) = (b{=}b')\ or\ b \in X$
$choose(X) \in X = True$

This specification is clearly hierarchically consistent (even though it is not sufficiently complete). However, the Protect category is empty, because the term $choose(\emptyset)$ can neither be equal to $True$ nor to $False$ (both choices induce $True = False$).
(Fortunately, this example can be easily specified without inconsistency using abstract data types with exception handling [Bid84] [GDLE84] [BBC86b] [Ber86], or with partial functions [BW82].)

fact 21 : Even if $Prot(SPEC, PRES)$ contains models, it has not necessarily an initial object.

Example 22 : Let $SPEC$ be the boolean specification with $True$ and $False$. Let $PRES$ be the presentation adding the constant operation $maybe$, without any axiom. $Prot(SPEC, PRES)$ contains two models, no one is initial.

fact 23 : If $PRES$ is sufficiently complete over $SPEC$, then either the category $Prot(SPEC, PRES)$ has an initial object, either it is empty.

Proof: If $PRES$ is consistent, then the initial model $T_{SPEC_{+PRES}}$ belongs to $Prot(SPEC, PRES)$; it is then necessarily initial in $Prot(SPEC, PRES)$. If $PRES$ is not hierarchically consistent, then Fact 18 implies that $Prot(SPEC, PRES)$ has no object. □

fact 24 : There are presentations $PRES$ which are not sufficiently complete over $SPEC$, such that $Prot(SPEC, PRES)$ is not empty and has an initial object.

It suffices to refer to Example 16, where the axioms characterize div by a "universal property among integers." The division is incompletely specified according to classical initial definition of sufficient completeness, but $Prot(SPEC, PRES)$ only contains one model ($\mathbb{Z}$) which is necessarily initial.

7 Conclusion

We have investigated how a hierarchical approach of abstract data types, with the notions of *hierarchical consistency* and *sufficient completeness*, could be defined when dealing with so-called *loose semantics*. The results shown in sections 2 to 5 seem to be somewhat pessimistic:

- The synthesis functor is "implementation dependent" with respect to the predefined specification (Fact 3).
- Sufficient completeness cannot be checked on all models (Fact 6).
- Hierarchical consistency cannot be checked on all models (Fact 8).
- Hierarchical consistency cannot be checked on all finitely generated models, a smaller class of models must be investigated (Fact 9).
- Combining hierarchically consistent presentations does not result on a hierarchically consistent presentation (Fact 11).

However, we showed some positive results:

- Checking sufficient completeness on all finitely generated algebras is equivalent to check it on the initial algebra only (Fact 7).
- Combining sufficiently complete presentations results on sufficiently complete presentations (Fact 10); the same occurs for presentations that are both sufficiently complete and hierarchically consistent (Fact 15).

In the last section (Section 6), we defined the category of models *protecting predefined sorts*. We have investigated the relations between the classical notions of completeness/consistency and the elementary properties of this category:

- The category is empty if the presentation is not hierarchically consistent, but the converse is false (Facts 18 and 19).
- The category has not necessarily initial models (Fact 21).
  It has initial models if the presentation is sufficiently complete and hierarchically consistent, but the converse is false (Facts 23 and 24).

In conclusion: From facts 3, 6, 8, 9 and 11, we showed that the synthesis functor of classical abstract data types "does not always preserve philosophy" when dealing with loose semantics. Moreover, with a loose semantics based on protection of predefined sorts, the corresponding category has few systematic relations with sufficient completeness or hierarchical consistency (facts 18 to 24).

Acknowledgements: It is a pleasure to express gratitude to Michel Bidoit, Christine Choppy and Marie-Claude Gaudel for encouragements to write this paper and careful proof readings. The title was suggested by Stephane Kaplan.

References

[ADJ76] Goguen J., Thatcher J., Wagner E. : "*An initial algebra approach to the specification, correctness, and implementation of abstract data types*", Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978. Also : IBM Report RC 6487, Oct. 1976.

[ADJ80] Ehrig H., Kreowski H., Thatcher J., Wagner J., Wright J. : "*Parameterized data types in algebraic specification langages*", Proc. 7th ICALP, July 1980.

[BBC86a] Bernot G., Bidoit M., Choppy C. : "*Abstract implementations and correctness proofs*", Proc. 3rd STACS, January 1986, Springer-Verlag LNCS 210, January 1986. Also : LRI Report 250, Orsay, Dec. 1985.

[BBC86b] Bernot G., Bidoit M., Choppy C. : "*Abstract data types with exception handling : an initial approach based on a distinction between exceptions and errors*", Theoretical Computer Science, Vol.46, No.1, p.13-45, November 1986.

[Ber86] Bernot G. : "*Une sémantique algébrique pour une spécification différenciée des exceptions et des erreurs : application à l'implémentation et aux primitives de structuration des spécifications formelles*", Thèse de troisième cycle, LRI, Université de Paris-Sud, Orsay, Février 1986.

[Bid82] Bidoit M. : "*Algebraic data types: structured specifications and fair presentations*", Proc. AFCET Symposium on Mathematics for Computer Science, Paris, March 1982.

[Bid84] Bidoit M. : "*Algebraic specification of exception handling by means of declarations and equations*", Proc. 11th ICALP, Springer-Verlag LNCS 172, July 1984.

[BPW82] Broy M., Pair C., Wirsing M. : "*A systematic study of models of abstract data types*", Theoretical Computer Sciences, p. 139-174, vol. 33, October 1984.

[BW82] Broy M., Wirsing M. : "*Partial abstract data types*", Acta Informatica, Vol.18-1, Nov. 1982.

[EKMP80] Ehrig H., Kreowski H., Mahr B., Padawitz P. : "*Algebraic implementation of abstract data types*", Theoretical Computer Science, Oct. 1980.

[EKP80] Ehrig H., Kreowski H., Padawitz P. : "*Algebraic implementation of abstract data types: concept, syntax, semantics and correctness*", Proc. ICALP, Springer-Verlag LNCS 85, 1980.

[FGJM85] Futatsugi K., Goguen J., Jouannaud J-P., Meseguer J. : "*Principles of OBJ2*", Proc. 12th ACM Symp. on Principle of Programming Languages, New Orleans, January 1985.

[Gau78] Gaudel M-C. : "*Spécifications incomplètes mais suffisantes de la représentation des types abstraits*", Laboria Report 320, 1978.

[Gau84] Gaudel M-C. : "*A first introduction to PLUSS*", LRI Report, Orsay, December 1984.

[GDLE84] Gogolla M., Drosten K., Lipeck U., Ehrich H.D. : "*Algebraic and operational semantics of specifications allowing exceptions and errors*", Theoretical Computer Science 34, North Holland, 1984.

[GH83] Guttag J.V., Horning J.J. : "*An introduction to the LARCH shared language*", Proc. IFIP 83, REA Mason ed., North Holland Publishing Company, 1983.

[Kam80] Kamin S. : "*Final data type specifications : a new data type specification method*", Proc. of the 7th POPL Conference, 1980.

[SW82] Sannella D., Wirsing M. : "*Implementation of parameterized specifications*", Report CSR-103-82, Department of Computer Science, University of Edinburgh.

[SW83] Sannella D., Wirsing M. : "*A kernel language for algebraic specification and implementation*", Proc. Intl. Conf. on Foundations of computation Theory, Springer-Verlag, LNCS 158, 1983.

[Wir82] Wirsing M. : "*Structured algebraic specifications*", Proc. of AFCET Symposium on Mathematics for Computer Science, Paris, March 1982.