ORSAY
n° d'ordre : *4018*

UNIVERSITE DE PARIS-SUD
CENTRE D'ORSAY

THESE

présentée
Pour obtenir

Le DIPLOME de DOCTEUR de 3e CYCLE

PAR

Gilles BERNOT

———

UNE SEMANTIQUE ALGEBRIQUE POUR UNE SPECIFICATION DIFFERENCIEE DES EXCEPTIONS ET DES ERREURS ; APPLICATION A L'IMPLEMENTATION ET AUX PRIMITIVES DE STRUCTURATION DES SPECIFICATIONS FORMELLES

soutenue le 26 Février 1986 devant la Commission d'examen

MM. *JOUANNAUD Jean-Pierre* président
BIDOIT Michel
CHOPPY Christine
GAUDEL Marie-Claude
GUESSARIAN Irène
VIDAL-NAQUET Guy

REMERCIEMENTS

Michel Bidoit et Christine Choppy dirigent mes travaux depuis le début de mon D.E.A. ; je les remercie de m'avoir tant appris et tant stimulé, lisant avec constance les nombreuses versions qui ont précédé ce texte. Nous avons souvent partagé, face à un coin de tableau noir, des sourires triomphants (resp. moues dépitées) devant une démonstration élégante (resp. un contre-exemple insidieux) d'un énoncé qui, intuitivement, "ne pouvait qu'être vrai".
Marie-Claude Gaudel a su, malgré un emploi du temps extrêmement chargé, m'apporter un soutien constant et me prodiguer de très nombreux conseils, particulièrement dans le domaine de l'implémentation abstraite.
Tous trois m'ont patiemment et chaleureusement guidé sur la voie difficile de la recherche en informatique. Qu'ils soient assurés de mon affection.

Je remercie Jean-Pierre Jouannaud de m'avoir fait l'honneur de présider ce jury, et pour l'attention qu'il a porté à mon travail ; nos discussions ont été fructueuses, particulièrement dans le domaine du traitement d'exceptions.

Je remercie vivement Guy Vidal-Naquet pour l'intérêt constant qu'il a porté à mon travail depuis le D.E.A., pour ses encouragements, et pour sa participation à ce jury.

Je remercie Irène Guessarian pour sa participation à ce jury, et pour l'intérêt qu'elle a manifesté pour mon travail.

Merci aussi à tous les membres de l'équipe Programmation – Génie Logiciel pour leur soutien amical.
Un clin d'œil aux "hackers" du L.R.I. ... sans lesquels je n'aurais pas disposé d'un environnement aussi souple après chaque "login" ! Et une mention spéciale pour Francis Capy, pour son aide, son amitié, et la complicité d'un bureau partagé.

Je dois enfin beaucoup, et plus encore, à mes parents qui ont toujours eu mes études à cœur, et sur lesquels j'ai toujours pu compter.

TABLE DES MATIERES GENERALE

TABLE DES MATIERES GENERALE (SUITE)

INTRODUCTION

INTRODUCTION

Le but de cette thèse est principalement de présenter un nouveau formalisme de traitement d'exceptions (les *exception-algèbres*), de l'utiliser pour traiter l'implémentation abstraite en présence d'exceptions, et plus généralement de montrer que, les résultats fondamentaux obtenus pour les exception-algèbres étant analogues aux résultats fondamentaux bien connus des types abstraits algébriques "classiques", la plupart des primitives de structuration des spécifications algébriques obtenues dans le cas sans traitement d'exceptions peuvent être étendues sans difficulté aux exception-algèbres.

Dans le cadre du traitement d'exceptions, les questions *hiérarchiques* (telles que la consistance hiérarchique ou la suffisante complétude) constituent une part importante de nos résultats, mais nous avons aussi étudié le problème de l'*implémentation abstraite*. Il s'avère que, même sans traitement d'exceptions, les formalismes existants d'implémentation abstraite n'offraient pas des critères totalement satisfaisants pour prouver qu'une implémentation abstraite est correcte. Ces critères étaient essentiellement *sémantiques*, fondés sur une description exhaustive de certaines algèbres relatives à l'implémentation. Or il est clair que des critères principalement fondés sur la *spécification* de l'implémentation sont nettement préférables sitôt que les exemples traités sont complexes, car une description complète des algèbres sous-jacentes est alors difficile.

Cette thèse est constituée de trois parties :

- La partie A développe un nouveau formalisme d'*implémentation abstraite* dans le cas "classique" (c'est-à-dire sans traitement d'exceptions). Ce formalisme a pour principal avantage d'offrir des critères fondés sur la spécification de l'implémentation pour décider de sa correction. Nous présentons aussi une notion de correction "forte" ; laquelle permet d'assurer la compatibilité des implémentations abstraites avec la *réutilisation*. La notion de réutilisation sera définie dans cette partie A ; elle implique en particulier que la *composition* de deux implémentations correctes reste correcte.

- La partie B expose le formalisme des *exception-algèbres*. Elle démontre que les résultats fondamentaux relatifs aux exception-algèbres sont les mêmes que ceux relatifs à la théorie des types abstraits algébriques sans traitement d'exceptions ; puis elle prouve que, grâce à ceux-ci, une approche *hiérarchique* similaire à celle sans traitement d'exceptions peut être définie.
  Par rapport aux formalismes existants de traitement d'exceptions, les exception-algèbres apportent une notion "d'étiquettes d'exception" qui modélise les messages d'erreur ; de plus, ce formalisme fournit tous les aspects requis, ou utiles, pour le traitement d'exceptions (spécification de structures bornées, récupération d'erreurs, traitement des valeurs erronées elles mêmes, propagation implicite des exceptions et des erreurs, existence de congruences minimales, existence d'algèbres initiales, spécifications hiérarchiques ...). Aucun des formalismes existants ne permettait de traiter *à la fois* tous ces aspects.

- La partie C traite l'*implémentation abstraite en présence d'exceptions*. On constate que le formalisme d'implémentation abstraite exposé en partie A est étendu aux exception-algèbres sans aucune difficulté. On peut alors spécifier l'implémentation de structures bornées au moyen d'autres structures bornées, et par exemple déterminer les rapports entre les bornes respectives de ces structures pour que l'implémentation soit correcte, ou encore spécifier simplement comment transcrire les messages d'erreur de la structure "implémentante" en messages d'erreur de la structure "implémentée".

Au début de chacune de ces trois parties, le lecteur trouvera un bref résumé des résultats qui y sont exposés, ainsi qu'une table des matières plus détaillée que celle fournie dans les pages suivantes.

Naturellement, l'implémentation abstraite en présence d'exceptions n'est qu'une extension possible du formalisme des exception-algèbres, parmi d'autres. De nombreuses autres extensions restent à étudier, par exemple l'étude de critères simples pour assurer le suffisante complétude ou la consistance hiérarchique, les rapports entre les exception-algèbres et la réécriture, les approches non initiales, la paramétrisation ...
Cette thèse concentre ses efforts sur l'implémentation abstraite parce que c'est un sujet qui semble déjà difficile sans traitement d'exceptions ; par conséquent, le fait que l'on puisse étendre les résultats obtenus sans traitement d'exceptions au formalisme des exception-algèbres nous semble prouver que ce formalisme d'exception-algèbres est "prometteur". D'autre part, un formalisme d'implémentation abstraite sans traitement d'exceptions ne permet pas, en général, de traiter l'implémentation de structures bornées ; or il est clair que ceci ne pouvait que nuire à la "crédibilité pratique" de l'implémentation abstraite.

PARTIE A :

IMPLEMENTATION ABSTRAITE
SANS
TRAITEMENT D'EXCEPTIONS

Résumé de cette partie A

Le but de cette partie est de développer un formalisme d'*implémentation abstraite* pour lequel les *preuves de corrections* soient simples à mettre en œuvre dans les exemples. Bien entendu, ce formalisme est orienté de telle sorte qu'il s'étendra sans efforts majeurs aux types abstraits algébriques avec traitement d'exceptions. Cette partie se scinde en deux chapitres : le premier développe une approche assez intuitive de l'implémentation abstraite en se référant aux travaux antérieurs qui ont inspiré notre formalisme, le second décrit notre formalisme.

Durant le premier chapitre, nous verrons que le problème de l'implémentation abstraite se pose de la manière suivante :

- on dispose de deux spécifications algébriques : celle de la structure déjà implémentée, et celle de la structure que l'on veut implémenter ; ces deux spécifications *décrivent* respectivement les propriétés connues de la structure déjà implémentée et les propriétés souhaitées de la structure à implémenter
- on cherche à spécifier algébriquement une implémentation "*constructive*" de la structure que l'on veut implémenter, au moyen de la structure déjà implémentée ; la question est de prouver dans quelle mesure cette *implémentation abstraite* est *correcte* par rapport à la spécification que l'on veut implémenter.

Nous verrons que deux méthodes principales sont connues :

- on peut chercher à *synthétiser* les sortes à implémenter au moyen des sortes déjà implémentées (*abstraction*)
- on peut chercher à donner une *représentation* de chaque terme fermé à implémenter, parmi les valeurs déjà implémentées.

Le formalisme que nous proposons dans le chapitre II est fondé sur l'usage *simultané* de ces deux notions (synthèse et représentation), ainsi que de la notion de *représentation de l'égalité*. Nous verrons que cette approche présente, entre autres, l'avantage de ramener le problème de la correction d'implémentations à des notions bien connues des types abstraits algébriques : la suffisante complétude et la consistance hiérarchique.

TABLE DES MATIERES

page 12 : **CHAPITRE I : Première approche**

TABLE DES MATIERES (SUITE)

page 40 : **CHAPITRE II : Notre formalisme d'implémentation abstraite**

Chapitre I :

Première approche

Ce chapitre contient 5 sections. La section 1 motive l'approche algébrique pour l'implémentation (*implémentation abstraite*) et décrit les buts poursuivis. La section 2 explique de manière intuitive les notions d'*abstraction* et de *représentation*. La section 3 développe le formalisme de [EKP 80, EKMP 80] en montrant ses corrélations avec la notion d'abstraction. La section 4 décrit brièvement [SW 82] qui propose un formalisme où la composition d'implémentations semble a priori prometteuse. Enfin la section 5 résume l'ensemble des méthodes exposées, et récapitule les points sur lesquels ces formalismes ne sont pas entièrement satisfaisants.

1. MOTIVATION

1.1. SITUONS LE PROBLEME

Durant le développement d'un programme, il est très important de manipuler des structures de données bien adaptées aux besoins particuliers du problème traité. Le programme est alors plus aisé à concevoir, et plus aisément lisible.
Mais les structures de données, ou le langage, dont on dispose au départ ne sont pas nécessairement assez puissants, ou bien adaptés, pour l'application envisagée. Une bonne méthode est alors d'écrire des primitives spécifiques au problème traité. On crée ainsi de nouvelles structures de données au moyen de celles dont on disposait au départ, puis on traite l'application envisagée au-dessus de ces nouvelles structures.
Une telle méthode facilite le développement du problème parce que l'on peut écrire le programme en se référant aux propriétés de ces nouvelles structures de "plus haut niveau". On laisse alors de côté la "traduction" des opérations de plus haut niveau en terme des opérations de départ.
Prenons un exemple simple. Supposons que l'application envisagée doive manipuler une structure de piles, dont on ne dispose pas au départ. On a alors intérêt à spécifier *d'abord* les opérations primitives des piles (*empty*, *push*, *pop* et *top*). On peut ensuite se baser sur les propriétés élémentaires bien connues des piles pour développer notre application. Le résultat est beaucoup plus lisible, rapide et simple à obtenir, car la manipulation effective des piles au moyen des structures de plus bas niveaux est uniquement traitée lors de la définition des opérations *empty*, *push*, *pop* et *top*.

Le passage entre les structures de données de plus haut niveau et celles de plus bas niveau est appelé l'*implémentation*. Le rôle de l'implémentation est donc la construction de nouvelles structures de

données au moyen de structures existantes. Elle doit en quelque sorte gérer la manipulation des opérations de plus haut niveau au moyen de celles de plus bas niveau ; et ceci en ne laissant visible "de l'extérieur" que les propriétés pertinentes de ces nouvelles opérations. L'utilisateur d'une structure de données implémentée pourra alors se reporter aux propriétés intéressantes des opérations qu'il manipule, sans se soucier des détails de leur implémentation.
Remarquons que la notion d'implémentation couvre une large étendue : cela va de la simple implémentation de quelques opérations au moyen d'opérations de plus bas niveau, jusqu'à l'implémentation d'un langage complet au moyen d'un autre langage (compilation, cf. [Gau 80]).

Les types abstraits algébriques permettent de *décrire de manière abstraite* des structures de données manipulées par l'utilisateur. De telles spécifications abstraites aident l'utilisateur parce qu'elles ignorent les détails d'implémentation de ces structures. Par exemple, lorsque l'on spécifie abstraitement la structure de pile, ce qui intéresse l'utilisateur sont des propriétés telles que :

l'accès à une pile fournit le dernier élément empilé.

Cette propriété s'exprime au moyen de l'axiome

$$top(push(x,X)) = x$$

De même, lorsqu'on ajoute un élément dans une pile (opération *push*), et qu'on la dépile ensuite (opération *pop*), on retrouve la pile de départ :

$$pop(push(x,X)) = X.$$

De tels axiomes sont *descriptifs* : ils ne reflètent généralement pas la manière *constructive* dont est effectivement implémentée la structure de pile. Souvent, on préfère implémenter les piles au moyen de tableaux : une pile est caractérisée par un tableau (qui contient les éléments de la pile) et un entier (qui délimite le haut de la pile dans le tableau) :

push (	x ,	x_0	x_1	~	...	)	=	x_0	x_1	x	...
pop (		x_0	x_1	x_2	...	)	=	x_0	x_1	x_2	...
top (		x_0	x_1	x_2	...	)	=	x_2			

Les types abstraits algébriques présentent l'avantage de fournir à l'utilisateur l'ensemble des "propriétés pertinentes" des piles. Mais le problème suivant se pose immédiatement : *comment vérifier si l'implémentation des piles au moyen des tableaux répond bien à la spécification algébrique des piles ?*
Le but de cette partie A est de traiter ce type de problèmes au niveau des types abstraits algébriques. L'idée est de se munir d'une *spécification algébrique* de l'implémentation elle-même ; on pourra alors utiliser les outils des types abstraits algébriques pour savoir quels sont les critères de correction et obtenir des méthodes générales de preuves de correction. Cette démarche est appelée l'*implémentation abstraite*.

Par exemple, l'implémentation des piles par les tableaux peut être spécifiée de la manière suivante :

$$\begin{aligned} empty &= \langle t,0\rangle \\ push(x,\langle t,i\rangle) &= \langle t[i]:=x,succ(i)\rangle \\ pop(\langle t,succ(i)\rangle &= \langle t,i\rangle \\ top(\langle t,succ(i)\rangle &= t[i] \end{aligned}$$

empty dénote la pile vide ; elle est implémentée par un tableau quelconque muni de l'entier 0, indiquant qu'aucun élément n'est empilé. Le tableau *t*, du couple $\langle t,i\rangle$, contient les éléments de la pile (de *0* à *i-1*), tandis que l'indice *i* pointe sur le rang du tableau où doit être placé le prochain élément de la pile. L'indice *i* est donc la hauteur de la pile. Les spécifications abstraites des piles et des

tableaux sont données en annexe 1. Pour simplifier, nous avons ici ignoré les cas d'exceptions tels que *top(empty)* .

Terminologie :
Dans toute la suite, pour différencier la spécification "pure" des piles (cf. *pop(push(x,X))=X*) de celle caractérisant l'implémentation des piles au moyen des tableaux et entiers (cf. *pop(<t,succ(i)>)=<t,i>*) , nous qualifierons la première de spécification *descriptive*, et la seconde de spécification *constructive*. Cette dénomination est justifiée par le fait que la première spécification décrit les propriétés abstraites caractérisant la structure de piles, alors que la seconde est censée modéliser un "algorithme d'implémentation" des piles au moyen de tableaux.

1.2. LES BUTS POURSUIVIS

1.2.1. PREUVE(S) DE CORRECTION

Notre propos est avant tout de trouver des méthodes pour prouver qu'une telle spécification *constructive* "simule" chaque opération de la spécification *descriptive* des piles. Nous employons ici le verbe "simuler" au sens français courant ; cette notion sera traduite algébriquement par la *correction d'une implémentation* (en fait, la "simulation" est légèrement différente de la notion de correction, elle est seulement incluse dans la correction d'une implémentation ; la notion de simulation a été systématiquement développée dans [SW 82]).

Le seul fait de définir cette *correction* n'est pas simple. En effet, on pourrait penser qu'il suffit de vérifier que les spécifications descriptives et constructives ont la même sémantique. Il n'en est rien. Par exemple, deux tableaux distincts munis de l'indice *i=0* représentent la même pile (*empty*) ; plus généralement deux tableaux distincts mais contenant les mêmes éléments entre *0* et *i-1* représentent la même pile. Ainsi, on constate que deux objets *constructifs* distincts peuvent représenter le même objet *descriptif*. Pour prouver la correction d'une implémentation abstraite il faudra donc d'abord caractériser les objets constructifs représentant le même objet descriptif. Nous verrons plus loin que ce problème n'est pas le seul soulevé par les preuves de correction d'implémentations.
Une autre approche pour définir la correction d'une implémentation peut être la suivante : même si les deux sémantiques sont différentes, l'implémentation sera correcte pourvu que "ce que l'utilisateur en observe" ne lui permette pas des les distinguer. Une telle définition de la correction d'une implémentation conduit à des sémantiques "terminales" ([Kam 80]). Malheureusement, les preuves de correction sont alors très difficiles à mettre en œuvre. De plus, la notion de correction dépend de l'étendue des "observateurs" que l'on donne à l'utilisateur ; on est alors en particulier confronté au problème suivant : que se passe-t'il si l'on enrichit une structure de données déjà implémentée ? (sachant qu'un enrichissement peut fort bien ajouter des opérations d'observation). Cette question entre dans le cadre plus général du problème de *réutilisation* d'implémentations déjà faites.

1.2.2. REUTILISATION D'IMPLEMENTATIONS

Supposons que les piles soient implémentées au moyen des tableaux, comme précédemment. Un utilisateur de cette structure de données l'incluera probablement dans divers programmes. Au niveau des types abstraits algébriques, ceci signifie que l'utilisateur va concevoir divers enrichissements au dessus de la spécification des piles (un enrichissement pouvant modéliser un "programme abstrait" au dessus de la structure de données implémentée). Mais l'utilisateur n'est nullement tenu de savoir comment est faite l'implémentation des piles. En d'autres termes, il connait la *spécification descriptive* des piles, mais il ne connait pas la *spécification constructive* de son implémentation. Il en résulte

que toute preuve concernant cet enrichissement sera établie en fonction de la spécification descriptive usuelle des piles, et non de celle utilisant les tableaux. Dès lors, rien ne prouve *a priori* que l'enrichissement en question, effectué au dessus de l'implémentation, fournisse les résultats attendus. Un cas particulier de ce problème est la *composition d'implémentations* (c'est-à-dire une implémentation faite au dessus de structures de données précédemment implémentées). Toutes les preuves de correction sont faites en regard de la spécification *descriptive* des structures précédemment implémentées. Là encore, *a priori*, rien ne prouve que la composition de ces implémentations soit correcte, même si chacune des implémentations est séparément correcte. Par exemple, le formalisme d'implémentation abstraite développé dans [EKP 80] et [EKMP 80] n'est pas compatible avec la composition d'implémentations : la composée de deux implémentations correctes peut ne pas l'être.

1.2.3. LE TRAITEMENT D'EXCEPTIONS

Le traitement d'exceptions est particulièrement crucial pour spécifier des implémentation abstraites réalistes. Prenons encore un exemple simple. Si l'on veut spécifier l'implémentation des *files* au moyen de tableaux. L'absence de traitement d'exceptions nous conduit à spécifier des files et des tableaux *non bornés* (appliquer l'opération *add_ to_* à une file pleine résultant sur un cas d'exception). On peut néanmoins donner une spécification constructive des files comme suit :

$$\begin{aligned} emptyqueue &= \langle t,i,i\rangle \\ add\ x\ to\ \langle t,i,j\rangle &= \langle t[j]:=x,i,succ(j)\rangle \\ remove(\langle t,i,j\rangle) &= \langle t,succ(i),j\rangle \\ first(\langle t,i,j\rangle) &= t[i] \end{aligned}$$

Une file est alors représentée par un triplet $\langle t,i,j\rangle$. Le tableau *t* contient les éléments de la file entre l'indice *i* et l'indice *j-1*. L'indice *i* pointe sur le premier élément de la file, alors que l'indice *j* pointe sur le rang du tableau où doit être placé le prochain élément de la file. L'opération *add_ to_* affecte l'élément ajouté à la place *j* du tableau et incrémente *j* de *1*, tandis que l'opération *remove_* incrémente simplement *i* de *1*.
Hormis les mauvais résultats obtenus avec *remove* et *first* lorsque la file est vide (on peut y remédier ici en ajoutant un prédicat *is-empty*, mais la spécification est alors incomplète, cf. [Gau 80]), on constate de plus une "fuite vers l'infini" de la file au fur et à mesure de l'application de l'opération *remove*. Pour remédier à ce problème, on est conduit à limiter la taille des files (en spécifiant des files bornées), afin de pouvoir en donner une implémentation "circulaire", dans des tableaux bornés.

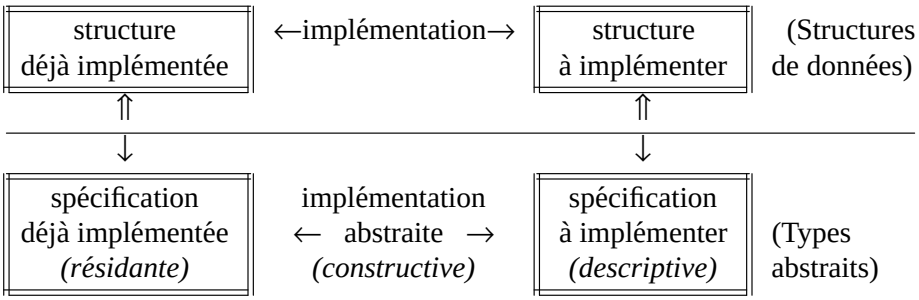
On constate donc sur cet exemple que le traitement d'exception est nécessaire si l'on veut spécifier des implémentations abstraites réalistes, aptes à gérer des structures bornées. C'est là l'un des buts de cette thèse : développer un formalisme d'implémentation abstraite avec traitement d'exceptions. La troisième partie de cette thèse (partie C) résoud cette question.
On verra que l'on peut en particulier spécifier une telle implémentation circulaire ainsi que l'implémentation des divers messages d'erreur s'y afférant, et prouver qu'elle est correcte (si et seulement si la taille maximale des tableaux est au moins égale à la longueur maximale des files, bien sûr).

2. ABSTRACTION ET REPRESENTATION

Suivant la discussion faite en section 1.1, nos données de départ sont les suivantes :

- □ on dispose de la spécification algébrique de la structure déjà implémentée, dite spécification *résidante* (par exemple les entiers naturels *NAT* et les tableaux *ARRAY*)
- □ on dispose de la spécification algébrique *descriptive* de la structure que l'on veut implémenter (par exemple les piles *STACK*)
- □ on veut spécifier algébriquement l'implémentation de *STACK* au moyen de *ARRAY* et *NAT* et lui donner une sémantique, ce que nous appellons une *implémentation abstraite*.

Figure 1 : *implémentation et types abstraits*



Deux méthodes se présentent naturellement à nous :

- → on peut partir des données déjà implémentées, et *synthétiser* les données à implémenter. Cette méthode est appelée l'*abstraction*, parce qu'elle permet "d'abstraire" des n-uplets de données résidantes (par exemple des couples <*tableau,entier*>) en des données à implémenter (*piles*).
- ← symétriquement, on peut partir des termes à implémenter et leur associer les valeurs qui les implémentent. Cette méthode est la *représentation* puisqu'elle fournit à chaque *terme* à implémenter (de sorte *STACK*) les objets qui le représentent (ici des couples <*tableau,entier*>).

2.1. ABSTRACTION

Pour *synthétiser* les sortes à implémenter (piles) au moyen des sortes résidantes (tableaux et entiers), on utilise une opération d'abstraction, notée A .

Exemple 1 :
Pour notre exemple, l'arité de l'opération d'abstraction est :

$$A : ARRAY\ NAT \rightarrow STACK .$$

Les axiomes constructifs spécifiant l'implémentation abstraite sont alors les suivants (on supposera qu'il s'agit de piles et tableaux d'entiers naturels afin de spécifier complètement l'opération *top*, cf. cinquième axiome) :

$$
\begin{array}{lrcl}
(1) & empty & = & A(t,0) \\
(2) & push(x,A(t,i)) & = & A(t[i]:=x,succ(i)) \\
(3) & pop(A(t,0)) & = & A(t,0) \\
(4) & pop(A(t,succ(i))) & = & A(t,i) \\
(5) & top(A(t,0)) & = & 0 \\
(6) & top(A(t,succ(i))) & = & t[i]
\end{array}
$$

Suivant les troisième et cinquième axiomes, nous posons que les cas d'exception *pop(empty)* et *top(empty)* sont respectivement égaux à *empty* et *0* ; la partie C traitera complètement les rapports entre *traitement d'exceptions* et *implémentation abstraite*.

Remarques 1 :

- □ Ces axiomes fournissent successivement l'implémentation de chacune des opérations sur les piles : (1) pour l'opération *empty*, (2) pour l'opération *push*, (3 & 4) pour l'opération *pop*, (5 & 6) pour l'opération *top*. Ainsi, toutes les opérations sont traitées si bien que l'on peut déterminer récursivement l'implémentation de tout terme de sorte *STACK*. Nous verrons que *traiter toutes les opérations* se définit au moyen d'un critère de correction appelé l'*opération-complétude* (introduit par [EKP 80]).

- □ Remarquons que le premier axiome laisse toute liberté quant au choix d'un tableau implémentant la pile vide. Le fait que la hauteur de la pile soit *0* suffit pour caractériser la pile vide, peu importe la teneur du tableau choisi.

La vision "abstraction" a un inconvénient majeur : elle synthétise trop d'objets dans les sortes à implémenter. Par exemple, l'élément de sorte *STACK* *A(create,4)* ne représente aucune pile (*create* est le tableau non initialisé). En effet, les 4 premières places du tableau *create* (de *create[0]* à *create[3]*) ne sont pas initialisées ; or pour représenter une pile de hauteur 4, il faut (au moins) que les 4 premiers éléments empilés soient définis.
Nous avions déjà souligné le fait que deux n-uplets distincts peuvent représenter le même objet à implémenter ; et que par conséquent il faut caractériser les n-uplets distincts qui implémentent le même objet afin de prouver la correction d'une implémentation. Nous venons de dégager ici un second obstacle aux preuves de correction d'implémentations : certains n-uplets ne représentent aucun objet à implémenter. Il faut donc aussi caractériser quels sont ceux qui sont "utiles", afin de prouver la correction d'une implémentation.
On pourrait penser que ce second obstacle est uniquement dû au traitement d'exceptions : si l'on convient que le tableau vide *create* contient uniformément *0*, la question ne se pose plus. En fait, il est de nombreux exemples sans cas exceptionnels où l'abstraction synthétise des n-uplets qui n'implémentent aucun objet. En voici un très simple : l'implémentation des ensembles (*SET*) par les chaînes (*STRING*).

Exemple 2 :
On suppose que *STRING* est muni des opérations "chaîne vide" notée λ, "ajoût d'un élément" *add(x,s)*, "retrait d'un élément" *remove(x,s)* qui enlève toute occurence de l'élément *x* dans la chaîne *s*, et *occurs(x,s)* qui retourne *True* si *x* est dans la chaîne *s*, *False* sinon. Les spécifications descriptives de *STRING* et *SET* sont données en annexe 1.
L'arité de l'opération d'abstraction est alors :

$$A : \quad STRING \rightarrow SET$$

et les axiomes d'implémentation sont simplement :

$$\begin{array}{rcrcl} & & \varnothing & = & A(\lambda) \\ occurs(x,s)=True & \Rightarrow & insert(x,A(s)) & = & A(s) \\ occurs(x,s)=false & \Rightarrow & insert(x,A(s)) & = & A(add(x,s)) \\ & & delete(x,A(s)) & = & A(remove(x,s)) \\ & & x \in A(s) & = & occurs(x,s) \end{array}$$

On constate que *insert(x,X)* est constructivement spécifié de telle sorte que si l'élément *x* est déjà dans la chaîne *s* implémentant *X*, on n'ajoute pas d'occurence inutile de *x* dans *s*. Il en resulte que les chaînes redondantes n'implémentent aucun ensemble. Ceci fournit donc un exemple où l'abstraction synthétise certains objets de sorte *SET* qui ne sont pas requis pour l'implémentation (ici les valeurs de la forme *A("une chaîne redondante")*).
De même, les axiomes d'implémentation spécifiés ici n'impliquent nullement la commutativité de l'insertion ensembliste : les objets synthétisés *A("ab")* et *A("ba")* sont *distincts*.

Cet exemple est donc révélateur des deux problèmes que nous avons soulignés :

- Lorsque l'on spécifie une implémentation, on crée certains objets "inutiles", qui ne sont pas atteints par les opérations à implémenter. Ceci impose de caractériser les objets utiles à l'implémentation avant de pouvoir prouver sa correction.

- Deux objets distincts pour la spécification constructive de l'implémentation peuvent implémenter le même objet. Ceci impose de caractériser quels sont les objets qui représentent le même élément à implémenter avant de pouvoir prouver la correction d'une implémentation.

En fait, ces deux problèmes sont les seuls obstacles rencontrés pour formaliser les implémentations abstraites. Ils soulèvent cependant de nombreuses difficultés, comme on le verra plus loin.

En ce qui concerne le premier obstacle, on peut tenter de caractériser les objets "utiles" synthétisés par l'opération d'abstraction. Cette caractérisation s'appelle *les invariants de représentation* (cf. [Gau 80]) :

- Seuls les couples $<t,i>$ tels que t est initialisé entre *0* et *i-1* sont atteints par opérations des piles.

- Seules les chaînes non redondantes sont atteintes par les opérations ensemblistes.

On peut aussi caractériser les n-uplets représentant le même objet à implémenter. Cette caractérisation s'appelle *la représentation de l'égalité* (cf. [Gau 80]). La représentation de l'égalité peut s'exprimer algébriquement :

- pour les piles :

$$A(t,0) = A(t',0)$$
$$A(t,i)=A(t',i) \ \& \ t[i]=t[j] \quad \Rightarrow \quad A(t,succ(i)) = A(t',succ(i))$$

  ce qui implique récursivement que les abstractions des couples *(t,i)* et *(t',j)* sont égales si et seulement si *i=j* , et *t* et *t'* contiennent les même éléments entre *0* et *i-1*.

- pour les ensembles :

$$A(add(x,add(y,s))) = A(add(y,add(x,s)))$$

  ce qui implique que les abstractions de deux chaînes contenant les mêmes éléments, mais en ordres éventuellement différents, sont égales.

Toutefois, les *invariants de représentation* et la *représentation de l'égalité* ne sont pas satisfaisants a

priori, car ils sont établis d'une manière totalement subjective. Encore faudrait-il une méthode générale pour les déduire de la spécification de l'implémentation, (ou tout au moins prouver qu'ils sont corrects ; c'est ce que nous ferons dans le chapitre II).

On peut raisonnablement penser que la *représentation* résoudra le problème des invariants de représentation. En effet, la représentation fournit, pour chaque terme à implémenter, des objets résidants qui l'implémentent ; en conséquence, elle ne devrait pas créer de nouveaux objets parmi les sortes à implémenter.

2.2. REPRESENTATION

Puisque la représentation associe à chaque terme fermé à implémenter les objets résidants qui l'implémentent, on la spécifie récursivement par rapport aux opérations à implémenter. Revenons à notre exemple de l'implémentation abstraite de *STACK* par *ARRAY×NAT*. La représentation associe un couple *<t,i>* à tout terme fermé de type *STACK* ; on spécifie la représentation au moyen d'une *opération de représentation* notée ρ :

$$\begin{aligned} \rho(empty) &= <t,0> \\ \rho(push(x, <t, i>)) &= <t[i]:=x,succ(i)> \\ \rho(pop(<t, 0>)) &= <t,0> \\ \rho(pop(<t, succ(i)>)) &= <t,i> \\ \rho(top(<t, 0>)) &= 0 \\ \rho(top(<t, succ(i)>)) &= t[i] \end{aligned}$$

Cette fois, nous rencontrons l'écueil suivant : comment donner une interprétation algébrique à de tels axiomes ?

- □ pour se placer dans un cadre algébrique, "< , >" doit être une opération, quelle est son arité ?
- □ quelle est l'arité de ρ ?

On voit que l'on applique (par exemple) *pop* à *<t,i>*. Par conséquent, la cible de l'opération < , > est *STACK*. D'autre part, la source de < , > est un couple tableau+entier, donc on obtient nécessairement la même arité que l'opération d'abstraction :

$$< , > : ARRAY\ NAT \rightarrow STACK$$

De la même manière, l'arité de ρ est nécessairement :

$$\rho : STACK \rightarrow STACK \quad (^1).$$

En fait, vu sous cet angle, l'opération ρ devient inutile (c'est l'identité), et à un renommage près, < , > n'est autre que l'opération d'abstraction (comme on le constate en comparant les axiomes relatifs à l'abstraction avec ceux relatifs à la représentation).

Une autre tentative de définition de la représentation pourrait être de considérer *deux* opérations de représentation :

$$\rho_1 : STACK \rightarrow ARRAY$$
$$\rho_2 : STACK \rightarrow NAT$$

ρ_1 (resp. ρ_2) ferait alors correspondre à chaque terme de sorte pile le tableau (resp. l'entier) qui le

(1) ou $\rho : NAT \rightarrow NAT$, a cause des deux derniers axiomes

représente.
Mais dans ce cas, $\rho_1(empty)$ doit être spécifié comme un tableau particulier (par exemple $\rho_1(empty) = create$). En effet, si l'on veut spécifier que $\rho_1(empty)$ peut être égal à n'importe quel tableau, avec $\rho_2(empty) = 0$ comme dans le cas de l'abstraction, on est conduit à écrire un axiome de la forme : $\rho_1(empty) = t$. Or ceci a pour effet de rendre tous les tableaux égaux entre eux. Il en résulte que cette tentative de scinder la représentation en plusieurs opérations n'est pas suffisamment puissante dans la plupart des cas, et risque d'induire des inconsistances.
Il serait utile de définir une opération ρ_1 faisant correspondre à chaque terme à implémenter, non pas un tableau, mais un *ensemble* de tableaux. Un formalisme introduisant le non déterminisme dans les types abstraits algébriques (par exemple [Hes 85]) pourrait être utilisé, mais les résultats obtenus actuellement dans ce domaine ne sont pas encore assez puissants pour offrir des critères simples de correction d'une implémentation.

On voit donc que formaliser la notion de représentation n'est pas simple. Il serait pourtant très utile de pouvoir donner une formalisation rigoureuse de la représentation, car elle ne présente pas les inconvénients de l'abstraction (elle résoud le problème des invariants de représentation). En fait, on aimerait une opération de représentation dont la cible soit un *produit*. Nous formaliserons ceci dans le chapitre II, et nous verrons qu'il est préférable d'utiliser *en même temps* les opérations d'abstraction et de représentation. Pour commencer, nous allons développer le formalisme de [EKP 80] et [EKMP 80] (fondé sur l'abstraction), car il contient les premiers éléments d'une réelle formalisation algébrique de l'implémentation abstraite.

3. IMPLEMENTATION SELON [EKMP 80]

Le but de cette section n'est pas de decrire le formalisme de [EKP 80, EKMP 80] dans le détail (une analyse complète en est déjà faite dans [Ber 84]), mais seulement de montrer comment les problèmes liés aux *invariants de représentation* et à la *représentation de l'égalité* y sont traités de manière algébrique. De plus, nous décrivons le formalisme de [EKP 80, EKMP 80] dans un esprit différent des articles originaux :

- Nous allons le développer sous une forme systématiquement fonctorielle (mais en montrant l'aspect intuitif) ; car cette forme nous permettra ultérieurement de mieux comprendre comment étendre la notion d'implémentation abstraite aux types abstraits algébriques avec traitement d'exceptions.

- Nous allons inclure d'emblée la notion de composante cachée, car nous pensons que la composante cachée est en fait un outil indispensable pour spécifier des implémentations *structurées* (contrairement à l'interprétation de [EKMP 80] qui place la notion de structuration uniquement au niveau de la *composition* des implémentations). Voir la section 3.2 plus loin.

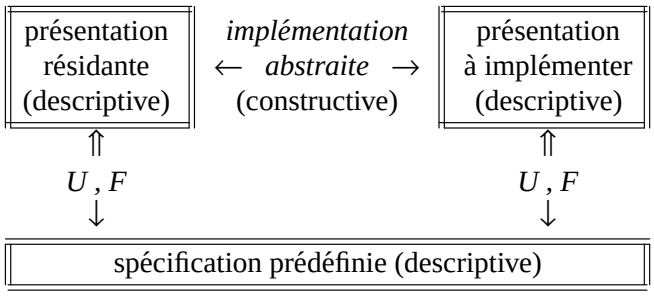
- De plus, nous verrons que ce formalisme est très lié à la notion d'abstraction, et nous choisirons des notations cohérentes avec l'abstraction pour mieux faire ressortir ce fait.

L'idée principale introduite par [EKP 80] et [EKMP 80] est la distinction entre les niveaux *syntaxique* et *sémantique* d'une implémentation. Cette distinction offre une base solide pour dégager les critères de *correction* d'implémentation ; c'est pourquoi nous développons brièvement ce formalisme ici.

3.1. LES DONNEES DU PROBLEME

Nous avons vu (figure 1) que l'on dispose de deux spécifications algébriques "descriptives" : celle déjà implémentée (résidante) et celle que l'on veut implémenter. En fait, il peut y avoir une partie commune entre ces deux spécifications. Par exemple, les *éléments* mémorisés dans les tableaux sont ceux que l'on veut placer dans les piles. Cela signifie que les spécifications résidantes (tableaux), et à implémenter (piles) sont toutes deux des *présentations* au-dessus de la spécification *prédéfinie ELEM*. On peut schématiser comme suit la situation de départ d'une implémentation abstraite :

Figure 2 : *situation de départ*



où *U* et *F* sont les habituels foncteurs d'oubli et de synthèse.

Notation 1 :
Dans toute la suite du formalisme [EKMP 80], on notera :

- □ $\mathbf{P}=<\mathbf{S_P}, \mathbf{\Sigma_P}, \mathbf{E_P}>$ la spécification prédéfinie (par exemple *ELEM*)
- □ $\mathbf{PRES_0}=<\mathbf{S_0}, \mathbf{\Sigma_0}, \mathbf{E_0}>$ la présentation (au-dessus de **P**) déjà implémentée (par exemple *ARRAY* et *NAT*). $\mathbf{PRES_0}$ est la présentation *résidante*.
- □ $\mathbf{PRES_1}=<\mathbf{S_1}, \mathbf{\Sigma_1}, \mathbf{E_1}>$ la présentation descriptive (au-dessus de **P**) que l'on veut implémenter (par exemple *STACK*).

Ces ensembles de sortes (**S**) et opérations (**Σ**) sont supposés disjoints deux à deux, et le formalisme [EKMP 80] ne considère que des ensembles d'équations non conditionnelles (**E**). On suppose de plus que $\mathbf{PRES_1}$ et $\mathbf{PRES_0}$ *sont toutes deux persistantes* par rapport à la spécification prédéfinie **P**.

Remarque 2 :
Le formalisme [EKMP 80] vise à *remplacer* la structure de données résidante par la structure à implémenter spécifiée algébriquement. Ainsi, partant d'une structure (tableaux+entiers+éléments), on obtient une structure (piles+éléments) ; la structure de tableau n'apparaît plus explicitement dans la structure finalement implémentée. C'est pour cette raison que l'on doit distinguer la partie prédéfinie *ELEM* car, contrairement aux tableaux, elle apparaît dans les deux spécifications.

3.2. COMPOSANTE CACHEE ET STRUCTURATION

Comme nous l'avons déjà souligné au début de ce chapitre, les opérations et spécifications internes à l'implémentation abstraite ne seront pas "visibles" de l'exterieur. Ce que l'utilisateur d'une implémentation connaît c'est seulement la spécification *descriptive* de la structure à implémenter, c'est à dire $\mathbf{P}+\mathbf{PRES_1}$, et non la spécification *constructive* de l'implémentation.
On peut exploiter ce fait en faisant usage d'*opérations cachées* pour spécifier l'implémentation : elles

ne seront pas visibles pour l'utilisateur, et ne pourront être utilisées que "localement" dans l'implémentation abstraite. Ainsi, lors de la conception d'une implémentation abstraite, on n'hésitera pas à ajouter autant d'opérations cachées qu'il peut sembler utile pour faciliter la spécification de l'implémentation ; ces opérations cachées n'encombreront pas la spécification descriptive que manipule l'utilisateur.

Revenons à l'exemple de l'implémentation des ensembles par les chaînes (exemple 2, section 2.1). Nous avons supposé que la spécification *STRING* contenait les opérations *occurs* et *remove* , ce qui nous a permis une spécification très simple de l'implémentation en utilisant ces opérations.
Supposons que la spécification *STRING* ne soit pas munie de ces opérations ; une spécification de l'implémentation abstraite sans utiliser *occurs* et *remove* serait alors plus complexe et moins lisible. Une "bonne methode" est de spécifier ces opérations avant de spécifier l'implémentation elle même : *occurs* et *remove* sont alors des *opérations cachées* internes à l'implémentation qui facilitent son écriture. On obtient ainsi une spécification plus structurée, et donc plus lisible, de l'implémentation abstraite.

Il se peut même que, dans des exemples plus complexes, les opérations cachées nécessitent elles mêmes l'intervention d'autres opérations cachées pour être spécifiées (qui peuvent elles aussi en faire intervenir d'autres ...etc..). De ce fait, une *composante cachée* offre une méthode de structuration *interne* de l'implémentation abstraite très puissante, et même indispensable. Ainsi, la composante cachée facilite la spécification constructive de l'implémentation, sans surcharger la spécification descriptive finalement implémentée ($\mathbf{P}+\mathbf{PRES_1}$).

Par contre, il en résulte que ces opérations cachées ne peuvent pas être *réutilisées* hors de l'implémentation. C'est pourquoi une composante cachée est un outil de structuration *locale* de l'implémentation. Elle n'offre aucune potentialité de réutilisation. La structuration d'implémentations par le biais de *réutilisations* d'implémentations déjà faites est modélisée par la *composition d'implémentations abstraites*. Pour implémenter une structure de "haut niveau" au moyen d'une structure de "bas niveau", on peut commencer par implémenter une structure intermédiaire au moyen de la structure de plus bas niveau, puis implémenter la structure de plus haut niveau au moyen de cette structure intermédiaire ; ce qui revient à *composer* ces deux implémentations. Cette méthode relève de la notion de *réutilisation*, car on réutilise l'implémentation de la structure intermédiaire pour faire celle de plus haut niveau.
Cette méthode de structuration par *composition* n'est pas locale à une implémentation. La composante cachée d'une implémentation et la composition d'implémentations sont deux méthodes complémentaires de structuration d'implémentations abstraites. Contrairement à [EKP 80, EKMP 80, SW 82], nous pensons que la composition d'implémentations à elle seule n'est pas un outil de structuration suffisant ; la composante cachée est un outil à part entière, indispensable pour structurer localement une implémentation et la rendre plus facilement lisible.

Cette parenthèse étant close, développons le formalisme [EKMP 80]. Il se scinde en deux niveaux : le niveau *syntaxique* et le niveau *sémantique*.

3.3. LE NIVEAU SYNTAXIQUE

Le niveau syntaxique (avec composante cachée) est défini comme suit :

Définition 1 :
Une *implémentation syntaxique* est un triplet $\mathbf{IMPL}=(\mathbf{\Sigma_{ABS}},\mathbf{C},\mathbf{E_{OP}})$ tel que :

- □ $\mathbf{\Sigma_{ABS}}$ est l'ensemble des *opérations d'abstraction* (appelées "opérations de synthèse" dans [EKMP 80]) de cible dans $\mathbf{S_1 \cup S_C}$

- □ **C** est la *composante cachée* de l'implémentation ; $\mathbf{C}=<\mathbf{S_C}, \mathbf{\Sigma_C}, \mathbf{E_C}>$ (sortes, opérations et équations cachées)

- □ $\mathbf{E_{OP}}$ est l'ensemble des *équations d'implémentation des opérations* ($\mathbf{E_{OP}}$ spécifie récursivement l'implémentation "constructive" de toutes les opérations à implémenter).

On impose de plus les conditions syntaxiques suivantes :

$$\mathbf{SORTimpl} = \mathbf{P} + \mathbf{PRES_0} + <\mathbf{S_1 \cup S_C}, \mathbf{\Sigma_{ABS}}, \emptyset>$$

et

$$\mathbf{OPimpl} = (\mathbf{SORTimpl} + <\emptyset, \mathbf{\Sigma_C}, \mathbf{E_C}>) + <\emptyset, \mathbf{\Sigma_1}, \mathbf{E_{OP}}>$$

doivent être des spécifications. Ces conditions se traduisent comme suit :

- □ **SORTimpl** est une présentation au-dessus de la spécification résidante. Elle ajoute les sortes cachées et les sortes à implémenter, mais pas les opérations cachées ni les opérations à implémenter. Elle ajoute aussi les opérations d'abstraction, qui peuvent donc prendre leur arité aussi bien dans les sortes résidantes que les sortes cachées ou à implémenter.

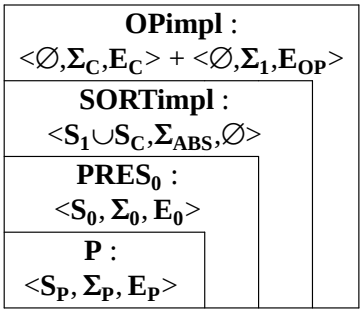
- □ $<\emptyset, \mathbf{\Sigma_C}, \mathbf{E_C}>$ est une présentation au-dessus de **SORTimpl**. C'est la partie majeure de la composante cachée ; elle définie le rôle exact des opérations cachées.

- □ Enfin $<\emptyset, \mathbf{\Sigma_1}, \mathbf{E_{OP}}>$ est la spécification proprement dite de l'implémentation. Elle ajoute les opérations que l'on veut implémenter, et décrit leur implémentation via les équations de $\mathbf{E_{OP}}$ qui peuvent utiliser toutes les opérations précédentes (entre autres les opérations cachées).

Tous ces ensembles de sortes ou opérations sont supposés deux à deux disjoints.

[EKMP 80] appelle *implémentation standard* une implémentation sans composante cachée, et ne développe la notion d'implémentation avec composante cachée que comme une extension possible de ce formalisme. Nous préférons développer directement la version avec composante cachée, cf. section 3.2 précédente.

Figure 3 : *syntaxe de* **IMPL**



On voit que la syntaxe de l'implémentation abstraite contient toutes les parties des spécifications invoquées, sauf les équations descriptives de $\mathbf{PRES_1}$ ($\mathbf{E_1}$).

Remarques 3 :

- □ Dans la plupart des exemples, les opérations de $\mathbf{\Sigma_{ABS}}$ seront de cible dans les nouvelles sortes à implémenter. Dès lors, la vision intuitive de $\mathbf{\Sigma_{ABS}}$ correspond bien à la notion d'*opération*

d'abstraction que l'on a brièvement développée précédemment : simplement, du fait que l'on peut avoir plusieurs sortes à implémenter ($\mathbf{S_1}$ n'est pas toujours réduit à un singleton), on aura aussi plusieurs opérations d'abstraction (une pour chaque sorte à implémenter).

□ [EKMP 80] note **Σ**SORT au lieu de $\mathbf{\Sigma_{ABS}}$, et nomme *opérations de synthèse* les opérations d'abstraction (sans même faire référence à la notion classique d'abstraction). Ceci provient de la remarque intuitive suivante. Une opération d'abstraction (telle que celle des piles au moyen des tableaux et des entiers) *synthétise* la sorte à implémenter (piles) en la "remplissant" de n-uplets (<*tableau*,*entier*>). Par conséquent la composante **SORTimpl** "implémente les sortes nouvelles" au moyen des opérations d'abstraction. Il est à noter que dans le cas particulier où une opération d'abstraction est unaire (cf. l'exemple de l'implémentation des ensembles au moyen des chaînes), elle n'est autre qu'une opération de copie, au même titre qu'un produit monocomposante est une copie.

□ $\mathbf{E_{OP}}$ est exactement l'ensemble d'équations constructives que l'on a déjà décrit pour l'abstraction. Il indique récursivement la façon dont sont implémentées chacune des opérations de $\mathbf{\Sigma_1}$. Ainsi la spécification **OPimpl** indique *d'abord* comment est gérée la partie cachée de l'implémentation, puis fournit grâce à $\mathbf{E_{OP}}$ la partie majeure de l'implémentation, à savoir le lien entre les opérations à implémenter et celles qui le sont déjà.

Exemple 3 :
Revenant à l'exemple de l'implémentation de *SET* par *STRING*, l'ensemble des opérations d'abstraction $\mathbf{\Sigma_{ABS}}$ est réduit au singleton

$$\{ \quad A : \ STRING \ \rightarrow \ SET \quad \} .$$

En ce qui concerne la composante cachée, $\mathbf{S_C}$ est vide, $\mathbf{\Sigma_C}$ contient les opérations

$$occurs : \ ELEM\ STRING \rightarrow BOOL$$
$$remove : \ ELEM\ STRING \rightarrow STRING$$

et $\mathbf{E_C}$ contient les équations suivantes :

$$\begin{aligned} occurs(x, \lambda) &= False \\ occurs(x,add(y,s)) &= if\ eq(x,y)\ then\ true\ else\ occurs(x,s) \\ remove(x, \lambda) &= \lambda \\ remove(x,add(y,s)) &= if\ eq(x,y)\ then\ remove(x,s)\ else\ add(y,remove(x,s)) \end{aligned}$$

Enfin $\mathbf{E_{OP}}$ est l'ensemble d'équations constructives déjà spécifié dans l'exemple 2, section 2.1.

Comme on le voit, il n'est pas difficile de traduire dans le formalisme [EKP 80, EKMP 80] tous les exemples d'implémentation utilisant l'abstraction.

3.4. LE NIVEAU SEMANTIQUE

Le niveau sémantique de [EKMP 80] se partage en 3 foncteurs : *synthèse*, *restriction* et *identification*.

$$\begin{array}{ccccccccc} \text{Alg}(\mathbf{PRES_0}) & -synth\rightarrow & \text{Alg}(\mathbf{OPimpl}) & -restr\rightarrow & \text{Gen}(\mathbf{\Sigma_P \cup \Sigma_1}) & -ident\rightarrow & \text{Gen}(\mathbf{PRES_1}) \\ T_{\mathbf{PRES_0}} & \vdash synth\rightarrow & T_{\mathbf{OPimpl}} & \vdash restr\rightarrow & \text{REP}_{\mathbf{IMPL}} & \vdash ident\rightarrow & \text{SEM}_{\mathbf{IMPL}} \end{array}$$

3.4.1. LE FONCTEUR DE SYNTHESE

SORTimpl+**OPimpl** est une présentation au-dessus de $\mathbf{P}+\mathbf{PRES_0}$. Cette présentation contient en fait ce que nous avons appelé la *spécification constructive* de l'implémentation. Comme pour toute présentation, l'effet sémantique en est décrit au moyen du foncteur de *synthèse* qui lui est associé (adjoint à gauche au foncteur d'oubli). Il en résulte que $T_{\mathbf{OPimpl}}$ modélise la structure de données effectivement construite après que l'implémentation soit faite.

Exemples 4 :
Dans l'implémentation des piles par tableaux+entiers (exemple 1), $T_{\mathbf{OPimpl}}$ contient dans la sorte *STACK* tous les couples *(tableau,entier)*. De plus on sait quelle est l'action de chacune des opérations *push*, *pop* ... sur ces couples.
Dans l'exemple de *SET* par *STRING*, $T_{\mathbf{OPimpl}}$ contient dans la sorte *SET* une copie des chaînes d'éléments ; et on connaît l'action des opérations ensemblistes sur ces chaînes.

3.4.2. LE FONCTEUR DE RESTRICTION

On sait que l'abstraction synthétise trop d'éléments dans les sortes à implémenter (cf. *A(create,4)* pour les piles, et les chaînes redondantes, *A*("$\alpha\alpha$"), pour les ensembles).
Ces objets synthétisés en trop sont "disponibles au sein de l'implémentation", *mais* puisque l'utilisateur d'une structure de données implémentée ne *doit pas* faire usage de l'opération d'abstraction, il ne les rencontrera jamais. Les seules opérations laissées visibles pour l'utilisateur sont celles de $\mathbf{\Sigma_1 \cup \Sigma_P}$.

Cette "visibilité restreinte" donnée à l'utilisateur est traduite via le foncteur de restriction. L'action de ce foncteur sur toute algèbre $A \in \mathrm{Alg}(\mathbf{OPimpl})$ consiste simplement en l'oubli des sortes de $\mathbf{S_0}$ puis en l'élagage des éléments de A qui ne sont pas engendrés par les opérations de $\mathbf{\Sigma_1 \cup \Sigma_P}$. Ainsi, $\mathrm{REP}_{\mathbf{IMPL}}$ est égal à l'ensemble des objets de $T_{\mathbf{OPimpl}}$ qui sont atteints par les opérations à implémenter. $\mathrm{REP}_{\mathbf{IMPL}}$ contient donc toutes les données (synthétisées par l'implémentation) qui *représentent* des objets à implémenter.

De manière précise : la spécification **OPimpl** a pour signature $< \mathbf{S_P \cup S_0 \cup S_C \cup S_1}\ ,\ \mathbf{\Sigma_P \cup \Sigma_0 \cup \Sigma_{ABS} \cup \Sigma_C \cup \Sigma_1} >$; elle contient donc la signature de $\mathbf{PRES_1}$, $<\mathbf{S_P \cup S_1}\ ,\ \mathbf{\Sigma_P \cup \Sigma_1}>$. Le foncteur de restriction commence donc par faire un *oubli* de **OPimpl** sur la signature $<\mathbf{S_P \cup S_1}\ ,\ \mathbf{\Sigma_P \cup \Sigma_1}>$; puis il fait un *élagage*. Cet élagage est le foncteur défini comme suit :

$$\begin{array}{ccc} \mathrm{Alg}(<\mathbf{S_P \cup S_1}, \mathbf{\Sigma_P \cup \Sigma_1}>) & -\textit{élagage}\rightarrow & \mathrm{Gen}(<\mathbf{S_P \cup S_1}, \mathbf{\Sigma_P \cup \Sigma_1}>) \\ X & \vdash\textit{élagage}\rightarrow & \textit{init}_X(T_{\mathbf{\Sigma_P \cup \Sigma_1}}) \end{array}$$

où $\textit{init}_X$ est le morphisme initial $T_{\mathbf{\Sigma_P \cup \Sigma_1}} \rightarrow X$ (cf. [Ber 86]).

Exemples 5 :
Dans l'exemple des piles, $\mathrm{REP}_{\mathbf{IMPL}}$ contient dans la sorte *STACK* les couples *A(t,i)* tels que *t[k]* est initialisé pour tout $0 \le k < i$.
De la même façon, dans l'exemple des ensembles, $\mathrm{REP}_{\mathbf{IMPL}}$ contient l'abstraction de la chaîne vide λ (représentant *empty*) , et les chaînes non redondantes (par insertions). Par contre, il ne contient aucune chaîne redondante : les chaînes redondantes ne sont pas atteintes par les opérations ensemblistes.

On constate donc que le foncteur de *restriction* résoud sémantiquement le problème des *invariants de représentation*.

3.4.3. LE FONCTEUR D'IDENTIFICATION

Au point où nous en sommes, $REP_{\mathbf{IMPL}}$ modélise exactement ce que fait l'implémentation : les données gérées par l'implémentation sont modélisées par $T_{\mathbf{OPimpl}}$, et $REP_{\mathbf{IMPL}}$ est la *partie utile* de $T_{\mathbf{OPimpl}}$. Toutefois la question de la *représentation de l'égalité* n'est pas encore traitée : dans l'exemple des ensembles, les termes *insert(a,insert(b,empty))* et *insert(b,insert(a,empty))* dénotent le même ensemble {*a*,*b*} mais ne sont pas implémentés par les mêmes chaînes ("*ab*" et "*ba*"). Les deux objets *A("ab")* et *A("ba")* sont *distincts* dans $REP_{\mathbf{IMPL}}$; mais l'utilisateur *ne s'en apercevra pas*. Ceci est dû au fait qu'aucune opération visible depuis l'utilisateur ne permettra de les différencier. Le rôle du foncteur d'*identification* est simplement d'amalgamer les objets distincts de $REP_{\mathbf{IMPL}}$ qui représentent le même élément ; autrement dit, le foncteur d'identification résoud sémantiquement le problème de la *représentation de l'égalité*.

Nous avons déjà évoqué ce phénomène (section 1.2.1) : c'est une question de choix d'*observabilité*. Si l'on se contente des opérations visibles depuis l'utilisateur lorsque l'implémentation vient d'être faite, $REP_{\mathbf{IMPL}}$ caractérise bien l'implémentation abstraite (observée par rapport aux sortes prédéfinies de $\mathbf{S_P}$, cf. [Kam 80]). Mais si l'on vise une notion d'implémentation abstraite qui soit compatible avec d'éventuels enrichissements ultérieurs, de tels enrichissements risquent fort d'ajouter des observateurs. On est alors contraint d'inclure une caractérisation de la *représentation de l'égalité* dans $REP_{\mathbf{IMPL}}$.
[EKMP 80] n'a pas fait ce choix ; la notion d'implémentation abstraite développée ici n'est donc pas compatible avec les enrichissements.

Pour définir la *correction* d'une implémentation, il faut comparer la sémantique de l'implémentation avec la sémantique de la spécification descriptive que l'on veut implémenter. Il n'est pas simple de comparer $REP_{\mathbf{IMPL}}$ avec l'algèbre initiale de $\mathbf{PRES_1}$. Il faudrait que $REP_{\mathbf{IMPL}}$ soit une $\mathbf{PRES_1}$-algèbre, ce qui n'est pas le cas (car **OPimpl** ne contient pas $\mathbf{E_1}$). Qu'à cela ne tienne, on peut artificiellement transformer $REP_{\mathbf{IMPL}}$ en une $\mathbf{PRES_1}$-algèbre : il suffit de quotienter $REP_{\mathbf{IMPL}}$ par les équationss de $\mathbf{E_1}$; c'est là le rôle du foncteur d'*identification*.
Le foncteur d'identification est donc défini comme suit :

$$\begin{array}{ccc} \mathrm{Gen}(<\mathbf{S_P}\cup\mathbf{S_1}, \mathbf{\Sigma_P}\cup\mathbf{\Sigma_1}>) & -\mathit{identification}\rightarrow & \mathrm{Gen}(\mathbf{PRES_1}) \\ X & \vdash\mathit{identification}\rightarrow & X/_{\equiv \mathbf{E_1}} \end{array}$$

où $\equiv \mathbf{E_1}$ est la congruence minimale sur X associée à $\mathbf{E_1}$. Le foncteur d'identification est l'adjoint à gauche du foncteur d'oubli entre $\mathbf{PRES_1}$ et sa signature. $SEM_{\mathbf{IMPL}} = \mathit{identification}(REP_{\mathbf{IMPL}})$ est appelé le *resultat sémantique* de l'implémentation abstraite.

Remarque 4 :
Le foncteur d'identification utilise donc les équationss descriptives de $\mathbf{E_1}$, afin de quotienter $REP_{\mathbf{IMPL}}$ en $SEM_{\mathbf{IMPL}} = (REP_{\mathbf{IMPL}})/_{\equiv \mathbf{E_1}}$. Il faut bien voir que dans le formalisme [EKMP 80], ce foncteur n'est qu'un outil théorique pour faciliter les preuves de correction d'implémentation ; il ne modélise rien de concret. En effet, le but même d'une implémentation est de ne pas utiliser les équations descriptives de $\mathbf{PRES_1}$ (sinon, autant faire directement de la réécriture sur $\mathbf{E_1}$), on veut les *remplacer* par les equations constructives de l'implémentation ($\mathbf{E_{OP}}$). C'est $REP_{\mathbf{IMPL}}$ qui modélise la structure de données obtenue après implémentation ; le résultat sémantique $SEM_{\mathbf{IMPL}}$ n'est qu'un outil théorique permettant des comparaisons avec la structure descriptive $T_{\mathbf{PRES_1}}$.

3.5. LES PREUVES DE CORRECTION

Pour qu'une implémentation abstraite soit correcte, il est clair que nous voulons d'une part que l'implémentation abstraite détermine (récursivement) l'implémentation de tout terme fermé à implémenter (cf. remarque 1) ; nous voulons d'autre part que cette implémentation "simule" la spécification descriptive à implémenter. Ces deux propriétés s'appellent respectivement l'*opération complétude* (ou plus simplement l'*op-complétude*), et la *consistance* de l'implémentation abstraite.

3.5.1. L'OPERATION-COMPLETUDE

L'opération complétude signifie que l'implémentation abstraite permet d'associer à tout terme fermé à implémenter (c'est-à-dire tout terme fermé sur la signature $\mathbf{\Sigma_P \cup \Sigma_1}$), un terme synthétisé par les opérations d'abstraction ($\mathbf{\Sigma_{ABS}}$) et les opérations résidantes ($\mathbf{\Sigma_P \cup \Sigma_0}$). Rappelons que la signature de **SORTimpl** est justement ($\mathbf{\Sigma_{ABS} \cup \Sigma_P \cup \Sigma_0}$), et que **OPimpl** spécifie l'implémentation abstraite. Il en résulte que l'op-complétude s'exprime comme suit :

Définition 2 :
L'implémentation abstraite **IMPL** est dite *op-complète* si et seulement si pour tout $\mathbf{\Sigma_P \cup \Sigma_1}$-terme fermé t ($\in T_{\mathbf{\Sigma_P \cup \Sigma_1}}$), il existe un terme $\alpha \in T_{\mathbf{\Sigma(SORTimpl)}}$ tel que t est congru à α modulo **OPimpl**. Ce qui s'écrit :

$$\forall\, t \in T_{\mathbf{\Sigma_P \cup \Sigma_1}}, \exists\, \alpha \in T_{\mathbf{\Sigma(SORTimpl)}} \ \textit{tel que}\ \ t = \alpha \ \ \textit{dans}\ T_{\mathbf{OPimpl}}$$

L'opération-complétude n'est pas difficile à vérifier sur les exemples. En effet, elle concerne l'ensemble des $\mathbf{\Sigma_P \cup \Sigma_1}$-termes fermés ; par conséquent, il suffit de raisonner par induction structurelle vis à vis des opérations de $\mathbf{\Sigma_P \cup \Sigma_1}$. C'est d'ailleurs pour cette raison que cette propriété s'appelle l'*opération-complétude* : elle signifie que chaque opération à implémenter est complètement spécifiée.

Exemple 6 :
Notre implémentation abstraite de *SET* par *STRING* (exemples 2 & 3) est op-complète :

- □ *empty* est congru à $A(\lambda)$ modulo **OPimpl**.
- □ En ce qui concerne l'opération *insert* ; si x et X sont congrus à des $\mathbf{\Sigma}$(**SORTimpl**)-termes, alors x est nécessairement congru à un $\mathbf{\Sigma_P}$-terme (disons p), et X est nécessairement de la forme *A(s)*. Ainsi, *insert(x,X)* est congru à *insert(p,A(s))*, lui même congru à *A(if occurs(p,s) then s else add(p,s))* ; et notre conclusion résulte du fait que l'opération cachée *occurs* est complètement définie dans **C**.
- □ Le même raisonnement vaut pour les opérations "*delete*" et "$\in$".

On dispose de plus de la proposition suivante, qui donne une condition *suffisante* pour l'op-complétude :

Proposition 1 :
Pour que l'implémentation abstraite **IMPL** soit op-complète, il *suffit* que la présentation **OPimpl** au-dessus de **SORTimpl** soit suffisamment complète.

Cette condition *n'est pas nécessaire* (cf. [Ber 84]). Il peut y avoir des exemples qui soient opération-complets, mais où cette proposition ne s'applique pas. Ce fait n'est pas très gênant, la suffisante complétude se vérifiant généralement par induction structurelle. Cette proposition est cependant utile chaque fois que l'on peut utiliser un outil syntaxique tel que les présentation gracieuses ([Bid 82]).

Nous ne donnerons pas ici la démonstration de la proposition 1, on peut la trouver dans [Ber 84] ; un résultat similaire est de plus démontré dans le chapitre II, dans le cadre de notre formalisme d'implémentation abstraite.

3.5.2. LA CONSISTANCE DE L'IMPLEMENTATION

Nous voulons maintenant traduire algébriquement le fait que l'implémentation "simule" la spécification descriptive à implémenter. Nous avons noté (section 3.4.3) que [EKMP 80] traduit cette notion uniquement par rapport à la spécification prédéfinie. Nous allons donc traduire le fait que, pour tout $\mathbf{\Sigma_P \cup \Sigma_1}$-terme de sorte prédéfinie, la spécification constructive de l'implémentation lui donne la même valeur prédéfinie que le ferait la spécification descriptive que l'on veut implémenter :

- □ Remarquons que le critère d'op-complétude nous assure déjà que tout $\mathbf{\Sigma_P \cup \Sigma_1}$-terme, *t*, de sorte prédéfinie, est congru à un $\mathbf{\Sigma_P}$-terme modulo l'implémentation. Soit *p* ce terme. On a donc $t \equiv p$ modulo $\mathrm{REP}_{\mathbf{IMPL}}$.

- □ On sait d'autre part que le terme t est congru à une valeur prédéfinie modulo la spécification descriptive $\mathbf{PRES_1}$, disons *p'* . Ce fait résulte de l'hypothèse de persistance de $\mathbf{PRES_1}$ par rapport à **P** (cf. notation 1). On a donc $t \equiv p'$ modulo $\mathbf{E_1}$.

- □ Supposons un instant que l'implémentation abstraite "implémente mal" $\mathbf{PRES_1}$. Cela voudrait dire que, pour au moins un terme *t* , *p* est différent de *p'* dans $T_{\mathbf{P}}$. Mais, après passage du foncteur d'identification (qui quotiente $\mathrm{REP}_{\mathbf{IMPL}}$ par $\mathbf{E_1}$), on aurait $p = p'$ dans le résultat sémantique $\mathrm{SEM}_{\mathbf{IMPL}}$. Ce qui conduit à ce que $\mathrm{SEM}_{\mathbf{IMPL}}$ ne soit pas consistant par rapport à $T_{\mathbf{PRES_1}}$.

Il suffit donc que le résultat sémantique $\mathrm{SEM}_{\mathbf{IMPL}}$ soit consistant par rapport à $T_{\mathbf{PRES_1}}$ pour que l'implémentation abstraite "simule" $\mathbf{PRES_1}$ (relativement à **P**). C'est là la définition de [EKMP 80].

Définition 3 :
L'implémentation abstraite **IMPL** est *consistante* si et seulement si le morphisme initial *init*

$$init : T_{\mathbf{PRES_1}} \rightarrow \mathrm{SEM}_{\mathbf{IMPL}}$$

est un monomorphisme (i.e. injectif).

Remarquant que, d'une part $\mathrm{SEM}_{\mathbf{IMPL}}$ est toujours finiment généré par rapport à la signature de $\mathbf{PRES_1} + \mathbf{P}$, et d'autre part $\mathrm{SEM}_{\mathbf{IMPL}}$ est un quotient de $\mathrm{REP}_{\mathbf{IMPL}}$, le théorème suivant est intuitivement clair (une démonstration précise se trouve dans [Ber 84]) :

Théorème 1 :
Les conditions suivantes sont deux à deux équivalentes :

- □ **IMPL** est consistante
- □ $\mathrm{SEM}_{\mathbf{IMPL}}$ est isomorphe à $T_{\mathbf{PRES_1}}$
- □ $\mathrm{SEM}_{\mathbf{IMPL}}$ est initiale dans $\mathrm{Alg}(\mathbf{PRES_1})$
- □ Il existe un $\mathbf{\Sigma_P \cup \Sigma_1}$-morphisme entre $\mathrm{REP}_{\mathbf{IMPL}}$ et $T_{\mathbf{PRES_1}}$
- □ Si deux $\mathbf{\Sigma_P \cup \Sigma_1}$-termes *t* et *t'* sont égaux dans $T_{\mathbf{OPimpl}}$, alors ils le sont aussi dans $T_{\mathbf{PRES_1}}$

Malheureusement, en pratique, aucune de ces conditions n'est facile à mettre en œuvre. En effet, ce sont des conditions *purement sémantiques* ; elle supposent toutes de connaître une description totale

de $REP_{\mathbf{IMPL}}$ ou $SEM_{\mathbf{IMPL}}$, or aucun outil formel simple (tel que la consistance hiérarchique) ne permet de les décrire facilement. Ceci est dû au fait que le foncteur de restriction est un foncteur n'ayant pas de "bonnes propriétés" algébriques.

Pour certains exemples, il est possible d'appliquer la proposition suivante, qui elle, fournit une condition utilisable plus simple :

Proposition 2 :
Pour que l'implémentation abstraite **IMPL** soit consistante, il *suffit* que **OPimpl**+<∅,∅,$\mathbf{E_1}$> soit une présentation consistante au-dessus de **P**+$\mathbf{PRES_1}$.

On peut alors traiter la correction d'une implémentation au moyen des outils habituels des types abstraits algébriques concernant la *consistance hiérarchique*.
Malheureusement, il subsiste beaucoup d'exemples pour lesquels cette proposition n'est pas applicable, ce qui détruit souvent la possibilité de faire des preuves simples de correction d'implémentation. En voici un (un exemple similaire est exposé dans [EKMP 80]) :

Exemple 7 :
Il s'agit simplement d'implémenter les entiers naturels (avec $0_{\mathbf{N}}$, $succ_{\mathbf{N}}$ et un prédicat d'égalité) au moyen des entiers relatifs. On aura donc l'opération d'abstraction suivante :

$$A : INT \rightarrow NAT$$

avec les équations d'implémentation suivantes :

$$\begin{aligned} 0_{\mathbf{N}} &= A(0_{\mathbf{Z}}) \\ succ_{\mathbf{N}}(A(z)) &= A(succ_{\mathbf{Z}}(z)) \\ eq_{\mathbf{N}}(A(z), A(z')) &= eq_{\mathbf{Z}}(z, z') \end{aligned}$$

Les objets de sorte *NAT* synthétisés inutilement par l'abstraction sont ici ceux de la forme *A(z)* avec *z<0*. Il n'empêche que cette implémentation est correcte (les éléments négatifs sont élagués par le foncteur de *restriction*, si bien que $REP_{\mathbf{IMPL}}$, et a fortiori $SEM_{\mathbf{IMPL}}$, sont isomorphes à **N**).
L'ennui se situe au niveau de la facilité à *prouver* que cette implémentation abstraite est correcte. Pour prouver qu'elle est consistante, c'est-à-dire que deux objets distincts ne sont pas implémentés par le même élément, le seul outil formel des types abstraits algébriques apte à traiter cette propriété de manière simple est la *consistance hiérarchique*. Pour ce faire, on est obligé de considérer la spécification contenant *à la fois* la spécification descriptive de *NAT* et la spécification constructive de l'implémentation (les 3 équations précédentes + *INT*) ; on peut alors tester si cette grosse spécification est consistante par rapport à la spécification descriptive (*NAT*). Ceci revient à tenter d'appliquer la proposition 2 précédente.
Voici la spécification de *NAT* :

$$\begin{aligned} eq_{\mathbf{N}}(0_{\mathbf{N}}, 0_{\mathbf{N}}) &= True \\ eq_{\mathbf{N}}(0_{\mathbf{N}}, succ_{\mathbf{N}}(m)) &= False \\ eq_{\mathbf{N}}(succ_{\mathbf{N}}(n), 0_{\mathbf{N}}) &= False \\ eq_{\mathbf{N}}(succ_{\mathbf{N}}(n), succ_{\mathbf{N}}(m)) &= eq_{\mathbf{N}}(n, m) \end{aligned}$$

et voici ce que l'on obtient vis à vis de la consistance :

$$True = eq_{\mathbf{N}}(0_{\mathbf{N}}, 0_{\mathbf{N}}) = eq_{\mathbf{N}}(0_{\mathbf{N}}, A(succ_{\mathbf{Z}}(-1))) = eq_{\mathbf{N}}(0_{\mathbf{N}}, succ_{\mathbf{N}}(A(-1))) = False\ .$$

Comme on le voit, la possibilité de faire des preuves simples de correction d'implémentations abstraites est fortement compromise avec le formalisme [EKP 80, EKMP 80] (et plus généralement avec la vision "abstraction"), à cause de l'élément *A(-1)* qui est inutilisé par l'implémentation

constructive, mais cependant synthétisé par l'abstraction.

Nous ne développerons pas ici la notion de *composition d'implémentations* selon [EKMP 80]. Ceci pour deux raisons principales : la composition de deux implémentations abstraites correctes (pour le formalisme [EKMP 80]) ne fournit pas toujours un résultat correct, et de plus, [EKMP 80] cherche à définir une notion de composition entièrement syntaxique, ce qui conduit à une définition assez complexe des critères correspondants.
Cette composition d'implémentation pour le formalisme [EKMP 80] est développée dans [Ber 84]. Le fait que deux implémentations correctes composées ne donnent pas un résultat correct provient encore du foncteur de restriction ; les éléments indésirables engendrés par les opérations de synthèse peuvent créer des inconsistances similaires à celle décrite dans l'exemple 7 précédent.

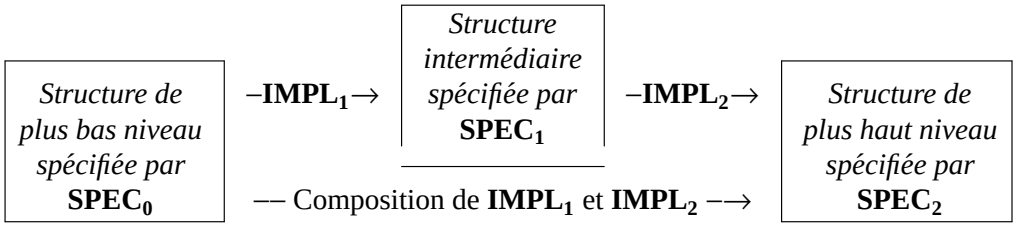
4. COMPOSITION D'IMPLEMENTATIONS, ET [SW 82]

Cette section se scinde en deux : elle montre d'abord quelle est l'utilité de la composition d'implémentations abstraites, et dans quel cadre elle se place ; puis elle décrit de quelle façon [SW 82] formalise la composition (pour ce formalisme, la composition de deux implémentations correctes reste toujours correcte).

4.1. LA COMPOSITION D'IMPLEMENTATIONS ABSTRAITES

On peut schématiser la composition de deux implémentations de la manière suivante :

Figure 4 : *composition d'implémentations*



Lorsque l'on prouve que l'implémentation $\mathbf{IMPL_2}$ est correcte, on considère d'une part la spécification descriptive $\mathbf{SPEC_2}$ et d'autre part la syntaxe de $\mathbf{IMPL_2}$ au-dessus de $\mathbf{SPEC_1}$. On prouve donc que $\mathbf{OPimpl_2} + \mathbf{SPEC_1}$ en correcte vis à vis de $\mathbf{SPEC_2}$. L'implémentation constructive $\mathbf{IMPL_1}$ de $\mathbf{SPEC_1}$ n'est nullement prise en compte dans cette démarche.
Toutes nos preuves de corrections étant faites en considérant $\mathbf{SPEC_1}$ uniquement, rien ne prouve *a priori* que l'implémentation de $\mathbf{SPEC_1}$ n'induise pas des "effets de bord", détruisant ainsi la correction globale. Il se peut fort bien que ($\mathbf{SPEC_0}$ et $\mathbf{IMPL_1}$) implémentent correctement $\mathbf{SPEC_1}$, que ($\mathbf{SPEC_1}$ et $\mathbf{IMPL_2}$) implémentent correctement $\mathbf{SPEC_2}$; mais que ($\mathbf{SPEC_0}$, et $\mathbf{IMPL_1}$ *suivi de* $\mathbf{IMPL_2}$) n'implémentent pas correctement $\mathbf{SPEC_2}$. Des exemples de ce malheureux phénomène sont décrits dans [EKMP 80] ou [Ber 84].
Pourtant, la notion de composition d'implémentations abstraites revêt une importance capitale. Par exemple, considérons le compilateur de **LAL** ([2]), qui fournit un texte en **C**. Supposons que l'on arrive à prouver formellement que ce compilateur est correct, c'est à dire qu'il réponde à une spécification

(2) langage **LISP** actuellement développé par Patrick Amar, L.R.I., Orsay.

de **LAL** au vu de la spécification de **C**. Il est clair que l'on veut qu'il en soit encore ainsi après compilation du texte **C** !
Il est donc de première importance que la composition de deux implémentations correctes fournisse globalement quelque chose qui implémente correctement la spécification de plus haut niveau. Il ne s'agit pas là à proprement parler d'une notion de structuration interne d'une implémentation ; il s'agit plutôt de pouvoir *réutiliser* des outils déjà implémentés (cf. section 1.2.2).

Le formalisme d'implémentation abstraite présenté par [SW 82] semble particulièrement séduisant vis à vis de ce problème de composition ; puisque la composition de deux implémentations abstraites correctes y est non seulement correcte, mais fournit une syntaxe de l'implémentation composée. Malheureusement, nous allons voir que cette notion de composition ne modélise pas la composition telle que nous l'avons intuitivement définie ici (elle modélise plutôt un passage pas à pas d'un niveau complètement "descriptif" à un niveau plus "constructif").

4.2. LE FORMALISME [SW 82]

Nous avons vu que, dans le formalisme [EKMP 80], l'implémentation de $\mathbf{SPEC_1} = \mathbf{P} + \mathbf{PRES_1}$ au moyen de $\mathbf{SPEC_0} = \mathbf{P} + \mathbf{PRES_0}$ se fait via la spécification constructive **OPimpl**. D'autre part, la sémantique devient problématique en ce qui concerne les preuves de correction uniquement à partir du foncteur de restriction ; le foncteur de synthèse ($T_{\mathbf{SPEC_0}} \rightarrow T_{\mathbf{OPimpl}}$) n'induit aucune difficulté.
[SW 82] simplifie la notion d'implémentation en se concentrant uniquement sur la partie "délicate" de l'implémentation : le passage de **OPimpl** à $\mathbf{SPEC_1}$. L'enrichissement de $\mathbf{SPEC_0}$ en **OPimpl** est laissé au soins de l'utilisateur :
"*we would say that* **OPimpl** *implements* $\mathbf{SPEC_1}$ *and leave the enrichment from* $\mathbf{SPEC_0}$ *to* **OPimpl** *to the user*".
(Pour une citation exacte, [SW 82] note **T** au lieu de $\mathbf{SPEC_0}$, **T'** au lieu de **OPimpl**, et **T"** au lieu de $\mathbf{SPEC_1}$).

Il reste donc à faire le "pont" entre la spécification *constructive* (**OPimpl**) et la spécification *descriptive* ($\mathbf{SPEC_1}$). Ceci est fait au moyen d'un morphisme de signatures entre celle de $\mathbf{SPEC_1}$ et celle de **OPimpl**. Pour comprendre le lien avec [EKMP 80], remarquons que **OPimpl** contient la signature de $\mathbf{SPEC_1}$, le morphisme de signatures sera alors simplement cette inclusion.

$$\mu : \mathbf{\Sigma(SPEC_1)} \rightarrow \mathbf{\Sigma(OPimpl)}$$

(d'une manière générale, [SW 82] ne suppose pas que μ soit injectif, mais l'aspect intuitif reste le même).

En ce qui concerne les preuves de correction :

- Il n'est plus question d'op-complétude, puisque celle-ci se place en amont de $T_{\mathbf{OPimpl}}$. L'op-complétude est donc laissée à la vigilance du concepteur. Ceci n'est pas gênant car nous avons vu qu'elle est facile à vérifier.

- La notion de simulation est définie dans [SW 82] d'une manière similaire à [EKMP 80] : elle utilise le même foncteur de *restriction* avant de procéder à une *identification*. L'identification est effectuée d'une manière plus générale que dans [EKMP 80] (autorisant des spécifications incomplètes), mais le foncteur de restriction présente les mêmes propriétés que celui de [EKMP 80]. Il en résulte donc les mêmes problèmes pour prouver formellement qu'une implémentation est correcte ; avec des théorèmes de correction tout à fait similaires.

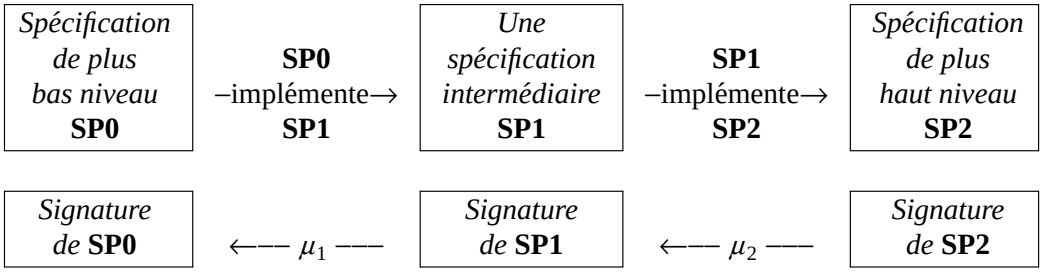
On n'a donc rien gagné quant à la faculté de prouver la correction d'une implémentation. Toutefois, les principaux avantages de ce formalisme sont :

- □ On peut définir des implémentations *paramétrées*, et les notions de correction restent valables quelque soit le paramètre. Nous ne développerons pas cet aspect, bien que ce soit une question importante. Nous préfèrerons nous concentrer (parties B et C suivantes) sur la question des implémentations en présence d'exceptions ; un axe ultérieur de recherche pourra se porter sur la paramétrisation en présence d'exceptions.

- □ La composition de deux implémentations est toujours une implémentation, y-compris au niveau syntaxique (évidemment : il suffit de prendre la composée des morphismes de signatures). Mais surtout, la composition de deux implémentations *correctes* est une implémentation *correcte*.

Il est temps maintenant de voir de plus près ce que signifie la composition d'implémentations compte tenu des hypothèses simplificatrices faites par [SW 82].

Figure 5 : *composition selon* [SW 82]



On constate que les morphismes de signatures vont en sens contraire de la sémantique [EKMP 80]. Ceci correspond à l'idée suivante : étant données les opérations *op2*, *op1* et *op0* de **SP2**, **SP1** et **SP0** ;

- □ $op1 = \mu_2(op2)$ signifie que l'opération *op1 représente* l'opération *op2* ;

- □ $op0 = \mu_1(op1)$ signifie que l'opération *op0* représente l'opération *op1* ;

- □ si bien qu'après composition, *op0* représente l'opération *op2*.

Il en est de même pour les sortes.
Il en résulte le fait nouveau suivant : toutes les sortes et opérations de la spécification de plus haut niveau (**SP2**) doivent nécessairement posséder des sortes et opérations correspondantes au plus bas niveau (**SP0**). En effet, pour chaque opération *op2* de **SP1**, il doit exister une opération $op0 = \mu_1(\mu_2(op2))$ qui le représente dans **SP0**.
Ce n'était pas le cas pour l'implémentation telle que nous l'entendions jusqu'à présent. Par exemple, la sorte *STACK* n'a aucune correspondance directe avec l'une des sortes *ARRAY* ou *NAT* ; a fortiori, aucune des opérations des piles (cf. *push* ou *pop*) n'a de correspondance directe avec l'une des opérations des tableaux.
A un renommage près, cela veut dire que la structure de plus bas niveau contient déjà toutes les sortes et opérations de celle de plus haut niveau. Cette contrainte a un effet nuisible sur la notion de composition telle que nous l'entendions. En effet, il n'est plus question d'utiliser une structure préexistante déjà implémentée pour en implémenter une de plus haut niveau : par exemple le langage **C** ne contient pas les fonctions primitives de **LISP**. La notion de composition selon [SW 82] obligerait à

étendre convenablement une implémentation déjà faite (**C**), avant de faire notre nouvelle implémentation (**LISP**). Ceci imposerait en particulier de refaire toutes les preuves de corrections relatives à l'implémentation de plus bas niveau, afin de prendre en compte les opérations nouvellement spécifiées. Ceci ne correspond nullement à ce que l'on entend faire.

Loin de nous l'idée de sous-estimer le formalisme de [SW 82]. Nous avons simplement montré que la notion de composition d'implémentations de [SW 82] n'est pas la même que celle qui nous préoccupe. Notre but est ici de modéliser sur le plan algébrique une *réutilisation*. Le but poursuivi (et atteint) par [SW 82] est, partant d'une spécification de haut niveau, d'aller pas à pas vers une spécification de suffisamment bas niveau pour que l'on puisse (presque) en déduire l'algorithme qui l'implémente. Et ce, de telle sorte que la correction de chacune des étapes franchies assure que la spécification de plus bas niveau simule bien la spécification de plus haut niveau. Autrement dit, la composition pour le formalisme [SW 82] ne cherche pas à modéliser une *réutilisation*, mais à fournir un outil *hiérarchique* de conception d'une (seule) implémentation ([3]).

5. RECAPITULATION

On peut résumer en trois points principaux les buts poursuivis ici concernant l'implémentation abstraite :

- □ On veut modéliser algébriquement l'implémentation d'une structure de données dont on connaît la spécification *descriptive*, au moyen d'une structure de données résidante (i.e. déjà implémentée) dont on connaît aussi la spécification *descriptive*.

- □ On aimerait, autant que possible, disposer de preuves de corrections formelles simples à mettre en œuvre.

- □ Un formalisme offrant des implémentations abstraites *reutilisables* est de première importance. Ceci est traduit algébriquement en deux critères : les implémentations abstraites doivent être compatibles avec les *enrichissements*, et la *composition* de deux implémentations abstraites correctes doit fournir un résultat correct.

Ceci étant, d'autres extensions sont envisageables ; citons par exemple la *paramétrisation*, l'implémentation de *spécifications incomplètes*, ou le *traitement d'exceptions*. Dans la partie C de cette thèse nous concentrerons nos efforts sur le traitement d'exceptions car de nombreuses implémentations peuvent difficilement être spécifiées de manière réaliste sans traitement d'exceptions (cf. section 1.2.3).

Les formalismes existants suivent essentiellement l'une des deux approches suivantes :

- □ On peut partir de la structure à implémenter et donner la *représentation* de chacun de ses termes. Cette représentation peut être principalement interprétée de deux manières :

    - On peut utiliser une seule fonction de représentation (ρ) qui fait correspondre un n-uplet d'objets résidants à chaque terme que l'on veut implémenter (cf. pile $\vdash\rho\rightarrow$ <tableau,entier>). On est malheureusement confronté au fait qu'il est difficile de donner une réelle existence algébrique aux axiomes qui en découlent ; dans la

(3) C'est du moins ce que j'ai compris après une discussion avec Don Sannella.

mesure où l'on veut traiter $< ... >$ et ρ comme des opérations, leur arité n'est pas évidente (en fait, $< ... >$ et ρ ne sont pas des fonctions, cf. section 2.2 ou [Gau 80]).

- On peut éclater la représentation en plusieurs opérations (cf. pile $\vdash\rho_1\rightarrow$ tableau , et pile $\vdash\rho_2\rightarrow$ entier). On est malheureusement confronté à des inconsistances (tous les tableaux sont amalgamés, cf. section 2.2). Là encore, ρ_1 n'est pas une fonction.

□ On peut inversement partir de la structure déjà implémentée, et synthétiser les n-uplets au moyen d'une opération d'*abstraction*. On peut donner une formalisation algébrique complète de cette démarche ([EKMP 80]), mais on est alors confronté au fait que l'abstraction synthétise "trop" d'objets produit. On est contraint d'utiliser un foncteur de restriction qui présente de mauvaises propriétés formelles, aussi bien envers la simplicité des preuves de correction qu'envers les enrichissements ou la composition d'implémentations abstraites.

Chapitre II :

Notre formalisme

d'implémentation abstraite

Ce chapitre contient sept sections. La première motive l'introduction d'un nouveau formalisme d'implémentations abstraites et décrit les choix effectués. Les sections 2 à 4 décrivent notre formalisme. La section 5 développe les divers critères de correction d'une implémentation abstraite. La section 6 fournit des conditions suffisantes utiles pour la réutilisation d'implémentations abstraites. Enfin la dernière section récapitule les résultats obtenus et dégage quelques développements ultérieurs possibles.

1. LA DEMARCHE SUIVIE

1.1. MOTIVATION

Nous avons remarqué au cours du chapitre précédent que les formalismes proposés ne permettent pas de prouver la correction d'une implémentation au moyen de critères formels. Ils fournissent tous des critères *purement sémantiques* de correction ; ce qui impose que le concepteur d'une implémentation soit capable de donner une description plus ou moins complète de certaines algèbres propres à la sémantique de l'implémentation (REP_{IMPL} ou SEM_{IMPL}). De tels critères n'apportent en fait aucune aide au concepteur. En effet, sitôt que les exemples d'implémentations sont un peu complexes, une description rigoureuse de ces algèbres est impossible (ou tout au moins source d'erreurs) car trop complexe. Pour aider efficacement à la conception d'une implémentation, il faut des critères de correction que l'on puisse vérifier de manière formelle ; c'est à dire directement à partir de la spécification de l'implémentation (de tels critères sont principalement l'induction structurelle, la suffisante complétude ou la consistance hiérarchique).

Le chapitre I a aussi montré que ces formalismes ne répondent pas de manière satisfaisante aux exigences de *réutilisation* d'implémentations abstraites. Pour des raisons similaires, tenter d'y inclure le *traitement d'exceptions* soulève de nombreux problèmes (voir [Ber 84]).

Il n'en reste pas moins que ces formalismes ont posé les bases d'une formalisation rigoureuse de l'implémentation abstraite, et en ont dégagé les points clefs. Le formalisme que nous proposons

maintenant s'en inspire fortement, et apporte quelques idées nouvelles qui permettent de résoudre les lacunes précédemment citées.

Nous avons choisi de nous placer dans le cadre des types abstraits algébriques avec *équations conditionnelles* positives, d'une part parce que les équations conditionnelles facilitent les spécifications (en évitant certaines opérations cachées), d'autre part parce que la prise en compte des équations conditionnelles nous offre une première étape vers l'implémentation abstraite en présence d'exceptions. En effet, nous développons en partie B une théorie de traitement d'exceptions utilisant une extension des axiomes conditionnels.

Notation :
Par convention, nous noterons les axiomes conditionnels positifs de la manière suivante :

$$x_1 = y_1 \ \wedge \ x_2 = y_2 \ \wedge \cdots \wedge \ x_n = y_n \quad \Rightarrow \quad x_{n+1} = y_{n+1}$$

tandis que, lorsque nous écrivions une équation de la forme :

$$u \ = \ \textit{if } b \textit{ then } v_1 \textit{ else } v_2$$

(par exemple pour le formalisme [EKMP 80] qui ne considère que des équations non conditionnelles) il s'agissait de l'opération *"if_ then_ else_"* habituelle

$$\textit{if\_ then\_ else\_ :} \quad \textit{BOOL SORT SORT} \quad \rightarrow \quad \textit{SORT}\ .$$

1.2. LES SOLUTIONS PROPOSEES

Comme nous l'avons dégagé dans la chapitre I, le rôle d'une implémentation abstraite est principalement d'assurer une correspondance entre chaque *terme à implémenter* et un *ensemble de n-uplets* qui l'implémente. Ce faisant, on soulève deux problèmes :

Restriction
certains n-uplets n'implémentent aucun terme ; il faut se *restreindre* aux n-uplets "utiles"

Identification
plusieurs n-uplets distincts peuvent implémenter le même objet ; il faut donc *identifier* les classes de n-uplets représentant le même objet.

Tous les obstacles rencontrés proviennent de ces deux points ; on peut donc raisonnablement penser qu'apporter une "bonne solution" à chacun d'eux résoudra la question. Les deux sous-sections suivantes motivent, et décrivent intuitivement, les solutions que nous proposons.

1.2.1. RESTRICTION ET PREUVES DE CORRECTION

Nous avons remarqué au cours du chapitre I que, si l'on suit l'approche "*abstraction*", c'est le problème de la *restriction* qui interdit des critères simples de correction d'une implémentation. Rappelons en effet l'exemple de l'implémentation des entiers naturels au moyen des entiers relatifs (exemple 7, section 3.5.2 du chapitre I) : c'est l'élément indésirable *A(–1)* qui empêche de ramener la correction de l'implémentation à un critère de consistance hiérarchique.
Le formalisme [EKP 80, EKMP 80], dont l'approche est de type "abstration", traite la restriction au moyen d'un *foncteur de restriction*. Pourtant ceci ne resoud pas la question des preuves de correction ; car le foncteur de restriction possède "de mauvaises propriétés formelles". Il serait préférable que la sémantique d'une implémentation abstraite puisse s'exprimer au moyen de foncteurs "classiques" en types abstraits algébriques (c'est à dire en fait : foncteur d'oubli et/ou son adjoint à gauche).
Intuitivement cela signifie qu'il faudrait refléter explicitement la restriction dès la spécification de l'implémentation. Les approches que nous avons décrites ne traitent la restriction qu'au niveau de la

sémantique, rien d'étonant donc que l'on ne puisse pas effectuer des preuves de correction uniquement fondées sur la spécification de l'implémentation.
Cette restriction pourrait être exprimée au moyen des *invariants de représentation* (cf. [Gau 80]), mais nous connaissons déjà un meilleur outil capable d'assurer cette restriction : la *représentation*. En effet, la représentation (ρ) fait correspondre des produits d'objets déjà implémentés *à chaque terme à implémenter* ; il en résulte que la représentation contient de manière intrinsèque la notion de *restriction* : la partie utile de l'implémentation est exactement son image.

Cependant, la représentation est difficile à formaliser algébriquement : il faudrait definir une opération "produit" (notée <...> dans le chapitre précédent, section 2.2), or cette opération produit conduit à des spécifications de type *abstraction* (cf. section 2.2 du chapitre précédent). C'est exactement ce que nous allons faire ; nous allons utiliser *à la fois* les approches "abstraction" et "représentation".

Intuitivement, avec l'abstraction, lorsque nous écrivons un axiome tel que

$$push(\, x\, ,\, A(t,i)\,) \quad = \quad A(\, t[i]:=x\, ,\, succ(i)\,)\ ,$$

on comprend bien que l'on ne spécifie pas réellement une opération sur les piles, mais plutôt une opération qui travaille sur les *couples <tableau,entier>*. Autrement dit, l'opération que l'on spécifie n'est pas réellement *push*, mais plutôt une opération "*constructive*" $\overline{push}$ qui en est sa *représentation*.
On commet donc une erreur d'approximation en disant que l'abstraction est de cible dans *STACK* ; elle est en fait de cible dans la sorte "produit" *Tableaux×Naturels* ; un terme *A(t,i)* dénote simplement un couple $<t,i>$.

$$\overline{push}(\, x\, ,<\, t, i\, >) \quad = \quad <\, t[i]:=x\, ,\, succ(i)\, >$$

On voit bien maintenant où nous voulons en venir :

- □ Partant des objets *résidants* (déjà implémentés), les opérations d'abstraction *synthétisent* des "n-uplets". Ces n-uplets ne sont plus de sorte pile (par exemple), mais appartiennent à une sorte "produit" intermédiaire.

- □ Dès lors, l'implémentation abstraite ne spécifie plus directement les opérations à implémenter (*empty*, *push*, *pop*, *top*), mais des opérations travaillant sur ces produits (que l'on notera $\overline{empty}$, $\overline{push}$, $\overline{pop}$ et $\overline{top}$).

- □ Enfin, la représentation sert simplement à exprimer que $\overline{empty}$ représente *empty* , $\overline{push}$ représente *push* ..etc..
  Récursivement, la représentation permet alors de faire correspondre son implémentation à chaque terme fermé à implémenter ; et son image est exactement l'ensemble des n-uplets "utiles à l'implémentation". Ainsi la restriction est exprimée *explicitement* au niveau des spécifications. Intuitivement, les n-uplets indésirables sont cette fois dans les sortes "produit" et non dans les sortes à implémenter. Si bien que ces sortes intermédiaires servent à "absorber" les éléments indésirables.

Terminologie :
On dira que la sorte produit "Tableaux×Entiers" *représente* la sorte *STACK*, et on la notera $\overline{STACK}$.
Rappelons que nous avons qualifié d'axiomes *constructifs* des axiomes tels que

$$\overline{pop}(<\, t\, ,\, succ(i)\, >) \quad = \quad <\, t\, ,\, i\, >$$

par opposition aux axiomes *descriptifs* tels que

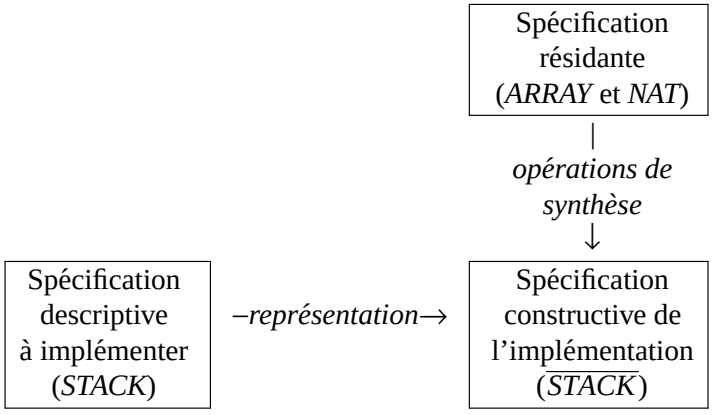
$$pop(push(x,X)) \quad = \quad X\ .$$

Par convention, on qualifiera de *constructives* les sortes "produit" telles que $\overline{STACK}$ (sur lesquelles

portent désormais les axiomes constructifs), par opposition aux sortes *descriptives* telles que *STACK* (sur lesquelles portent les axiomes descriptifs).
Enfin, la terminologie “opérations d’abstraction” n’est plus très bien choisie ici, puisque ces opérations ne retournent plus des objets “abstraits” de sorte *STACK* mais des n-uplets intermédiaires. Par contre, elles servent toujours à synthétiser ces n-uplets, c’est pourquoi nous les qualifierons d’*opérations de synthèse* (suivant ainsi la terminologie de [EKMP 80]).

Les idées que nous venons de développer sont résumées par la figure suivante :

Figure 6 : *implémentation abstraite : représentation + synthèse*



1.2.2. IDENTIFICATION ET REUTILISATION

Dans le cadre de la *réutilisation* d’implémentations abstraites, nous avons noté qu’une implémentation abstraite doit être compatible avec les *enrichissements*. Pour ce faire, il est nécessaire d’inclure *explicitement* une notion de *représentation de l’égalité* au sein même de l’implémentation abstraite. La raison technique en est la suivante.
Considérons l’exemple de l’implémentation de *SET* par *STRING*. Lorsque l’on voudra enrichir *SET*, la présentation associée contiendra éventuellement un ou plusieurs axiomes de la forme

$$X = Y \implies U = V .$$

Pour qu’une implémentation abstraite soit compatible avec cet enrichissement, il faut en particulier que la sémantique de cet axiome conditionnel soit la même au-dessus de la spécification de l’implémentation qu’au dessus de la spécification descriptive de *SET*. Il faut donc que chaque fois que la prémisse est satisfaite au-dessus de *SET*, elle le soit aussi au-dessus de l’implémentation.
Une instanciation possible d’un tel axiome est *X = insert(a,insert(b,empty)* et *Y = insert(b,insert(a,empty))* . Au niveau de la spécification descriptive de *SET*, *X* est égal à *Y* ; par conséquent cet axiome doit nécessairement s’appliquer, quelle que soit l’implémentation choisie de *SET*. Or, la représentation de *X* est *<"ab">* et celle de *Y* est *<"ba">* ; ces deux valeurs ne sont pas égales pour la spécification constructive de l’implémentation. Il est pourtant nécessaire que l’implémentation, quelle qu’elle soit, “sache” explicitement que *<"ab">* représente le même ensemble que *<"ba">*, afin d’appliquer cette instance d’axiome.
On reconnaît là notre problème d’*identification*. On sait que la *représentation de l’égalité* résoud ce point (cf. chapitre 1 section 2.1, ou [Gau 80]), sous réserve de prouver qu’elle est correcte (c’est ce que nous ferons).

Ainsi, pour qu'une implémentation abstraite soit compatible avec les enrichissements, il faut inclure explicitement la *représentation de l'égalité* dans sa spécification. Là encore, les formalismes décrits dans le chapitre précédent ne traitent l'identification qu'au niveau de la sémantique de l'implémentation ; comme pour la restriction nous choisissons de l'exprimer explicitement dans la spécification de l'implémentation, nous obtenons ainsi un formalisme d'implémentation abstraite répondant aux exigences de la réutilisation (compatibilité avec les enrichissements et la composition).

Remarquons que pour motiver l'introduction de la représentation de l'égalité nous avons utilisé des axiomes *conditionnels*. On pourrait penser qu'elle n'est pas utile avec des équations non conditionnelles. En fait, on sait que, quitte à ajouter des opérations cachées, on peut spécifier les mêmes classes d'algèbres avec ou sans axiomes conditionnels ; par conséquent la représentation de l'égalité est encore utile pour les equations non conditionnelles si l'on veut répondre aux exigences de réutilisation d'implémentations abstraites. D'autre part, on verra que la représentation de l'égalité facilite les preuves de correction d'une implémentation.

1.3. PRESENTATION DU FORMALISME

Nos principaux chevaux de bataille pour l'implémentation abstraite seront donc les *opérations de synthèse*, la *représentation* et la *représentation de l'égalité*. Intuitivement, ces outils correspondent respectivement aux foncteurs de la sémantique [EKMP 80] : *synthèse*, *restriction* et *identification* ; ils présentent l'avantage de s'exprimer dès le niveau des spécifications.

Nous suivrons une démarche similaire à celle proposée par [EKP 80] et [EKMP 80]. Nous utiliserons aussi les composantes de synthèse **SORTimpl** et **OPimpl** décrites dans le chapitre précédent, section 3 ; mais nous leur ajouterons deux autres composantes correspondant aux deux outils supplémentaires que nous nous sommes donnés : la *représentation* et la *représentation de l'égalité*.

Voici comment est décrite notre situation de départ :

- La structure de données *résidante* (déja implémentée) est spécifiée par la spécification $\mathbf{SPEC_0} = < \mathbf{S_0}, \mathbf{\Sigma_0}, \mathbf{A_0} >$; où $(\mathbf{S_0}, \mathbf{\Sigma_0})$ est la signature, et $\mathbf{A_0}$ est un ensemble d'axiomes (*conditionnels positifs* ([4])). Pour notre implémentation des piles, $\mathbf{SPEC_0}$ est la spécification de *ARRAY* et *NAT* ; pour celle des ensembles, $\mathbf{SPEC_0}$ est la spécification de *STRING* et *ELEM*.
  Remarquons que $\mathbf{SPEC_0}$ est une spécification *descriptive*. $\mathbf{SPEC_0}$ décrit les *propriétés abstraites* caractérisant la structure résidante, mais $T_{\mathbf{SPEC_0}}$ ne reflète pas nécessairement l'implémentation constructive (précedemment effectuée) des sortes résidantes.

- La structure de données que l'on veut implémenter est spécifiée par $\mathbf{SPEC_1} = < \mathbf{S_1}, \mathbf{\Sigma_1}, \mathbf{A_1} >$. $\mathbf{SPEC_1}$ est une spécification *descriptive*. Elle décrit les propriétes *que l'on veut obtenir* après que l'implémentation soit faite (par exemple *STACK+NAT*, ou *SET+ELEM*).

- Les spécifications $\mathbf{SPEC_0}$ et $\mathbf{SPEC_1}$ ne sont pas necessairement disjointes ; on notera $\mathbf{P} = <\mathbf{S},\mathbf{\Sigma},\mathbf{A}>$ la spécification commune entre $\mathbf{SPEC_0}$ et $\mathbf{SPEC_1}$, $\mathbf{P}=\mathbf{SPEC_0} \cap \mathbf{SPEC_1}$, (par exemple *ELEM*).
  Dans toute la suite, on supposera que $\mathbf{SPEC_0}$ et $\mathbf{SPEC_1}$ sont toutes deux *persistantes* au-dessus de **P**.

(4) Dans la suite de ce chapitre, le mot *axiome* signifiera *équation conditionnelle positive*.

Notre formalisme est exposé selon trois niveaux :

- □ Le *niveau textuel* décrit les informations minimales que doit fournir le concepteur à propos de l'implémentation.
- □ Le *niveau des présentations* est automatiquement déduit du niveau textuel. Il contient les diverses présentations au-dessus de la spécification résidante ($\mathbf{SPEC_0}$) qui définissent la spécification constructive de l'implémentation. On y retrouve en particulier les présentations **SORTimpl** et **OPimpl**, similaires au niveau syntaxique de [EKMP 80].
- □ Enfin le *niveau sémantique* est automatiquement déduit du niveau des présentations. Il décrit la sémantique de l'implémentation abstraite au moyen de deux foncteurs très simples.

2. LE NIVEAU TEXTUEL

Définition 4 :
Une *implémentation abstraite*, notée **IMPL**, est un quintuplet :

$$\mathbf{IMPL} = < \rho , \mathbf{\Sigma_{SYNTH}} , \mathbf{C} , \mathbf{A_{OP}} , \mathbf{A_{EQ}} >$$

dont les 5 composantes sont définies de la manière suivante.

1) ρ est l'isomorphisme de signatures défini comme suit :

- On note $\overline{\mathbf{S_1}}$ une copie de $\mathbf{S_1}$; $\overline{\mathbf{S_1}}$ est l'ensemble des *sortes constructives* : pour chaque sorte descriptive s de $\mathbf{SPEC_1}$, on se donne une sorte constructive $\bar{s}$ qui la représente.
- Pour chaque opération de $\mathbf{SPEC_1}$, $op \in \mathbf{\Sigma_1}$, d'arité $op: s_1\, s_2 \cdots s_n \rightarrow s$, on se donne l'*opération constructive* qui la représente, $\overline{op}$, dont l'arité est "translatée" vers les sortes constructives : $\overline{op}: \overline{s_1} \cdots \overline{s_n} \rightarrow \bar{s}$, où les $\overline{s_i}$ et $\bar{s}$ sont les sortes constructives de $\overline{\mathbf{S_1}}$ représentant les s_i et s, comme définies plus haut. On note $\overline{\mathbf{\Sigma_1}}$ l'ensemble de ces opérations constructives.

ρ est simplement l'isomorphisme de signatures entre $< \mathbf{S_1}, \mathbf{\Sigma_1} >$ et $< \overline{\mathbf{S_1}}, \overline{\mathbf{\Sigma_1}} >$ qui fait correspondre à chaque sorte $s \in \mathbf{S_1}$ la sorte constructive $\bar{s} \in \overline{\mathbf{S_1}}$ qui la représente, et à chaque opération $op \in \mathbf{\Sigma_1}$ l'opération constructive $\overline{op} \in \overline{\mathbf{\Sigma_1}}$ qui la représente. ρ est appelée la *représentation*.

2) $\mathbf{\Sigma_{SYNTH}}$ est l'ensemble des *opérations de synthèse* : pour chaque sorte constructive $\bar{s}$, $\mathbf{\Sigma_{SYNTH}}$ contient une (et une seule) opération de synthèse $< \cdots >_s : r_1 \cdots r_m \rightarrow \bar{s}$, où les sortes r_i sont des sortes résidantes (de $\mathbf{S_0}$).

3) **C** est la *composante cachée*. C'est une présentation $\mathbf{C} = < \mathbf{S_C}, \mathbf{\Sigma_C}, \mathbf{A_C} >$ au-dessus de $\mathbf{SORTimpl} = \mathbf{SPEC_0} + < \overline{\mathbf{S_1}}, \mathbf{\Sigma_{SYNTH}} >$.

4) $\mathbf{A_{OP}}$ est un ensemble d'axiomes sur la signature $< \mathbf{S_0} + \mathbf{S_C} + \overline{\mathbf{S_1}} , \mathbf{\Sigma_0} + \mathbf{\Sigma_C} + \mathbf{\Sigma_{SYNTH}} + \overline{\mathbf{\Sigma_1}} >$. Il décrit l'*implémentation des opérations constructives*.

5) Enfin, $\mathbf{A_{EQ}}$ est un ensemble d'axiomes pouvant utiliser toutes les opérations de l'implémentation abstraite ; il définit la *représentation de l'égalité*.

Cette définition appelle quelques commentaires.
Les *sortes constructives* de $\overline{\mathbf{S_1}}$ sont les sortes "produit" décrites en section 1.2.1 (par exemple $\overline{STACK}$) ; elles contiennent des "n-uplets" synthétisés par les *opérations de synthèse* de $\mathbf{\Sigma_{SYNTH}}$ ($< \_ , \_ >_{STACK} : ARRAY\ NAT \rightarrow \overline{STACK}$).
Chaque opération à implémenter (par exemple *push : ELEM STACK* $\rightarrow$ *STACK*) est représentée par une opération travaillant sur ces n-uplets ($\overline{push} : \overline{ELEM}\ \overline{STACK} \rightarrow \overline{STACK}$), dont l'implémentation constructive est spécifiée par les axiomes de $\mathbf{A_{OP}}$ ($\overline{push}(< x >_{ELEM} , < t, i >_{STACK}) =< t[i]:= x, succ(i) >_{STACK}$). Notons que cette spécification peut faire appel à des opérations cachées définies dans la *composante cachée* **C** (cf. section 3.2 du chapitre précédent).
La *représentation* ρ exprime quelle sorte ou opération constructive (par exemple $\overline{STACK}$ ou $\overline{push}$) représente chacune des sortes ou opération de $\mathbf{SPEC_1}$ (par exemple *STACK* ou *push*).
Enfin la représentation de l'égalité $\mathbf{A_{EQ}}$ spécifie quand plusieurs n-uplets représentent le même objet à implémenter, par exemple :

$$< t, 0 >_{STACK} \ = \ < t', 0 >_{STACK}$$
$$< t, i >_{STACK} =< t', i >_{STACK} \ \wedge \ t[i] = t'[i] \ \Longrightarrow \ < t, succ(i) >_{STACK} = < t', succ(i) >_{STACK}$$

On peut ainsi facilement traduire dans notre formalisme tous les exemples d'implémentation abstraite déjà vus ; voici celui de l'implémentation des ensembles au moyen des chaînes.

Exemple 8 :
L'implémentation des ensembles par les chaînes est définie comme suit au niveau textuel.

1) La représentation est l'isomorphisme de signatures ρ défini par :

SET	$\vdash\rho\rightarrow$	$\overline{SET}$	*empty*	$\vdash\rho\rightarrow$	$\overline{empty}$
BOOL	$\vdash\rho\rightarrow$	$\overline{BOOL}$	*insert*	$\vdash\rho\rightarrow$	$\overline{insert}$
ELEM	$\vdash\rho\rightarrow$	$\overline{ELEM}$	*delete*	$\vdash\rho\rightarrow$	$\overline{delete}$
			$\in$	$\vdash\rho\rightarrow$	$\bar{\in}$
			True	$\vdash\rho\rightarrow$	$\overline{True}$
			False	$\vdash\rho\rightarrow$	$\overline{False}$
			eq	$\vdash\rho\rightarrow$	$\overline{eq}$
			autres opérations de ELEM ...		

2) L'ensemble des opérations de synthèse, $\mathbf{\Sigma_{SYNTH}}$, contient les trois opérations

$$< \_ >_{SET} : STRING \rightarrow \overline{SET}$$
$$< \_ >_{ELEM} : ELEM \rightarrow \overline{ELEM}$$
$$< \_ >_{BOOL} : BOOL \rightarrow \overline{BOOL}$$

ce qui signifie que la sorte constructive $\overline{SET}$ est une copie de *STRING* ; et naturellement, la sorte constructive associée à chacune des sortes résidantes ($\overline{ELEM}$, $\overline{BOOL}$) est une copie de la sorte résidante elle même (*voir remarque 5 plus loin*).

3) L'ensemble des sortes cachées ($\mathbf{S_C}$) est vide ; les opérations cachées ($\mathbf{\Sigma_C}$) sont *occurs* et *remove* comme dans l'exemple 3 du chapitre précédent, avec les axiomes cachés ($\mathbf{A_C}$) suivants :

$$\begin{array}{rcrcl} & & \mathit{occurs}(x, \lambda) & = & \mathit{False} \\ & & \mathit{occurs(x,add(x,s))} & = & \mathit{True} \\ \mathit{eq(x,y)} = \mathit{False} & \Rightarrow & \mathit{occurs(x,add(y,s))} & = & \mathit{occurs(x,s)} \\ & & \mathit{remove}(x, \lambda) & = & \lambda \\ & & \mathit{remove(x,add(x,s))} & = & s \\ \mathit{eq(x,y)} = \mathit{False} & \Rightarrow & \mathit{remove(x,add(y,s))} & = & \mathit{add(y,remove(x,s))} \end{array}$$

(peu importe que *remove* n'enleve que la première occurence de *x* dans *s*, puisque seules les chaînes non redondantes sont atteintes par les opérations ensemblistes).

4) Les axiomes d'implémentation des opérations ($\mathbf{A_{OP}}$) sont les suivants :

$$\begin{array}{rcrcl} & & \overline{\mathit{empty}} & = & < \lambda >_{SET} \\ \mathit{occurs(x,s)} = \mathit{True} & \Rightarrow & \overline{\mathit{insert}}(< x >_{ELEM}, < s >_{SET}) & = & < s >_{SET} \\ \mathit{occurs(x,s)} = \mathit{False} & \Rightarrow & \overline{\mathit{insert}}(< x >_{ELEM}, < s >_{SET}) & = & < \mathit{add}(x, s) >_{SET} \\ & & \overline{\mathit{delete}}(< x >_{ELEM}, < s >_{SET}) & = & < \mathit{remove}(x, s) >_{SET} \\ & & < x >_{ELEM} \bar{\in} < s >_{SET} & = & < \mathit{occurs}(x, s) >_{BOOL} \end{array}$$

5) La représentation de l'égalité est spécifiée par :

$$\mathbf{A_{EQ}} = \{ < \mathit{add}(x, \mathit{add}(y, s)) >_{SET} = < \mathit{add}(y, \mathit{add}(x, s)) >_{SET} \} .$$

Remarque 5 :
L'isomorphisme de signatures ρ s'applique uniformément sur la signature de $\mathbf{SPEC_1}$, sans distinguer les sortes prédéfinies (celles de **P**). Ceci nous conduit à synthétiser des "copies constructives" des sortes et opérations prédéfinies, bien qu'elles soient déjà implémentées (cf. les sortes $\overline{BOOL}$ et $\overline{ELEM}$ dans l'exemple précédent). Il en résulte que nos axiomes d'implémentation ne s'expriment plus exactement comme avant. Par exemple, on écrit

$$\mathit{occurs}(x, s) = \mathit{False} \quad \Rightarrow \quad \overline{\mathit{insert}}(< x >_{ELEM}, < s >_{SET}) = < \mathit{add}(x, s) >_{SET}$$

au lieu de

$$\mathit{occurs}(x, s) = \mathit{False} \quad \Rightarrow \quad \mathit{insert}(x, < s >_{SET}) = < \mathit{add}(x, s) >_{SET}$$

Cette formulation unifie notre notion d'opération constructive : une opération constructive $\overline{op}$ ne prend son arité *que* dans des sortes constructives ; elle ne travaille pas directement sur la sorte *ELEM*, mais sur la sorte constructive qui la représente $\overline{ELEM}$.
Parfois, cela peut paraître moins naturel : l'axiome

$$< x >_{ELEM} \bar{\in} < s >_{SET} = < \mathit{occurs}(x, s) >_{BOOL}$$

est un peu déroutant. On préfèrerait une sorte *BOOL* commune aux spécifications descriptives et constructives. Néanmoins, une telle systématisation simplifie notre formalisme : il n'est pas nécessaire de distinguer les sortes à implémenter qui sont déjà résidantes de celles qui ne le sont pas (encore), ce qui simplifie la définition de ρ .
En fait, la motivation plus profonde de cette "copie constructive" systématique des sortes prédéfinies est la suivante : nous voulons modéliser la notion de *restriction* au moyen de la *représentation* ; or les produits synthétisés "en trop" peuvent induire, par contrecoup, des éléments indésirables aussi dans les sortes prédéfinies (cf. *top(<create,4>)=?*). En conséquence, il nous faut à tout prix faire porter la restriction (donc la représentation) *aussi* sur les sortes prédéfinies. D'où un traitement uniforme des nouvelles sortes à implémenter et de celles qui sont déjà résidantes. Quoi qu'il en soit, pratiquement, tout ceci sera *totalement transparent à l'utilisateur*, comme nous l'allons voir ci-dessous.

On constate que l'isomorphisme de signatures ρ est assez pauvre en informations : il se contente d'un renommage de s en $\bar{s}$, et de op en $\overline{op}$. De même, les opérations de synthèses associées aux sortes

prédéfinies ne sont que des opérations de copie.
En fait, les informations que doit apporter le concepteur d'une implémentation pourraient s'ecrire de manière plus réduite que notre définition textuelle. Les deux points précédemment cités peuvent être laissés implicites, le contexte permettant de les reconstruire. Ceci d'autant plus qu'ils ne modélisent pas réellement quelque chose à implémenter physiquement. La représentation ρ n'est qu'un un outil formel qui modélisera la *restriction*. Prenons l'exemple de *STACK*. On peut clairement laisser le concepteur d'une implémentation dans l'ignorance des sortes "$\overline{s}$" et opérations "$\overline{op}$", y compris pour ce qui concerne $\overline{STACK}$. La seule information que l'on ne puisse pas "inventer" de manière automatique, c'est que les piles sont représentées par un couple <*tableau,entier*> ; c'est à dire la partie gauche de l'arité de $< \ldots >_{STACK} : ARRAY\ NAT \rightarrow \overline{STACK}$. Information que le concepteur d'une implémentation pourra donner sous une forme telle que :

$$< \_ , \_ >_{STACK} : \ ARRAY\ NAT \ \text{--}implémente\text{--}> \ STACK$$

En ce qui concerne la composante cachée, il faut évidemment que le concepteur explicite le rôle des opérations cachées. Mais il peut le faire en spécifiant des opérations cachées d'arité dans les sortes de $\mathbf{S_1}$. Il suffit alors de translater toutes ces arités vers les sortes constructives (*s* devenant $\overline{s}$). Ceci aura même l'avantage de nous assurer d'office une des conditions de correction de l'implémentation (comme prouvé dans la proposition 3, section 5.2 plus loin).
Dès lors, les axiomes de $\mathbf{A_{OP}}$ peuvent être donnés sous une forme simplifiée comme suit (sans faire intervenir les opérations "$\overline{op}$" ni les opérations de synthèse des sortes prédéfinies) :

$$\begin{array}{rcl} empty & \approx & < t , 0 >_{STACK} \\ push\{ x , < t , i >_{STACK} \} & \approx & < t[i] := x , succ\ i >_{STACK} \\ pop\{ < t , succ\ i >_{STACK} \} & \approx & < t , i >_{STACK} \\ top\{ < t , succ\ i >_{STACK} \} & \approx & t[i] \\ & \text{..etc..} & \end{array}$$

où la notation "≈" peut être lue "*est implémenté par*" ; et la notation "$\{\cdots\}$" peut être lue "*appliqué à*" . Ainsi, le second axiome se lit : "L'opération *push* appliquée à $(x , < t, i >_{STACK})$ est implémentée par $< t[i] := x , succ\ i >$".

On obtient finalement la syntaxe très simple utilisée dans les premières approches de l'implémentation abstraite [Hoa 72, GMH 76, Gau 78 ...], mais nous pouvons maintenant *traduire* ces axiomes de manière à obtenir un niveau textuel ayant une réelle existence algébrique.
On voit donc que modulo cette convention qui laisse implicites les sortes constructives et la représentation, *notre formalisme peut être rendu totalement transparent à l'utilisateur*. On peut aisément reconstruire la définition textuelle complète d'une implémentation.

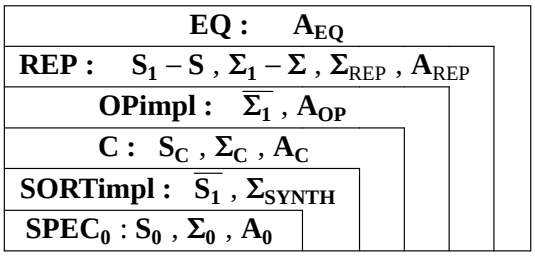
Tous les exemples d'implémentations donnés en annexe (annexes 2 et 4) sont spécifiés en suivant cette convention.

Partant du niveau textuel, nous allons maintenant montrer comment en déduire *de manière automatique* le niveau des présentations, puis le niveau sémantique.

3. LE NIVEAU DES PRESENTATIONS

Le niveau des présentations est défini par cinq présentations au-dessus de la spécification résidante $\mathbf{SPEC_0}$:

Figure 7 : *les présentations*



SORTimpl est une présentation au-dessus de $\mathbf{SPEC_0}$; **C** est une présentation au-dessus de $\mathbf{SPEC_0}$ + **SORTimpl** ; ... etc.
Par abus de langage, pour ne pas alourdir les notations, il nous arrivera souvent par la suite de noter "**EQ**" au lieu de $\mathbf{SPEC_0}$**+SORTimpl+C+OPimpl+REP+EQ** ; idem pour "**REP"**, "**OPimpl**" ... etc. Le contexte permettra de différencier s'il s'agit de la *présentation* **EQ** seule, ou de la *spécification* complète qu'elle domine.

Ces cinq présentations reflètent les cinq composantes décrites au niveau textuel : **SORTimpl** pour les *opérations de synthèse* ($\mathbf{\Sigma_{SYNTH}}$), **C** pour la *composante cachée*, **OPimpl** pour les axiomes d'*implémentation des opérations* constructives ($\mathbf{A_{OP}}$), **REP** pour la *représentation* (ρ), et **EQ** pour la *représentation de l'égalité* ($\mathbf{A_{EQ}}$).

On constate que la partie de la syntaxe comprise entre $\mathbf{SPEC_0}$ et **OPimpl** est quasiment la même que celle de [EKP 80, EKMP 80]. Nous lui avons ajouté deux niveaux : la *représentation* **REP**, qui traduit syntaxiquement la notion de restriction ; et la *représentation de l'égalité* **EQ**, qui traduit syntaxiquement la notion d'identification. Ces deux notions n'étaient que sémantiques dans le formalisme [EKMP 80].

Les parties $\overline{\mathbf{S_1}}$, $\mathbf{\Sigma_{SYNTH}}$, **C**, $\overline{\mathbf{\Sigma_1}}$, $\mathbf{A_{OP}}$, et $\mathbf{A_{EQ}}$ ont déjà été définies dans la section 2 précédente. Il nous reste donc à décrire $\mathbf{\Sigma_{REP}}$ et $\mathbf{A_{REP}}$.

- $\mathbf{\Sigma_{REP}}$ est l'ensemble des *opérations de représentation*. Pour chaque sorte $s \in \mathbf{S_1}$, $\mathbf{\Sigma_{REP}}$ contient une (et une seule) opération de représentation dont l'arité est :
$$\overline{\rho_s} : \; s \; \rightarrow \; \bar{s} \qquad \text{où } \bar{s} = \rho(s) \; .$$

- $\mathbf{A_{REP}}$ est l'ensemble des *axiomes de représentation*. Il traduit syntaxiquement le fait que la représentation associe son implémentation constructive à chaque opération à implémenter. Ceci signifie que pour chaque opération $op \in \mathbf{\Sigma_1}$, $\mathbf{A_{REP}}$ contient l'axiome :
$$\overline{\rho_s}(op(x_1, \ldots, x_n)) \; = \; \rho(op)(\, \overline{\rho_{s_1}}(x_1), \ldots, \overline{\rho_{s_n}}(x_n)\,) \quad (5)$$
où s est la sorte cible de op, et s_i est la sorte de x_i .
Il faut encore spécifier que $\overline{\rho_s}$ et $< \cdots >_s$ ne sont que des opérations de *copies* lorsque s est une sorte prédéfinie. Par conséquent, pour chaque sorte $s \in \mathbf{S}$, $\mathbf{A_{REP}}$ contient l'axiome suivant :
$$< x >_s \; = \; \overline{\rho_s}(x)$$
(par exemple cet axiome implique que $\bar{0} = < 0 >_{NAT}$ et que $\overline{succ}(< n >_{NAT}) = < succ(n) >_{NAT}$ dans $\overline{NAT}$).
Ces axiomes étant posés, il faut encore traduire le fait que si deux termes à implémenter

(5) rappelons que $\rho(op)$ est noté $\overline{op}$

possèdent la même représentation, alors ils apparaissent comme égaux après que l'implémentation soit faite. Par conséquent, pour chaque sorte $s \in \mathbf{S_1}$, $\mathbf{A}_{\text{REP}}$ contient l'axiome supplémentaire suivant :

$$\overline{\rho_s}(x) = \overline{\rho_s}(y) \quad \Rightarrow \quad x = y$$

Exemple 9 :
Pour l'implémentation des ensembles par les chaînes, $\mathbf{\Sigma}_{\text{REP}}$ contient les opérations

$$\overline{\rho_{SET}} : SET \rightarrow \overline{SET}$$
$$\overline{\rho_{BOOL}} : BOOL \rightarrow \overline{BOOL}$$
$$\overline{\rho_{ELEM}} : ELEM \rightarrow \overline{ELEM}$$

et $\mathbf{A}_{\text{REP}}$ contient les axiomes :

$$\begin{aligned}
\overline{\rho_{SET}}(empty) &= \overline{empty} \\
\overline{\rho_{SET}}(insert(x, X)) &= \overline{insert}(\overline{\rho_{ELEM}}(x), \overline{\rho_{SET}}(X)) \\
\overline{\rho_{SET}}(delete(x, X)) &= \overline{delete}(\overline{\rho_{ELEM}}(x), \overline{\rho_{SET}}(X)) \\
\overline{\rho_{BOOL}}(x \in X &= \overline{\rho_{ELEM}}(x) \bar{\in} \overline{\rho_{SET}}(X) \\
\overline{\rho_{BOOL}}(eq(x, y)) &= \overline{eq}(\overline{\rho_{ELEM}}(x), \overline{\rho_{ELEM}}(y)) \\
\overline{\rho_{BOOL}}(True) &= \overline{True} \\
\overline{\rho_{BOOL}}(False) &= \overline{False}
\end{aligned}$$

ainsi que :

$$\overline{\rho_{BOOL}}(b) = < b >_{BOOL}$$
$$\overline{\rho_{ELEM}}(x) = < x >_{ELEM}$$
$$\overline{\rho_{SET}}(X) = \overline{\rho_{SET}}(Y) \quad \Rightarrow \quad X = Y$$
$$\overline{\rho_{BOOL}}(b) = \overline{\rho_{BOOL}}(b') \quad \Rightarrow \quad b = b'$$
$$\overline{\rho_{ELEM}}(x) = \overline{\rho_{ELEM}}(y) \quad \Rightarrow \quad x = y$$

Ces axiomes sont nombreux, mais il faut bien voir qu'ils ne modélisent rien de "réel". Ce n'est qu'une interprétation théorique qui permet de spécifier explicitement la *restriction* (via la représentation). Approximativement, la partie représentation (**REP**) sera effectuée en pratique par un "contrôle syntaxique", qui ne laissera à l'utilisateur que le droit d'utiliser les opérations "reconnues" comme implémentées (celles de $\mathbf{SPEC_1}$). D'autre part, soulignons que $\mathbf{\Sigma}_{\text{REP}}$ et $\mathbf{A}_{\text{REP}}$ se déduisent *automatiquement* de la définition textuelle de l'implémentation, à partir de l'isomorphisme de signatures ρ, sans aucune spécification explicite du concepteur de l'implémentation.

Remarque 6 :
On pourrait penser que, du fait que notre formalisme impose une spécification explicite de la représentation de l'égalité, il ne peut s'appliquer que lorsque celle ci est exprimable algébriquement ; et qu'il est donc moins général que les formalismes décrits dans le chapitre précédents. En fait, *il n'en est rien*. En effet, rappelons que les axiomes de $\mathbf{A_{EQ}}$ peuvent utiliser *toutes les opérations* intervenant dans la spécification de l'implémentation. En particulier rien n'empêche de choisir $\mathbf{A_{EQ}} = \mathbf{A_1}$; dans ce cas, puisque la restriction est effectuée *avant* **EQ** (dans la composante **REP**), on retrouve exactement le même résultat sémantique que pour le formalisme [EKP 80, EKMP 80].

4. LE NIVEAU SEMANTIQUE

Puisque **SORTimpl**=<$\overline{\mathbf{S_1}}$, $\mathbf{\Sigma_{SYNTH}}$> , l'algèbre initiale $T_{\mathbf{SORTimpl}}$ contient, en plus des sortes résidantes, les sortes *constructives* de $\overline{\mathbf{S_1}}$. Elles sont synthétisées par les opérations de synthèses $< \cdots >_s$; et puisque **SORTimpl** ne contient aucun axiome, les éléments des sortes constructives sont seulement des "produits libres" : ce sont les n-uplets que l'on utilisera pour effectuer l'implémentation.
Puisque **OPimpl**=**C**+<$\overline{\mathbf{\Sigma_1}}$, $\mathbf{A_{OP}}$> , l'algèbre initiale $T_{\mathbf{OPimpl}}$ décrit l'implémentation constructive des opérations au moyen des axiomes de $\mathbf{A_{OP}}$ (avec l'aide de la composante cachée). $T_{\mathbf{OPimpl}}$ décrit donc l'action des opérations $\overline{op}$ sur les n-uplets de $T_{\mathbf{SORTimpl}}$.
Puisque **REP**=<$\mathbf{S_1 - S}$, $\mathbf{\Sigma_1 - \Sigma}$, $\mathbf{\Sigma}_{\text{REP}}$, $\mathbf{A}_{\text{REP}}$> , l'algèbre initiale $T_{\mathbf{REP}}$ contient maintenant les sortes à implémenter ($\mathbf{S_1}$) munies des opérations à implémenter ($\mathbf{\Sigma_1}$). De plus, les opérations et axiomes de représentation indiquent quelle est l'implémentation constructive de chaque terme fermé à implémenter. **REP** ne contenant aucune opération de cible dans $\mathbf{S_1}$−**S**, hormis celles de $\mathbf{\Sigma_1}$, $T_{\mathbf{REP}}$ ne contient aucun élément non atteignable dans les sortes à implémenter. Ainsi, $T_{\mathbf{REP}}$ joue un rôle similaire à l'algèbre $\text{REP}_{\mathbf{IMPL}}$ de [EKMP 80], mais résoud la question de *restriction* sans faire appel au "mauvais" foncteur de restriction.
Enfin, l'action de la représentation de l'égalité **EQ** est "remontée" vers les sortes descriptives (de $\mathbf{S_1}$) grâce aux axiomes de $\mathbf{A}_{\text{REP}}$ de la forme

$$\overline{\rho_s}(x) = \overline{\rho_s}(y) \quad \Longrightarrow \quad x = y \ ;$$

si bien que deux $\mathbf{\Sigma_1}$-termes ayant la même représentation modulo la représentation de l'egalité possèdent la même valeur dans $T_{\mathbf{EQ}}$. Il en résulte que $T_{\mathbf{EQ}}$ traite l'*identification*.

Tout ceci montre que l'algèbre initiale $T_{\mathbf{EQ}}$ modélise bien la sémantique de l'implémentation, à ceci près qu'elle contient encore les sortes constructives intermédiaires ($\overline{\mathbf{S_1}}$). Notre sémantique sera donc tout à fait simple, puisque réduite à deux foncteurs :

$$\begin{array}{ccccccc} \text{Alg}(\mathbf{SPEC_0}) & -F_{\mathbf{EQ}}\rightarrow & \text{Alg}(\mathbf{EQ}) & -U_{<\mathbf{S_1},\mathbf{\Sigma_1}>}\rightarrow & \text{Alg}(<\mathbf{S_1},\mathbf{\Sigma_1}>) \\ T_{\mathbf{SPEC_0}} & \vdash F_{\mathbf{EQ}}\rightarrow & T_{\mathbf{EQ}} & \vdash U_{<\mathbf{S_1},\mathbf{\Sigma_1}>}\rightarrow & \text{SEM}_{\mathbf{IMPL}} \end{array}$$

où $\text{SEM}_{\mathbf{IMPL}}$ est le *résultat sémantique* de l'implémentation, $\text{SEM}_{\mathbf{IMPL}} = U_{<\mathbf{S_1},\mathbf{\Sigma_1}>}(T_{\mathbf{EQ}})$.
Les foncteurs $F_{\mathbf{EQ}}$ et $U_{<\mathbf{S_1},\mathbf{\Sigma_1}>}$ sont les habituels foncteurs de synthèse et d'oubli. C'est bien là le but que nous nous étions assigné : ne décrire la sémantique d'une implémentation abstraite qu'au moyen de foncteurs d'oubli et/ou leurs adjoints à gauche, afin de disposer de moyens simples de vérification de correction.

5. LES PREUVES DE CORRECTION

Au vu de cette sémantique, les critères requis pour qu'une implémentation abstraite soit *correcte* sont clairs :

- □ On veut que tout terme fermé à implémenter possède une représentation parmi les n-uplets synthétisées par les opérations de synthèse.
- □ On veut que la "vision utilisateur" de l'implémentation soit isomorphe à la structure décrite par la spécification descriptive $\mathbf{SPEC_1}$. Ceci revient à dire que $\text{SEM}_{\mathbf{IMPL}}$ doit être isomorphe à $T_{\mathbf{SPEC_1}}$.

Nous scindons ces deux conditions en 4 critères (en éclatant la seconde condition en 3 critères) :

1) Le fait que tous les termes fermés possèdent une représentation est l'*opération-complétude*.

2) $SEM_{\mathbf{IMPL}}$ doit être finiment générée par rapport à $\mathbf{\Sigma_1}$. Ce critère est la *protection des données prédéfinies*.

3) $SEM_{\mathbf{IMPL}}$ doit être une $\mathbf{SPEC_1}$-algèbre c'est à dire qu'elle doit valider les axiomes de $\mathbf{SPEC_1}$ ($\mathbf{A_1}$). Ce critère est la *validité* de l'implémentation.

4) Enfin, parmi les $\mathbf{SPEC_1}$-algèbres, $SEM_{\mathbf{IMPL}}$ doit être initiale. Ce critère est la *consistance* de l'implémentation.

5.1. L'OPERATION-COMPLETUDE

Définition 5 :
L'implémentation abstraite **IMPL** est *opération-complète* (ou plus simplement *op-complète*) si et seulement si la représentation de tout terme fermé à implémenter possède une valeur parmi les n-uplets synthétisés. Ce qui s'écrit :

$$\forall\, t \in T_{\mathbf{\Sigma_1}}\ ,\ \exists\, \alpha \in T_{\mathbf{SORTimpl}}\ \ tel\ que\ \ \overline{\rho_s}(t) = \alpha\ \ dans\ T_{\mathbf{REP}}$$

(où s est la sorte de t)

Remarquons que lorsque l'on teste l'op-complétude, on ne veut en aucune façon que la représentation de l'égalité soit prise en compte. En effet, la spécification constructive de l'implémentation est récursivement définie dans la composante **OPimpl** (cf. remarque 1, section 2.1 du chapitre I précédent) ; la représentation de l'égalité ne sert qu'à spécifier les divers n-uplets qui implémentent le même objet descriptif à implémenter. Il en résulte que l'op-complétude doit être définie dans $T_{\mathbf{REP}}$ et non dans $T_{\mathbf{EQ}}$ ou $SEM_{\mathbf{IMPL}}$.

Le théorème suivant montre que l'on peut tester l'op-complétude directement sur les opérations constructives ; si bien que l'on est exactement ramenés à la notion d'op-complétude de [EKMP 80].

Théorème 2 :
Pour que **IMPL** soit op-complète, il faut et il suffit que pour tout $\overline{\mathbf{\Sigma_1}}$-terme fermé $\bar{t} \in T_{\overline{\mathbf{\Sigma_1}}}$ il existe un élément α de $T_{\mathbf{SORTimpl}}$ tel que $\bar{t} = \alpha$ dans $T_{\mathbf{OPimpl}}$.

Preuve :
Soit $\bar{\rho}$ l'isomorphisme entre $T_{\mathbf{\Sigma_1}}$ et $T_{\overline{\mathbf{\Sigma_1}}}$ déduit de l'isomorphisme de signatures ρ entre $<\mathbf{S_1}, \mathbf{\Sigma_1}>$ et $<\overline{\mathbf{S_1}}, \overline{\mathbf{\Sigma_1}}>$. D'après les axiomes de **REP**, il résulte que pour tout $\mathbf{\Sigma_1}$-terme fermé t , $\overline{\rho_s}(t)$ est dans la même classe d'équivalence, modulo **REP**, que le terme $\bar{t} = \bar{\rho}(t)$. Notre condition nécessaire et suffisante résulte donc du fait que $\bar{\rho}$ est un isomorphisme entre $T_{\mathbf{\Sigma_1}}$ et $T_{\overline{\mathbf{\Sigma_1}}}$. □

Ce résultat nous assure que l'op-complétude est simple à vérifier puisqu'elle peut se vérifier par induction structurelle sur les opérations de $\overline{\mathbf{\Sigma_1}}$, exactement comme pour le formalisme [EKMP 80]. Les exemples en sont aussi les mêmes, à une translation près de *op* à $\overline{op}$.
Nous retrouvons de plus sans effort la même proposition que pour le formalisme de [EKMP 80] :

Proposition 1*bis* :
Pour que l'implémentation abstraite **IMPL** soit op-complète, il *suffit* que la spécification **OPimpl** soit suffisamment complète au-dessus de **SORTimpl**.

Preuve :
Supposons que **OPimpl** soit suffisamment complète au-dessus de **SORTimpl**. La suffisante complétude nous assure que tout $\mathbf{\Sigma}$(**OPimpl**)-terme fermé de sorte dans $\overline{\mathbf{S_1}}$ est dans la classe d'un

Σ(**SORTimpl**)-terme. La conclusion résulte donc du fait que **Σ**(**OPimpl**) contient $\overline{\Sigma_1}$, et du théorème 2. □

La suffisante complétude de **OPimpl** *n'est pas une condition nécessaire* pour que **IMPL** soit op-complète. La raison intuitive en est la suivante : on sait que certains n-uplets ne sont pas utilisés par l'implémentation (cf. $< create, 4 >_{STACK}$). Peut importe dès lors si l'on ne sait pas associer un couple $< t, i >_{STACK}$ à un terme tel que $\overline{pop}(< create, 4 >_{STACK})$; on a alors affecté la suffisante complétude de **OPimpl** au-dessus de **SORTimpl**, mais nullement l'op-complétude de **IMPL** (puisque $\overline{pop}(< create, 4 >_{STACK})$ n'est jamais atteint par un terme fermé à implémenter).
Voici un autre exemple pour lequel la proposition précédente ne s'applique pas.

Exemple 10 :
Il s'agit d'implémenter les chaînes (d'éléments de sorte *ELEM*) par les tableaux (de ces mêmes éléments) : *STRING* par *ARRAY*. L'implémentation se fait de la manière suivante :

- □ Les éléments de la chaîne sont placés consécutivement dans le tableau qui la représente, à partir du rang 0.
- □ La fin de la chaîne est marquée sur le tableau grâce à un élément caché δ . L'élément δ est bien sûr un nouvel élément (caché) de sorte *ELEM* qui n'était pas présent avant (par exemple, en langage **C**, δ est égal à "\\0").

La composante cachée peut être spécifiée par :
$\mathbf{S_C} = \varnothing$
$\mathbf{\Sigma_C} = \{\ \delta\colon\ \rightarrow ELEM\ ,\ \ first\delta : ARRAY\ \ NAT \rightarrow NAT\ \}$
Où l'opération $first\delta(t, i)$ retourne le premier rang j supérieur à i tel que *t[j]* est égal à δ.

$$\mathbf{E_C} = \begin{array}{rcll} t[i] = \delta & \Rightarrow & first\delta(t,i) & = & i \\ eq(t[i], \delta) = False & \Rightarrow & first\delta(t,i) & = & first\delta(t, succ(i)) \end{array}$$

Les équations d'implémentation peuvent alors s'ecrire :

$$\begin{array}{rcl} \bar{\lambda} & = & t[0] := \delta \\ \overline{add}(< x >_{ELEM}, < t >_{STRING}) & = & < (t[succ(\ first\delta(t,0))] := \delta)\ [\ first\delta(t,0)] := x >_{STRING} \end{array}$$

Ce qui signifie que la chaîne vide est représentée par un tableau auquel on a affecté δ au rang 0 ; et que l'ajoût d'un élément à une chaîne représentée par *t* se fait en affectant *x* à la place de δ (le premier rencontré dans le tableau), et en décalant le repère de fin de chaîne, δ, d'un cran plus loin.

Pour cet exemple, **OPimpl** n'est pas suffisamment complet au-dessus de **SORTimpl**. En effet, si le tableau *t* ne contient aucune occurrence de δ, alors le terme $first\delta(t, 0)$ n'est égal à aucune valeur prédéfinie de sorte *NAT* car l'opération $first\delta$ n'est complètement définie que sur les tableaux contenant au moins une occurrence de δ après l'indice *i*. Il n'empêche que cette implémentation est op-complète, car tous les tableaux atteints par les opérations des chaînes contiennent au moins une occurrence de δ . C'est typiquement un cas où l'on ne peut prouver l'op-complétude que par induction structurelle :

- □ L'opération "chaîne vide" λ possède une représentation : tous les tableaux tels que *t[0]* égale δ. De plus, ce tableau contient au moins une occurrence de δ .

□ Supposons que s soit une chaîne implémentée par $< t >_{STRING}$ où le tableau t contient au moins une occurrence de δ . Alors, le terme *add(x,s)* est représenté par

$$< (t[succ(first\delta(t,0))]:=\delta)\,[\,first\delta(t,0)]:=x >_{STRING} .$$

Il nous faut prouver que ce tableau est un élément de $T_{\mathbf{SORTimpl}}$, c'est à dire qu'il est élément de T_{ARRAY} (car la partie de $T_{\mathbf{SORTimpl}}$ de sorte *ARRAY* est égale à T_{ARRAY}). Pour cela, il suffit de prouver que $first\delta(t,0)$ est élément de T_{NAT} ; ce qui se démontre par induction sur le rang de la première occurrence de δ dans t.

Pour completer notre raisonnement par induction structurelle, il faut encore prouver que le tableau

$$(t[succ(first\delta(t,0))]:=\delta)\,[\,first\delta(t,0)]:=x$$

contient au moins une occurrence de δ, ce qui est immédiat.

5.2. LA PROTECTION DES DONNEES PREDEFINIES

Nous voulons que le résultat sémantique $SEM_{\mathbf{IMPL}}$ soit $\mathbf{\Sigma_1}$-finiment généré. Le résultat suivant nous donne la solution :

Lemme 1 :
$SEM_{\mathbf{IMPL}}$ est $\mathbf{\Sigma_1}$-finiment générée si et seulement si **EQ** est suffisamment complète au-dessus de la spécification prédéfinie **P**.

Preuve :
Immédiat car $SEM_{\mathbf{IMPL}} = U_{<\mathbf{S_1},\mathbf{\Sigma_1}>}(T_{\mathbf{EQ}})$ et **EQ** ne contient aucune opération de cible dans $\mathbf{S_1} - \mathbf{S}$ autre que celles de $\mathbf{\Sigma_1}$. □

Définition 6 :
L'implémentation abstraite **IMPL** *protège les données prédéfinies* si et seulement si la spécification de l'implémentation, **EQ**, est suffisamment complète au-dessus de la spécification prédéfinie **P**.

Théorème 3 :
Si la composante cachée est suffisamment complète par rapport à la spécification prédéfinie, alors **IMPL** protège les données prédéfinies. Plus précisément, si la spécification $\mathbf{SPEC_0}$+**SORTimpl**+**C** est suffisamment complète au-dessus de la spécification prédéfinie **P**, alors $SEM_{\mathbf{IMPL}}$ est $\mathbf{\Sigma_1}$-finiment générée.

Preuve :
La spécification d'une implémentation abstraite ne contient aucune opération de cible dans une sorte de **S** autre que les opérations de $\mathbf{\Sigma_1}$, $\mathbf{\Sigma_0}$ et les opérations cachées. Il en résulte que la suffisante complétude de **C** au-dessus de la spécification prédéfinie **P** implique que $T_{\mathbf{EQ}}$ est finiment générée au-dessus de $T_{\mathbf{P}}$. □

En toute généralité, cette condition n'est pas nécessaire pour assurer que $SEM_{\mathbf{IMPL}}$ soit finiment générée : il suffirait que les axiomes ultérieurs ($\mathbf{A_{OP}}$ ou $\mathbf{A_{EQ}}$) rendent les opérations cachées complètement spécifiées. Toutefois, sur un plan méthodologique, il est clair que le rôle de $\mathbf{A_{OP}}$ et $\mathbf{A_{EQ}}$ n'est pas de spécifier les opérations cachées, si bien que ce théorème sera toujours appliqué en pratique.

La protection des données prédéfinies n'est pas difficile à établir sur les exemples, la suffisante complétude se vérifiant par induction structurelle ou par des critères syntaxiques tels que les présentations gracieuses (cf. [Bid 82]). Par exemple, notre implémentation de *STACK* protège les données prédéfinies parce que **C** est vide ; notre implémentation de *SET* les protège aussi parce que la

spécification de *occurs* et *remove* est une présentation gracieuse.

La protection des données prédéfinies signifie que la composante cachée doit être complètement spécifiée par rapport aux sortes de **S**, nullement par rapport aux sortes constructives de $\overline{\mathbf{S}}$:

Proposition 3 :
Si toutes les opérations cachées (de $\mathbf{\Sigma_C}$) sont de cible dans une sorte constructive ($\in\overline{\mathbf{S_1}}$), alors **IMPL** protège les données prédéfinies.

Preuve :
Immédiat, puisque dans ce cas, **SORTimpl+C** ne contient aucune opération de cible dans les sortes de $\mathbf{S_0}$, et $\mathbf{SPEC_0}$ est persistant au-dessus de **P**. □

Sachant que pour chaque sorte prédéfinie $s\in\mathbf{S}$, la sorte $\bar{s}$ n'est qu'une copie de *s*, ce résultat est très utile chaque fois que l'on ne tient pas à définir complètement les opérations cachées. C'est le cas de l'exemple 10 : pour que cette implémentation protège les données prédéfinies, il suffit de définir les opérations δ et $first\delta$ de cible dans $\overline{NAT}$ au lieu de *NAT*.

5.3. LA VALIDITE

Définition 7 :
L'implémentation abstraite **IMPL** est *valide* si et seulement si deux $\mathbf{\Sigma_1}$-termes fermés égaux dans $T_{\mathbf{SPEC_1}}$ le sont aussi dans $\mathrm{SEM}_{\mathbf{IMPL}}$; ce qui s'écrit :

$$\forall\, t \in T_{\mathbf{\Sigma_1}}\ ,\ \forall\, t' \in T_{\mathbf{\Sigma_1}}\ ,\ (\ t = t'\ dans\ T_{\mathbf{SPEC_1}}\ \Longrightarrow\ t = t'\ dans\ \mathrm{SEM}_{\mathbf{IMPL}}\ )$$

Notation 2 :
On note **IDimpl** = $\mathbf{SPEC_0}$**+SORTimpl+C+OPimpl+REP+EQ+**$<\varnothing,\varnothing,\mathbf{A_1}-\mathbf{A}>$; c'est à dire que **IDimpl** contient absolument tous les éléments de toutes les spécifications, aussi bien descriptives que constructives, invoquées dans notre formalisme. **IDimpl** ajoute les axiomes descriptifs (non résidants) à la spécification complète de l'implémentation.

Le théorème suivant montre qu'une implémentation est valide si et seulement si $T_{\mathbf{EQ}}$ valide les axiomes descriptifs de $\mathbf{SPEC_1}$. Par conséquent, la validité peut être vérifiée via des méthodes de *preuve de théorèmes*.

Théorème 4 :
Lorsque **IMPL** protège les données abstraites, les 7 conditions suivantes sont équivalentes :

- □ (1) **IMPL** est valide
- □ (2) Il existe un $\mathbf{\Sigma_1}$-morphisme de $T_{\mathbf{SPEC_1}}$ à $\mathrm{SEM}_{\mathbf{IMPL}}$
- □ (3) $\mathrm{SEM}_{\mathbf{IMPL}}$ valide les axiomes de $\mathbf{A_1}$ (i.e. est une $\mathbf{SPEC_1}$-algèbre)
- □ (4) $\mathrm{SEM}_{\mathbf{IMPL}}$ valide les axiomes de $\mathbf{A_1} - \mathbf{A}$
- □ (5) $T_{\mathbf{EQ}}$ valide les axiomes de $\mathbf{A_1} - \mathbf{A}$
- □ (6) Les axiomes de $\mathbf{A_1} - \mathbf{A}$ sont des théorèmes de **EQ** ([6])

□ (7) **IDimpl** est hiérarchiquement consistante au-dessus de **EQ** ([6])

Preuve :
[1⇔2] résulte du fait que notre définition de validité signifie qu'il existe un $\mathbf{\Sigma_1}$-morphisme de la partie finiment générée de $T_{\mathbf{SPEC_1}}$ vers la partie finiment générés de $\mathrm{SEM}_{\mathbf{IMPL}}$. Puisque $T_{\mathbf{SPEC_1}}$ est $\mathbf{\Sigma_1}$-finiment généré, cela équivaut à l'existence d'un $\mathbf{\Sigma_1}$-morphisme entre $T_{\mathbf{SPEC_1}}$ et $\mathrm{SEM}_{\mathbf{IMPL}}$.
[2⇔3] résulte de l'hypothèse de protection des données prédéfinies. En effet, $\mathrm{SEM}_{\mathbf{IMPL}}$ étant alors une $\mathbf{\Sigma_1}$-algèbre finiment générée, l'existence d'un $\mathbf{\Sigma_1}$-morphisme de l'algèbre initiale $T_{\mathbf{SPEC_1}}$ vers $\mathrm{SEM}_{\mathbf{IMPL}}$ (μ) implique que ce morphisme est surjectif. Il en découle que toute occurence d'axiome de $\mathbf{A_1}$ est validée par $\mathrm{SEM}_{\mathbf{IMPL}}$, puisqu'elle résulte, via μ, d'une occurence du même axiome sur $T_{\mathbf{SPEC_1}}$. Réciproquement, si $\mathrm{SEM}_{\mathbf{IMPL}}$ valide $\mathbf{A_1}$ alors c'est une $\mathbf{SPEC_1}$-algèbre, donc l'initialité de $T_{\mathbf{SPEC_1}}$ fournit un morphisme de $T_{\mathbf{SPEC_1}}$ vers $\mathrm{SEM}_{\mathbf{IMPL}}$.
[3⇔4] résulte du fait que $\mathrm{SEM}_{\mathbf{IMPL}}$ valide toujours les axiomes de $\mathbf{A}$, car $\mathrm{SEM}_{\mathbf{IMPL}}$ est une $\mathbf{\Sigma_1}$-sous-algèbre de $T_{\mathbf{EQ}}$ et **EQ** contient **P**.
[4⇔5] résulte directement du fait que les axiomes de $\mathbf{A_1 - A}$ ne concernent que les sortes descriptives, or $\mathrm{SEM}_{\mathbf{IMPL}}$ est l'oubli de $T_{\mathbf{EQ}}$ sur les sortes descriptives.
[5⇔6] est immédiat.
[5⇔7] résultent simplement du fait que **IDimpl** = **EQ**+($\mathbf{A_1 - A_0}$).
Ceci clôt notre preuve. □

Ce théorème démontre que la validité peut *toujours* être vérifié via des conditions de *consistance hiérarchiques*. C'est l'un des résultats majeurs de notre formalisme, car cela prouve que le problème de la correction d'une implémentation peut être ramené au problème bien connu de la consistance hiérarchique.

Exemple 11 :
Pour l'implémentation des ensembles par les chaînes, $\mathbf{A_1 - A}$ est :

$$\begin{array}{lcrcl}
 & & insert(x,insert(y,X)) & = & insert(y,insert(x,X)) \\
 & & delete(x,empty) & = & empty \\
 & & delete(x,insert(x,X)) & = & delete(x,X) \\
eq(x,y) = False & \Rightarrow & delete(x,insert(y,X)) & = & insert(y,delete(x,X)) \\
 & & x \in empty & = & False \\
 & & x \in insert(x,X) & = & True \\
eq(x,y) = False & \Rightarrow & x \in insert(y,X) & = & x \in X
\end{array}$$

Prenons par exemple le premier axiome. Les axiomes de représentation montrent que

$$\overline{\rho_{SET}}(insert(x, insert(y, X))) = \overline{insert}(\overline{\rho_{ELEM}}(x), \overline{insert}(\overline{\rho_{ELEM}}(y), \overline{\rho_{SET}}(X)))$$
$$\overline{\rho_{SET}}(insert(y, insert(x, X))) = \overline{insert}(\overline{\rho_{ELEM}}(y), \overline{insert}(\overline{\rho_{ELEM}}(x), \overline{\rho_{SET}}(X)))$$

et puisque notre implémentation est op-complète, on sait qu'il existe α, β et s tels que :

$$\overline{\rho_{ELEM}}(x) = <\alpha>_{ELEM} \quad \overline{\rho_{ELEM}}(y) = <\beta>_{ELEM} \quad et \quad \overline{\rho_{SET}}(X) = <s>_{SET}$$

Les axiomes d'implémentation des opérations constructives donnent alors :

$$\overline{insert}(<\alpha>_{ELEM}, \overline{insert}(<\beta>_{ELEM}, <s>_{SET})) = <add(\alpha, add(\beta, s))>_{SET}$$
$$\overline{insert}(<\beta>_{ELEM}, \overline{insert}(<\alpha>_{ELEM}, <s>_{SET})) = <add(\beta, add(\alpha, s))>_{SET}$$

(dans le cas où $\alpha \neq \beta$ et où α et β ne sont pas dans s ; mais les autres cas sont immédiats)
D'autre part, la représentation de l'égalité implique que

$$<add(\alpha, add(\beta, s))>_{SET} = <add(\beta, add(\alpha, s))>_{SET} .$$

si bien que $\overline{\rho_{SET}}(insert(x, insert(y, X))) = \overline{\rho_{SET}}(insert(y, insert(x, X)))$; et nous pouvons conclure

(6) plus précisément : $\mathbf{SPEC_0}$**+SORTimpl+C+OPimpl+REP+EQ**.

grâce à l'axiome qui "remonte" les identifications :

$$\overline{\rho_{SET}}(X) = \overline{\rho_{SET}}(Y) \quad \Longrightarrow \quad X = Y$$

Les autres axiomes se démontrent de la même façon, suivant ces méthodes de preuves de théorèmes bien connues.

Remarque 7 :
Dans l'exemple précédent, on peut détacher deux raisonnements qui sont les mêmes pour tous les exemples : le passage aux opérations constructives ($\overline{op}$) via la remarque que l'op-complétude nous assure que tout terme de la forme $\overline{\rho_s}(X)$ est égal à un terme de la forme $< t_1 \cdots t_n >_s$; et le fait de "remonter" les identifications via "$\overline{\rho_{SET}}(X) = \overline{\rho_{SET}}(Y) \Longrightarrow X = Y$".
Dans de nombreux cas, ces raisonnements conduisent à ne prouver que les "translations" des axiomes descriptifs vers leurs équivalents constructifs. Ainsi, le point clef de l'exemple précédent était la démonstration de l'égalité suivante :

$$\overline{insert}(< \alpha >_{ELEM}, \overline{insert}(< \beta >_{ELEM}, < s >_{SET})) = \overline{insert}(< \beta >_{ELEM}, \overline{insert}(< \alpha >_{ELEM}, < s >_{SET}))$$

On voit donc que souvent, il suffit de traiter les axiomes descriptifs auxquels on aura fait subir la transformation suivante :

- □ remplacer toutes les opérations op par leur représentation $\overline{op}$
- □ remplacer toutes les variables, disons x, par un n-uplet $< u_1 \cdots u_n >_s$ correspondant à leur sorte, les u_i étant donc des variables de sortes résidantes.

Par exemple, pour l'axiome descriptif $pop(push(x,X))=X$, il suffit de prouver l'axiome

$$\overline{pop}(\overline{push}(< \alpha >_{NAT}, < t, i >_{STACK})) \quad = \quad < t, i >_{STACK}$$

Ce qui revient à prouver que $< t[i] := \alpha, i >_{STACK} = < t, i >_{STACK}$, et résulte assez facilement de la représentation de l'égalité.

5.4. LA CONSISTANCE

Définition 8 :
L'implémentation abstraite **IMPL** est *consistante* si et seulement si : deux $\mathbf{\Sigma_1}$-termes fermés ne sont égaux dans $\mathrm{SEM}_{\mathbf{IMPL}}$ que s'ils le sont aussi dans $T_{\mathbf{SPEC_1}}$. Ce qui s'écrit :

$$\forall\, t \in T_{\mathbf{\Sigma_1}}\,, \forall\, t' \in T_{\mathbf{\Sigma_1}}\,, \ (\, t = t' \textit{ dans } \mathrm{SEM}_{\mathbf{IMPL}} \quad \Longrightarrow \quad t = t' \textit{ dans } T_{\mathbf{SPEC_1}}\,)$$

Le théorème suivant montre que la consistance signifie que le résultat sémantique $\mathrm{SEM}_{\mathbf{IMPL}}$ est consistant par rapport à l'algèbre initiale $T_{\mathbf{SPEC_1}}$.

Théorème 5 :
Si l'implémentation abstraite **IMPL** protège les données abstraites et est valide, alors les 6 conditions suivantes sont équivalentes :

- □ (1) **IMPL** est consistante
- □ (2) Le morphisme initial de $T_{\mathbf{SPEC_1}}$ dans $\mathrm{SEM}_{\mathbf{IMPL}}$ est un monomorphisme (i.e. est injectif)
- □ (3) $\mathrm{SEM}_{\mathbf{IMPL}}$ est initiale dans Alg($\mathbf{SPEC_1}$)
- □ (4) Il existe un $\mathbf{\Sigma_1}$-morphisme de $\mathrm{SEM}_{\mathbf{IMPL}}$ dans $T_{\mathbf{SPEC_1}}$
- □ (5) Le morphisme initial de $T_{\mathbf{SPEC_1}}$ dans $U_{<\mathbf{S_1},\mathbf{\Sigma_1}>}(T_{\mathbf{IDimpl}})$ est un monomorphisme
- □ (6) **IDimpl** est hiérarchiquement consistante au-dessus de $\mathbf{SPEC_1}$

Preuve :
[1⇔2] Tout d'abord, nos hypothèses de protection des données prédéfinies et de validité assurent que ce morphisme initial existe (théorème précédent). Il suffit alors de remarquer que la consistance de **IMPL** revient à dire que deux classes différentes de $T_{\mathbf{SPEC_1}}$ n'ont pas la même image dans $\mathrm{SEM}_{\mathbf{IMPL}}$ via le morphisme initial.
[2⇔3] résulte simplement du fait que $\mathrm{SEM}_{\mathbf{IMPL}}$ est finiment générée : le morphisme initial est toujours un épimorphisme, donc un isomorphisme ([7]).
[3⇔4] résulte de l'initialité de $T_{\mathbf{SPEC_1}}$: s'il existe un morphisme de $\mathrm{SEM}_{\mathbf{IMPL}}$ vers $T_{\mathbf{SPEC_1}}$, c'est nécessairement l'inverse du morphisme initial ; donc $\mathrm{SEM}_{\mathbf{IMPL}}$ est initiale si et seulement si il existe un morphisme de $\mathrm{SEM}_{\mathbf{IMPL}}$ vers $T_{\mathbf{SPEC_1}}$.
[2⇔5] Les hypothèses de protection des données prédéfinies et de validité assurent que $T_{\mathbf{EQ}}$ valide $\mathbf{A_1 - A}$. Par conséquent, $T_{\mathbf{EQ}}$ est égal à $T_{\mathbf{IDimpl}}$. Or par définition, $\mathrm{SEM}_{\mathbf{IMPL}} = U_{\mathbf{S_1}}(T_{\mathbf{EQ}})$, donc $\mathrm{SEM}_{\mathbf{IMPL}} = U_{\mathbf{S_1}}(T_{\mathbf{IDimpl}})$.
[5⇔6] résulte simplement du fait que le morphisme initial de $T_{\mathbf{SPEC_1}}$ dans $U_{\mathbf{S_1}}(T_{\mathbf{IDimpl}})$ est le morphisme d'adjonction associé à la présentation **IDimpl** au-dessus de $\mathbf{SPEC_1}$.
Ce qui clôt notre preuve. □

Pour vérifier la consistance hiérarchique de **IDimpl** au-dessus de $\mathbf{SPEC_1}$, il suffit de vérifier qu'aucun des axiomes de l'implémentation ne peut induire d'inconsistance sur les sortes descriptives. Nous venons donc de démontrer que le problème de la correction d'une implémentation peut être entièrement ramené au problème bien connu de la consistance hiérarchique. C'est là un de nos résultats majeurs.

Les seuls axiomes portant sur les sortes descriptives sont ceux-ci :

$$\overline{\rho_s}(x) = \overline{\rho_s}(y) \quad \Longrightarrow \quad x = y \qquad (s \in \mathbf{S_1})$$

La prémisse de chacun de ces axiomes nous conduit à prendre un à un les axiomes portant sur les sortes constructives (cibles des $\overline{\rho_s}$), et à vérifier qu'ils n'induisent pas d'inconsistance dans les sortes de $\mathbf{S_1}$. Les axiomes portant sur les sortes constructives sont ceux de $\mathbf{A_C} \cup \mathbf{A_{OP}} \cup \mathbf{A_{EQ}}$; ce sont donc eux qu'il faut considérer un à un.

Exemple 12 :
Pour l'implémentation de *SET* par *STRING*, on remarque que les axiomes cachés définissant *occurs* et *remove* sont consistants par rapport à **P** (présentation gracieuse sans équations entre les constructeurs, cf. [Bid 82]).
En ce qui concerne les équations de $\mathbf{A_{OP}}$, on prouve d'abord, par induction structurelle, que la chaîne $"x_1 x_2 \cdots x_n"$ (où les x_i sont deux à deux distincts) représente l'ensemble $\mathit{insert}(x_1, \mathit{insert}(x_2, \ldots, \mathit{insert}(x_n, \mathit{empty})..)$. (Facile mais long, nous ne le ferons pas ici).
Ensuite, il suffit de vérifier que chacun des axiomes de $\mathbf{A_{OP}}$ n'amalgame pas deux ensembles distincts, une fois mis sous cette forme. Par exemple :

$$\overline{\mathit{delete}}(< x >_{ELEM}, < s >_{SET}) = < \mathit{remove}(x, s) >_{SET}$$

signifie que l'opération *delete* appliquée à l'ensemble représenté par la chaîne $"x_1 \cdots x_n" = s$ retourne l'ensemble représenté par la chaîne $\mathit{remove}(x, "x_1 \cdots x_n")$.

- □ si x est différent de chacun des x_i, alors on obtient la même chaîne (s) (via les axiomes cachés, par récurrence sur la taille de la chaîne s). Il en résulte que *delete* retourne le même ensemble, ce qui ne crée par d'inconsistance.

---

(7) Le fait qu'un morphisme à la fois mono et épi soit un isomorphisme *n'est pas vrai dans toutes catégories*. C'est un résultat particulier aux catégories Alg(...) ; cf. [Ber 86].

- si x est égal à l'un des x_i, disons x_j, alors *remove(x,s)* est la chaîne "$x_1 .. x_{j-1} x_{j+1} .. x_n$" (récurrence sur la taille de la chaîne). Elle représente l'ensemble $insert(x_1, .. insert(x_{j-1}, insert(x_{j+1}, .. insert(x_n, empty)..)$. Ce qui ne crée toujours aucune inconsistance sur les ensembles.

Les autres axiomes se traitent de la même façon ; celui concernant *delete* était le plus délicat (notamment à cause de la disjonction des cas).
Enfin la consistance de la représentation de l'egalité est fort simple car l'axiome :

$$< add(x, add(y, s)) >_{SET} \quad = \quad < add(y, add(x, s)) >_{SET}$$

traduit exectement la commutativité de l'insertion :

$$insert(x,insert(y,X)) \quad = \quad insert(y,insert(x,X))$$

Remarquons au passage que la consistance des axiomes de $\mathbf{A_{EQ}}$ signifie précisément que la représentation de l'égalité n'est pas "trop forte".

Exemple 13 :
La consistance de l'implémentation des piles par les tableaux et les entiers est encore plus facile, nous n'aurons pas besoin de faire une disjonction de cas.
On prouve d'abord par induction structurelle que $< t, i >_{STACK}$ représente la pile *push(t[i-1],push(t[i-2],...,push(t[0],empty)..))*.
Puisque $\mathbf{A_C}$ est vide, il ne peut induire aucune inconsistance.
En ce qui concerne $\mathbf{A_{OP}}$, prenons par exemple l'axiome

$$\overline{pop}(< t, succ(i)) >_{STACK} \quad = \quad < t, i >_{STACK}$$

il se translate en :

$$pop(push(t[i-1],push(t[i-2],...,push(t[0],empty)..))) \quad = \quad push(t[i-2],...,push(t[0],empty)..)$$

ce qui ne crée bien sûr aucune inconsistance sur les piles à cause de l'axiome descriptif

$$pop(push(x,X)) \quad = \quad X$$

Enfin la représentation de l'égalité est clairement consistante :

$$< t, 0 >_{STACK} \quad = \quad < t', 0 >_{STACK}$$

$$< t, i >_{STACK} = < t', i >_{STACK} \wedge t[i] = t'[i] \implies < t, succ(i) >_{STACK} = < t', succ(i) >_{STACK}$$

Immédiat pour le premier axiome. Pour le second, il suffit de raisonner par récurrence sur i.

Nous avons détaillé les quatre critères de correction d'une implémentation abstraite. Nous pouvons maintenant poser la définition suivante :

Définition 9 :
L'implémentation abstraite **IMPL** est *correcte* si et seulement si :

- elle est *op-complète*
- elle *protège les données prédéfinies*
- elle est *valide*
- et elle est *consistante*.

Les méthodes de preuve de correction que nous avons données ici peuvent toutes se ramener à des critères tels que les présentations gracieuses, l'induction structurelle, ou la consistance hiérarchique. C'est en cela que réside l'intérêt majeur de notre formalisme, car elles reposent sur la connaissance de la *spécification* de l'implémentation, et non sur des connaissances relatives à sa sémantique.

5.5. LA CORRECTION FORTE

Cette section fournit des conditions suffisantes encore plus faciles à vérifier pour qu'une implémentation soit correcte. Elle démontre en particulier que les critères de *protection des données prédéfinies* et d'*op-complétude* sont très liés. Pour ce faire, nous définissons une notion de *correction forte* qui implique la correction. La plupart des exemples usuels d'implémentation abstraite sont fortement corrects. Nous montrerons d'autre part que la correction forte est une notion très utile pour dégager les implémentations compatibles avec la composition (section 6.2).

Définition 10 :
Soit **IMPL** une implémentation abstraite de la spécification $\mathbf{SPEC_1}$ au moyen de la spécification résidante $\mathbf{SPEC_0}$. On dira que **IMPL** *protège les termes résidants* si et seulement si la spécification de **IMPL** (i.e. **EQ**) est suffisamment complète au-dessus de la signature de $\mathbf{SPEC_0}$ (i.e. $<\mathbf{S_0}, \mathbf{\Sigma_0}>$).

Proposition 4 :
Si **IMPL** protège les termes résidants alors **IMPL** protège les données prédéfinies.

Preuve :
Immédiat, au vu des définitions 6 et 10.

Définition 11 :
L'implémentation abstraite **IMPL** est *fortement correcte* si et seulement si :

- □ elle est op-complète
- □ elle protège les *termes résidants*
- □ elle est valide
- □ et elle est consistante.

Les énoncés suivants fournissent une condition suffisante très simple pour assurer la correction forte.

Définition 12 :
L'implémentation abstraite **IMPL** est dite *fortement op-complète* si et seulement si la spécification **OPimpl** est suffisamment complète au-dessus de la *signature* de **SORTimpl**.

Proposition 5 :
Les trois conditions suivantes sont suffisantes pour que **IMPL** soit fortement correcte :

- □ **IMPL** est *fortement op-complète*
- □ **IMPL** est valide
- □ **IMPL** est consistante

Preuve :
Il suffit clairement de vérifier que si **IMPL** est fortement op-complète alors elle est op-complète et elle protège les termes résidants.
Le fait que l'op-complétude forte implique l'op-complétude résulte directement de la proposition 1*bis*.
Le fait que **SORTimpl** ne contienne aucune opération de cible dans $\mathbf{S_0}$ implique que **SORTimpl** est suffisamment complète au-dessus de la signature de $\mathbf{SPEC_0}$. Par conséquent, l'op-complétude forte

implique que **OPimpl** est suffisamment complète au-dessus de la signature de $\mathbf{SPEC_0}$; il suffit alors de remarquer que les axiomes de $\mathbf{A}_{REP}$ impliquent que **REP** (et a fortiori **EQ**) est suffisamment complète au-dessus de la signature de **OPimpl**. Il en résulte que l'op-complétude forte implique que **IMPL** protège les termes résidants. □

Grâce à cette proposition, il n'est pas difficile de vérifier que les exemples d'implémentation donnés dans ce chapitre sont fortement corrects, et donc corrects (les preuves relatives à l'op-complétude n'utilisent jamais les axiomes résidants). Seul l'exemple 10 (section 5.1 : implémentation des chaîne au moyen des tableaux) n'est pas fortement correct, car la composante cachée ajoute un élément δ dans la sorte prédéfinie *ELEM* (mais cet exemple ne protège pas non plus les données prédéfinies, voir aussi la proposition 3, section 5.2).

6. REUTILISATION D'IMPLEMENTATIONS

Nous avons déterminé (section 1.2.2 du chapitre précédent) que la question de *réutilisation* peut être traduite par deux critères : compatibilité avec les *enrichissements* et compatibilité avec la *composition*.

6.1. IMPLEMENTATIONS ET ENRICHISSEMENTS

Rappelons que si une spécification descriptive $\mathbf{SPEC_1}$ est abstraitement implémentée par **IMPL**, l'utilisateur de cette implémentation ne connaît que $\mathbf{SPEC_1}$. Lorsqu'il spécifie une présentation $\mathbf{PRES}=<\mathbf{S_{PRES}}, \mathbf{\Sigma_{PRES}}, \mathbf{A_{PRES}}>$ au-dessus de cette structure implémentée, toutes les preuves relatives à cet enrichissement concernent donc l'algèbre $T_{\mathbf{SPEC_1+PRES}}$. Pourtant, dans les faits, l'enrichissement sera effectué au-dessus de l'implémentation ; la structure de données résultante sera donc modélisée par $T_{\mathbf{EQ+PRES}}$ (rappelons que **EQ** est la spécification de l'implémentation). Il est donc souhaitable que la "vision utilisateur" de $T_{\mathbf{EQ+PRES}}$ soit isomorphe à $T_{\mathbf{SPEC_1+PRES}}$.
De la même façon que $SEM_{\mathbf{IMPL}} = U_{\mathbf{\Sigma_1}}(T_{\mathbf{EQ}})$ est la "vision utilisateur" de $T_{\mathbf{EQ}}$, la vision utilisateur de $T_{\mathbf{EQ+PRES}}$ est l'algèbre $U_{\mathbf{\Sigma_1+\Sigma_{PRES}}}(T_{\mathbf{EQ+PRES}})$. En effet, l'oubli $U_{\mathbf{\Sigma_1+\Sigma_{PRES}}}$ supprime des sortes et opérations internes de l'implémentation, auxquelles ni l'utilisateur, ni l'enrichissement n'ont accès (ces sortes et opérations sont *locales* à l'implémentation).

Le théorème suivant fournit une condition suffisante très utile pour qu'un enrichissement soit compatible avec une implémentation abstraite.

Théorème 6 :
Soit **IMPL** est une implémentation abstraite correcte de $\mathbf{SPEC_1}$. Soit **PRES** une présentation au-dessus de $\mathbf{SPEC_1}$. Si **PRES** est *consistante* au-dessus de $\mathbf{SPEC_1}$, et si elle est *suffisamment complète* au-dessus de $<\mathbf{S_1}, \mathbf{\Sigma_1}>$ alors :

$$U_{\mathbf{\Sigma_1+\Sigma_{PRES}}}(T_{\mathbf{EQ+PRES}}) \text{ est isomorphe à } T_{\mathbf{SPEC_1+PRES}}$$

Pour démontrer ce théorème, nous utiliserons le lemme suivant :

Lemme 2 :
Si $\mathbf{P_1}$ et $\mathbf{P_2}$ sont deux présentations persistantes, de signatures disjointes, au-dessus d'une spécification **SP**, alors $\mathbf{P_2}$ est encore une présentation persistante au-dessus de $\mathbf{SP} + \mathbf{P_1}$.

Preuve :
Puisque les sortes de $\mathbf{P_1}$ et $\mathbf{P_2}$ sont disjointes, les axiomes de $\mathbf{P_2}$ ne peuvent induire aucune inconsistance ou insuffisante complétude dans les sortes de $\mathbf{P_1}$. Il suffit donc de vérifier la persistance sur les

sortes de **SP**. Or, puisque $\mathbf{P_1}$ est persistante, l'oubli de $T_{\mathbf{SP+P_1}}$ sur les sortes de **SP** est isomorphe à $T_{\mathbf{SP}}$. De plus, puisqu'il n'y a pas de surcharge entre les opérations de $\mathbf{P_2}$ et celles de $\mathbf{SP+P_1}$, il en résulte que les occurences d'axiomes de $\mathbf{P_2}$ sont exactement les mêmes au-dessus de **SP** que de $\mathbf{SP+P_1}$; ce qui prouve que $\mathbf{P_2}$ a la même sémantique au-dessus de $T_{\mathbf{SP}}$ qu'au-dessus de $T_{\mathbf{SP+P_1}}$. Par conséquent, $\mathbf{P_2}$ est persistant au-dessus des sortes de **SP**, ce qui clôt la preuve du lemme. □

Preuve du théorème 6 :
La correction de **IMPL** assure que **IDimpl** est persistant au-dessus de $\mathbf{SPEC_1}$. Le lemme précédent implique donc que **IDimpl**+**PRES** est persistant au-dessus de $\mathbf{SPEC_1}$+**PRES**, c'est à dire que $T_{\mathbf{SPEC_1+PRES}}$ est isomorphe à $U_{\mathbf{\Sigma_1+\Sigma_{PRES}}}(T_{\mathbf{IDimpl+PRES}})$. Par conséquent, il suffit de prouver que $T_{\mathbf{IDimpl+PRES}}$ est isomorphe à $T_{\mathbf{EQ+PRES}}$.
Prouvons d'abord que **PRES** est persistant au-dessus de **EQ**. Puisque **PRES** est suffisamment complet au-dessus de $<\mathbf{S_1}, \mathbf{\Sigma_1}>$, **PRES** est a fortiori suffisamment complet au-dessus de **EQ** (qui contient $<\mathbf{S_1}, \mathbf{\Sigma_1}>$). D'autre part, puisque $U_{\mathbf{\Sigma_1+\Sigma_{PRES}}}(T_{\mathbf{IDimpl+PRES}}) \approx T_{\mathbf{SPEC_1+PRES}}$, **PRES** est persistant au-dessus de **IDimpl** ; **PRES** est donc en particulier consistant au-dessus **EQ** (car $\mathbf{IDimpl=EQ+A_1}$). Ceci prouve que **PRES** est persistant au-dessus de **EQ**.
Maintenant, la correction de **IMPL** assure que **IDimpl** est persistant au-dessus de **EQ**. Le lemme précédent implique donc que **IDimpl**+**PRES** est persistant au-dessus de **EQ**+**PRES**, c'est-à-dire que $T_{\mathbf{IDimpl+PRES}}$ est isomorphe à $T_{\mathbf{EQ+PRES}}$. □

Ce théorème résoudrait totalement la question s'il imposait seulement que **PRES** soit une présentation persistante au-dessus de $\mathbf{SPEC_1}$. Malheureusement, il faut vérifier que **PRES** soit suffisamment complète au-dessus de la *signature* de $\mathbf{SPEC_1}$ (c'est-à-dire sans utiliser les axiomes de $\mathbf{A_1}$). Remarquons néanmoins qu'il s'applique dans de nombreux cas, par exemple les présentations gracieuses ([Bid 82]) répondent à cette condition car l'ordre sur les opérations permet de réécrire tout $(\mathbf{\Sigma_1 + \Sigma_{PRES}})$-terme fermé de sorte dans $\mathbf{S_1}$ vers un $\mathbf{\Sigma_1}$-terme, sans utiliser les axiomes de $\mathbf{SPEC_1}$. Il n'en reste pas moins certains exemples de présentations persistantes qui ne sont pas compatibles avec une implémentation abstraite correcte :

Exemple 14 :
Considérons la spécification *NAT* contenant la sorte *NAT*, les opérations (*0:* $\rightarrow$ *NAT*) et (*succ: NAT* $\rightarrow$ *NAT*), et aucun axiome.
Considérons la spécification *PEANO* contenant la sorte *PEANO*, les opérations (*zero:* $\rightarrow$ *PEANO*) et (*next: PEANO* $\rightarrow$ *PEANO*), et l'axiome suivant :

$$next(x) = next(zero) \implies x = zero$$

Il est clair que T_{NAT} et T_{PEANO} sont isomorphes (à **N**). En particulier, on peut implémenter *PEANO* au moyen de *NAT* : il suffit de représenter *zero* par *0* et *next* par *succ*. Cette implémentation est clairement correcte.

Considérons maintenant la présentation **PRES** au-dessus de *PEANO* n'ajoutant aucune sorte, ajoutant l'opération constante τ, et l'axiome $\{next(\tau) = next(zero)\}$. **PRES** est persistante au-dessus de *PEANO* car l'axiome

$$next(x) = next(zero) \implies x = zero$$

implique que τ égale *zero*. C'est donc cet axiome qui rend la présentation **PRES** suffisamment complète au-dessus de *PEANO*.
Pourtant, l'implémentation de *PEANO* au moyen de *NAT* ne contenant pas cet axiome, la sémantique de **PRES** au-dessus de l'implémentation diffère de la sémantique au-dessus de *PEANO* (τ n'y est pas égal à *zero* bien que l'implémentation soit "isolément" correcte).

On conçoit cependant que de tels cas sont peu fréquents en pratique.

6.2. COMPOSITION D'IMPLEMENTATIONS ABSTRAITES

Supposons que l'on veuille implémenter une spécification $\mathbf{SPEC_2}$ au moyen d'une structure spécifiée par $\mathbf{SPEC_1}$; et que $\mathbf{SPEC_1}$ soit déjà implémentée au moyen de $\mathbf{SPEC_0}$. Là encore, toutes les preuves de corrections relatives à l'implémentation de $\mathbf{SPEC_2}$ ($\mathbf{IMPL_2}$) sont faites en regard de la spécification descriptive $\mathbf{SPEC_1}$, et nullement par rapport à l'implémentation constructive de $\mathbf{SPEC_1}$ ($\mathbf{IMPL_1}$). Pourtant, la structure effectivement implémentée est spécifiée par la réunion des spécifications des deux implémentations, à savoir :

$$\mathbf{SPEC_0} + (\mathbf{SORTimpl_1} + \mathbf{C_1} + \cdots + \mathbf{EQ_1}) + (\mathbf{SORTimpl_2} + \mathbf{C_2} + \cdots + \mathbf{EQ_2})$$

Remarque 8 :
Il est clair que la seconde implémentation ($\mathbf{IMPL_2}$) n'a pas accès aux sortes et opérations internes de la première ($\mathbf{IMPL_1}$). Par exemple, lorsqu'un texte écrit en **LAL** ([8]) est compilé vers un texte **C**, puis que le texte **C** obtenu est lui même compilé, l'implémentation se fait bien en deux temps : de **LAL** vers **C**, *puis* le source **C** est compilé.
Cela veut dire que les sortes et opérations intermédiaires de la première implémentation ($\mathbf{IMPL_1}$) sont *distinctes* de celles de la seconde ($\mathbf{IMPL_2}$). Ceci s'exprime par le fait que l'intersection des signatures de ($\mathbf{SORTimpl_1} + \cdots + \mathbf{EQ_1}$) et de ($\mathbf{SORTimpl_2} + \cdots + \mathbf{EQ_2}$) est vide.

Le théorème suivant fournit une condition suffisante très utile pour que la composition de deux implémentations fournisse un résultat correct du point de vue de l'utilisateur. Comme auparavant, le "point de vue de l'utilisateur" est obtenu par un foncteur d'oubli sur la signature de $\mathbf{SPEC_2}$.

Théorème 7 :
Etant données une implémentation, $\mathbf{IMPL_1}$, de $\mathbf{SPEC_1}$ au moyen de $\mathbf{SPEC_0}$, et une implémentation, $\mathbf{IMPL_2}$, de $\mathbf{SPEC_2}$ au moyen de $\mathbf{SPEC_1}$. Considérons la spécification **IMPL(1,2)** obtenue en sommant les spécifications des deux implémentations :

$$\mathbf{IMPL(1,2)} = \mathbf{SPEC_0} + (\mathbf{SORTimpl_1} + \cdots + \mathbf{EQ_1}) + (\mathbf{SORTimpl_2} + \cdots + \mathbf{EQ_2})$$

Si $\mathbf{IMPL_1}$ est *correcte* et si $\mathbf{IMPL_2}$ est *fortement correcte*, alors :

$$U_{\mathbf{\Sigma_2}}(T_{\mathbf{IMPL(1,2)}}) \text{ est isomorphe à } T_{\mathbf{SPEC_2}}$$

Preuve :
Puisque $\mathbf{IMPL_2}$ est fortement correcte, la présentation ($\mathbf{SORTimpl_2} + \cdots + \mathbf{EQ_2}$) est consistante au-dessus de $\mathbf{SPEC_1}$ et suffisamment complète au-dessus de $<\mathbf{S_1}, \mathbf{\Sigma_1}>$. Par conséquent, le théorème 6 précédent montre que $U_{\mathbf{SPEC_1+SORTimpl_2+..+EQ_2}}(T_{\mathbf{IMPL(1,2)}})$ est isomorphe à $T_{\mathbf{EQ_2}}$. En particulier, $U_{\mathbf{\Sigma_2}}(T_{\mathbf{IMPL(1,2)}})$ est isomorphe à $U_{\mathbf{\Sigma_2}}(T_{\mathbf{EQ_2}}) = \mathrm{SEM}_{\mathbf{IMPL_2}}$. Or la correction de $\mathbf{IMPL_2}$ nous assure que $\mathrm{SEM}_{\mathbf{IMPL_2}}$ est isomorphe à $T_{\mathbf{SPEC_2}}$; d'où la conclusion. □

Ce théorème peut clairement être étendu à toute composition finie d'implémentations, pourvu que la première soit correcte, et les suivantes soient fortement correctes.

Remarque 9 :
Bien que la composition de deux implémentations abstraites (fortement) correctes fournisse toujours un résultat sémantique correct, nous n'affirmons pas que la composée des deux définisse une nouvelle implémentation abstraite. En fait, ceci ne serait pas difficile à obtenir avec notre formalisme : il suffirait d'autoriser que l'arité des opérations de synthèse prenne sa source dans des sortes cachées (c'est d'ailleurs ce que font [EKP 80] et [EKMP 80]). On pourrait alors simplement inclure la spécification de la première implémentation ($\mathbf{IMPL_1}$) dans la composante cachée de la seconde. Il n'est pas

(8) Langage **LISP** actuellement développé par Patrick Amar, L.R.I., Orsay.

difficile de voir que tous nos résultats seraient encore valables (modulo la correction *forte*), car nous n'utilisons jamais cette hypothèse dans nos preuves.
Toutefois, ce que l'on veut, semble-t'il, c'est donner une *représentation* de chacun des termes à implémenter sous forme de *n-uplets* d'éléments "préexistants" (i.e. résidants, non pas cachés). Nous préférons donc adopter ici une attitude prudente sur ce point. Dans le cas contraire, on obtient un formalisme d'implémentation clos par composition (c'est à dire que la composée d'un nombre fini d'implémentations est encore une implémentation). On peut alors parler de la *sous-catégorie des implémentations fortement correctes* ... plaisant pour l'esprit, mais il n'est pas certain que cela traduise exactement l'idée que l'on se fait des implémentations.

7. CONCLUSION ET PERSPECTIVES

Le formalisme que nous avons proposé repose sur deux idées principales résolvant respectivement les problèmes de restriction et d'identification :

1) Nous introduisons des sortes intermédiaires, contenant des n-uplets d'éléments résidants (i.e. déjà implémentés). Ces n-uplets sont construits au moyen d'*opérations de synthèse*, tandis que des opérations de *représentation* fournissent explicitement la correspondance entre les termes à implémenter et leurs représentation.

2) Nous spécifions explicitement la *représentation de l'égalité*.

Les résultats nouveaux apportés par ce formalisme sont :

- Contrairement aux formalismes existants, les preuves de correction ne reposent que sur la connaissance de la *spécification* de l'implémentation, et non sur sa sémantique. Les critères de correction s'expriment tous au moyen de concepts tels que la consistance hiérarchique ou la suffisante complétude, ce qui ramène la question de correction d'une implémentation à des problèmes bien connus des types abstraits algébriques.

- Ce formalisme d'implémentation abstraite se place dans le cadre des *équations conditionnelles positives*, ce qui facilite les spécifications, et permettra une extension aisée au *traitement d'exceptions* (voir partie C de cette thèse).

- Des conditions simples et peu restrictives pour la *réutilisation* d'implémentations abstraites peuvent être dégagées.

Le formalisme que nous venons de développer présente donc un certain nombre de potentialités non négligeables, mais il présente aussi une certaine "lourdeur", en ce sens que les sortes intermédiaires compliquent la syntaxe. Nous avons montré (remarque 5 et fin de la section 2) que cette syntaxe peut être simplifiée pour le concepteur d'une implémentation. Mais il serait utile de systématiser ce procédé de simplification afin de l'inclure dans un environnement algébrique (de type *ASSPEGIQUE* [BC 85, BCV 85]), et tester alors quels sont les résultats obtenus en soumettant des preuves de correction à un démonstrateur automatique de théorèmes.

D'autre part, en ce qui concerne la notion de *réutilisation* d'implémentations, les résultats que nous obtenons sont assez probants, mais il serait bon d'automatiser cet aspect, et voir quelles techniques de gestion d'une "bibliothèque de spécifications" seraient les plus efficaces.

Sur un plan plus purement “types abstraits algébriques”, notre partie C montre que ce formalisme d’implémentation abstraite est facilement étendu aux types abstraits algébriques avec traitement d’exceptions. Ce résultat nous semble révélateur de la fiabilité et des potentialités de ces deux formalismes, car les implémentations abstraites et le traitement d’exceptions sont tous deux des points problématiques de la théorie des types abstraits algébriques. Le fait qu’ils soient compatibles laisse espérer d’autres extensions sans plus de difficultés.
Remarquons enfin que, bien que nous n’ayons pas (encore) étudié ce point, il ne devrait pas y avoir de difficulté à étendre ce formalisme d’implémentations abstraites au cas des spécifications paramétrées, car notre sémantique est entièrement fonctorielle et n’utilise que des foncteurs d’oubli ou de synthèse (rappelons que la notion de paramétrisation repose essentiellement sur les pushouts et les foncteurs de synthèse, cf. [ADJ 80]).

PARTIE B :

TRAITEMENT D'EXCEPTIONS

DANS LES

TYPES ABSTRAITS ALGEBRIQUES

Résumé de cette partie B

Cette partie développe dans le détail un nouveau formalisme de traitement d'exceptions dans le cadre des types abstraits algébriques, que nous nommerons le formalisme des *exception-algèbres*.

Les exception-algèbres répondent à tous les critères utiles les spécifications algébriques avec traitement d'exceptions : déclaration aisée des valeurs "*Ok*", spécification du comportement des valeurs erronées, spécification de récupérations éventuelles, propagation implicite des exceptions et des erreurs, existence d'une algèbre initiale, structuration des spécifications, redéfinition des propriétés de consistance hiérarchique et suffisante complétude. De plus, nous apportons deux nouveautés pour le traitement d'exceptions dans le cadre des types abstraits algébriques.

- Quitte à inclure un traitement d'exceptions dans les spécifications algébriques, autant modéliser aussi des "messages d'erreur". C'est pourquoi nous incluons dans la signature d'une spécification un nouvel ensemble, l'ensemble des *étiquettes d'exception*, qui modélisent les "diagnostics d'exception". Une exception-algèbre est alors définie comme une algèbre classique, à ceci près que ses valeurs peuvent être étiquetées par diverses étiquettes d'exception.
  Cette technique offre une grande souplesse de spécification car on peut alors inclure des conditions sur ces étiquetages dans les prémisses des axiomes. On peut ainsi traiter les exceptions d'une manière différenciée, selon les divers cas exceptionnels rencontrés. Ceci recouvre aussi bien des techniques de *récupérations* d'exceptions, qu'un traitement au sein même des valeurs erronées.

- D'autre part, nous montrons qu'il est de première utilité de pouvoir différencier les termes qui sont "*Ok*" par le traitement "normal" de la structure de données spécifiée, de ceux qui sont *Ok* parce qu'ils ont été *récupérés*. Les premiers peuvent être entièrement traités par les axiomes usuels du type (les "*Ok*-axiomes"), alors que les seconds doivent subir un traitement exceptionnel avant de réintégrer le rang des valeurs *Ok* (par récupération). D'une manière duale, on peut résumer cette idée par le "slogan" suivant :
  Il faut bien distinguer les *termes exceptionnels* des *valeurs erronées*.
  Ce n'est pas parce qu'un terme est exceptionnel (et donc mérite un traitement exceptionnel) que sa valeur est erronée (il peut être récupéré) ; et ce n'est pas parce que la valeur d'un terme est *Ok* qu'il n'est pas exceptionnel (toujours parce que sa valeur peut résulter d'une récupération, et par conséquent, ce terme peut demander un traitement en tant qu'exception avant d'être récupéré).

Ces deux idées permettent de développer une sémantique dont les propriétés sont très proches des propriétés classiques des types abstraits algébriques (i.e. sans traitement d'exceptions) : elle est compatible avec la notion de congruences, et surtout, on peut retrouver les résultats classiques (ADJ) d'existence de *congruences minimales*.

Cette existence de congruences minimales nous offre alors une moisson importante de résultats similaires aux résultats connus sans traitement d'exceptions, principalement :

- La catégorie des algèbres (finiment générées ou non) validant une exception-spécification possède toujours un objet initial (et un objet terminal).

- On peut *structurer* les exception-spécifications grâce aux notions de présentations et enrichissements. Et ceci de telle sorte que les foncteurs d'oubli et de synthèse existent, et sont

adjoints.

- Lorsque l'on considère une présentation, on peut définir la *consistance hiérarchique* et la *suffisante complétude* de manière simple (grâce à l'adjonction entre les foncteurs d'oubli et de synthèse).

- Plus généralement, du fait que la sémantique des exception-algèbres peut s'exprimer d'une manière entièrement fonctorielle, les primitives usuelles de structuration des spécifications formelles peuvent, pour la plupart, s'étendre à notre formalisme sans difficulté majeure. La partie C de cette thèse en est un exemple : elle étend le formalisme d'implémentations abstraites aux cas avec traitement d'exceptions.

TABLE DES MATIERES

TABLE DES MATIERES (SUITE)

TABLE DES MATIERES (SUITE)

Chapitre I :

Introduction

Ce chapitre contient quatre sections. La première rappelle les raisons pour lesquelles le traitement d'exceptions est un sujet crucial. La seconde dégage les critères requis pour un "bon" formalisme de traitement d'exceptions au sein des types abstraits algébriques. La section 3 décrit brièvement les formalismes existants sur ce sujet ; et la section 4 récapitule les résultats déjà obtenus ou encore à atteindre concernant le traitement d'exceptions dans les types abstraits algébriques.

1. MOTIVATION

Au cours de la partie A, nous avons montré que l'on peut efficacement utiliser les types abstraits algébriques pour traiter formellement le problème de l'*implémentation*. Cependant, le "pouvoir de modélisation" du formalisme que nous y avons développé est fortement limité par le manque de traitement d'exceptions : par exemple on ne peut pas traiter les structures bornées (cf. partie A, chapitre I, section 1.2.3). Pourtant, il est d'un intérêt majeur de pouvoir déterminer quelles sont les bornes des structures nouvellement implémentées, en fonction des bornes des structures utilisées par l'implémentation.
Plus généralement, le processus de conception d'un système informatique serait fortement amélioré si le traitement des cas exceptionnels était prévu dès l'étape préliminaire de spécification formelle. En effet, le traitement d'exceptions est le plus souvent négligé lors de la conception d'un système informatique, ce qui conduit à de nombreuses modifications faites a posteriori, et rend le traitement des cas exceptionnels dépendant de l'implémentation considérée. Il est clair qu'une spécification formelle du traitement des cas exceptionnels conduirait à une conception plus structurée, où la correction d'une implémentation prendrait en compte le traitement d'exceptions.

Le traitement d'exceptions est donc un sujet majeur des types abstraits algébriques. On constate que tous logiciels, même les plus simples, nécessitent un traitement de cas exceptionnels ; ce traitement d'exceptions doit être prévu dès l'étape préliminaire de conception, et par conséquent il est crucial que le formalisme des types abstraits algébriques puisse modéliser cet aspect. De plus, le traitement d'exceptions est pris en compte dans les langages relativement récents ; il serait dommage que les types abstraits algébriques ne suivent pas cette évolution.
Sous cet angle, il semble que le traitement d'exceptions soit l'un des points les plus cruciaux pour étendre le champs d'application des types abstraits algébriques. D'autres sujets tels que la paramétrisation, les spécifications incomplètes, les aspects de réécriture ..., sont aussi tout à fait cruciaux ; cependant le traitement d'exceptions aura notre faveur car, au moins pour l'implémentation abstraite,

c'est l'absence de traitement d'exception qui nuit le plus à la "crédibilité pratique" des résultats obtenus.

2. LES CRITERES UTILES

Si l'on veut développer un formalisme complet de traitement d'exceptions, il doit répondre à un minimum de contraintes pour offrir un pouvoir d'expression assez souple. Une partie de ces contraintes est imposée par ce que l'on veut pouvoir modéliser ; à ce sujet, les cinq principes exposés par Goguen dans [Gog 77] fournissent une bonne première approche :

- *Think about errors from the beginning, from the preliminary design stage on.*
- *Include all exceptional state behaviour, especially error messages, directly in the specifications.*
- *Provide maximum oppurtunity for the user to make errors of the sort which reveal inconsistencies in his conception of what he is doing.*
- *This implies that the langage designer should require "redundant" information of the user.*
- *As much information as is hepful about what went "wrong" (or exceptional) should be provided, as a basis for debugging (or further processing in an exceptional state).*

Ces cinq principes élémentaires nous conduisent :

- à inclure le traitement d'exception dès le niveau des types abstraits algébriques (qui peuvent être considérés comme une des étapes préliminaires de conception),
- à élaborer une notion de message d'erreur dans les spécifications, et à pouvoir spécifier le comportement des états exceptionnels,
- à définir des spécifications structurées, munie d'une notion de consistance apte à prendre en compte le traitement d'exceptions,
- enfin à définir une notion de "spécification complète" des valeurs erronées ou exceptionnelles (fournir "suffisamment d'informations" correspondant à la notion de spécification complète).

D'autres critères facilitent les spécifications :

- Pour ne pas devoir caractériser explicitement de nombreux cas erronés ou exceptionnels, il est nécessaire de disposer d'une *propagation implicite* des erreurs et des exceptions (assurée au niveau sémantique, simplifiant ainsi les specifications). Par exemple, si *succ(Maxint)* est exceptionnel (où *Maxint* est le plus grand entier non erroné), on aimerait bien ne pas devoir déclarer explicitement que *succ(succ(Maxint))*, *succ(succ(succ(Maxint)))* ...etc.. sont aussi exceptionnels.
- Dans le même ordre d'idée, si l'on déclare que les valeurs "*Ok*" sont celles de la forme $succ^n(0)$ avec $0 \leq n \leq Maxint$, on n'a nulle envie de déclarer explicitement que *0+0 0+succ(0)* ...etc ... sont eux aussi "*Ok*". Par conséquent, il est aussi très utile de pouvoir déclarer les valeurs "*Ok*" sous une forme "de référence" (sous forme normale par exemple, ici les termes de la forme $succ^n(0)$).

- □ On aura envie dans de nombreux cas de *récupérer* certaines exceptions. Par exemple, on peut vouloir dans certains cas (pas nécessairement tous les cas) qu'un terme tel que *(Maxint+1)–2*, qui est exceptionnel (par propagation implicite), soit tout de même récupéré en la valeur *Maxint–1* qui, elle, n'est pas erronée (et ce, malgré une étape intermédiaire via la valeur erronée *Maxint+1*).

Enfin, pour avoir quelques chances de définir des spécifications structurées munies de propriétées telles que la consistance hiérarchique ou la suffisante complétude (utiles pour l'implémentation abstraite ou toute autre primitive de structuration des spécifications), certaines contraintes sémantiques s'imposent :

- □ Il faut définir un *foncteur d'oubli* associé à une notion de *présentation*.
- □ Il faut aussi définir une notion de *foncteur de synthèse*, (adjoint à gauche au foncteur d'oubli) sur lequel reposent les notions de consistance hiérarchique et suffisante complétude.
- □ Plus particulièrement, on sait que l'existence d'un foncteur de synthèse est très liée aux notions de *congruence minimale* et d'*objet initial* (cf. [Ber 86]).

Ainsi les critères qui nous permettront de juger un formalisme algébrique de traitement d'exception seront principalement :

- □ L'aptitude à modéliser la notion de message d'erreur.
- □ La facilité à spécifier les cas "normaux" et les cas exceptionnels.
- □ La facilité à décrire le comportement des cas exceptionnels (messages d'erreur, dépendances entre les diverses exceptions ...).
- □ La possibilité de récupérer certains termes exceptionnels.
- □ La propagation implicite des exceptions et des erreurs.
- □ L'existence de critères tels que la consistance hiérarchique ou la suffisante complétude, de telle sorte que les cas exceptionnels ou les messages d'erreur soient pris en compte. Ce qui est très lié à l'existence d'un objet initial au niveau sémantique.

Munis de ces critères, nous allons décrire quelques travaux concernant le traitement d'exceptions au sein des types abstraits algébriques ; et nous allons constater qu'aucun d'eux ne les satisfait pleinement.

3. QUELQUES TRAVAUX EXISTANTS

Le but de cette section n'est pas d'étudier en détail les formalismes de traitement d'exceptions existants. Nous nous appuierons simplement sur l'étude faite dans [BG 83] pour montrer l'évolution de ce sujet relativement aux critères précédemment cités, et nous décrirons très brièvement deux formalismes algébriques développés plus récemment : [GDLE 84] et [Bid 84].

3.1. UNE IDEE SIMPLE QUI NE CONVIENT PAS

Tout d'abord, avant de concevoir un nouveau formalisme de traitement d'exceptions dans le cadre des types abstraits algébriques, remarquons que le formalisme "classique" des types abstraits algébriques ne permet pas de résoudre le problème de manière satisfaisante.

Une idée simple serait d'utiliser le formalisme algébrique habituel, en ajoutant un élément erroné "*UNDEF*" dans chaque sorte. Par exemple, on poserait *pred(0)=UNDEF* et *succ(Maxint)=UNDEF*.
Malheureusement, le seul fait d'ajouter une telle valeur dans chaque sorte pose immédiatement un problème de *complétude* de la spécification. En effet, il faudrait alors savoir comment traiter tous les surtermes de *UNDEF* : *succ(UNDEF), pred(UNDEF)* ...
Le fait que l'on veuille une propagation des erreurs nous conduirait à poser :

succ(UNDEF)=pred(UNDEF)=UNDEF.

Mais ceci conduit à des inconsistances : compte tenu de l'axiome

$$pred(succ(n)) \quad = \quad n$$

on obtient :

$$pred(succ(Maxint)) = Maxint = pred(UNDEF) = UNDEF$$

d'où l'on peut déduire que tout entier naturel est égal à *UNDEF*. Ce n'est certainement pas là ce que l'on voulait modéliser.
On pourrait encore tenter de poser : *pred(UNDEF)=Maxint* , afin d'éviter l'inconsistance décrite précédemment. Mais on obtiendrait alors *pred(pred(0))=pred(UNDEF)=Maxint* ; ce qui n'est certainement pas souhaitable.
Une autre tentative serait-alors d'avoir plusieurs valeurs erronées par sorte (une pour $pred(0) = UNDEF_1$, une pour $succ(Maxint) = UNDEF_2$, une pour $n \; div \; 0 = UNDEF_3$, etc ...). Mais cette fois, on est confronté de manière encore plus aigue aux problèmes de complétude : quelle valeur donner à $UNDEF_1 + UNDEF_3$?

Pour éliminer les inconsistances telles que celle décrite plus haut (*Maxint=UNDEF*), il est nécessaire de faire un traitement "plus fin" pour déterminer lorsqu'un axiome doit s'appliquer ou non. Un tel traitement a été proposé par le groupe ADJ, au moyen d'axiomes conditionnels (cf. [ADJ 76]), en introduisant un prédicat de "répartition" :

$$OK_s : \; s \; \rightarrow \; BOOL$$

permettant de préciser quelles sont les valeurs *Ok* ou erronées de chaque sorte *s*. Ce prédicat est alors utilisé dans les prémisses d'axiomes (conditionnels) pour limiter leur application.
Une telle approche permet de définir un formalisme correct du traitement d'exceptions dans les types abstraits algébriques. Cependant, comme le montre [BG 83], il conduit à des spécifications peu lisibles où les cas "normaux" et les cas d'erreur sont complètement mélangés. De plus, hormis l'existence d'une algèbre initiale, cette approche ne remplit aucun des critères que nous avions déterminé pour un traitement d'exceptions efficace. Il est donc nécessaire de définir un formalisme nouveau pour un bon traitement d'exceptions.

3.2. LES DIVERSES APPROCHES

[BG 83] montre que les diverses solutions proposées pour résoudre ce problème peuvent être classées en trois "écoles" principales :

- Les travaux qui abandonnent l'aspect algébrique au profit d'une approche algorithmique ou opérationnelle ; c'est le cas de [Loe 81, EPE 81]. Une telle approche ne rencontre évidemment plus les problèmes décrits précédemment, puisqu'un cas d'erreur se traduit simplement par un cas d'arrêt des algorithmes. On évite ainsi les phénomènes d'inconsistance ou de non

suffisante complétude, puisqu'ils se situent en aval de l'erreur rencontrée.
Toutefois, cela ne permet pas un traitement complet des exceptions. Puisque l'on stoppe à la première erreur rencontrée, on se refuse d'office tout traitement effectué sur les valeurs erronées elles-mêmes (amalgame ou comparaison de plusieurs erreurs). Dans le même ordre d'idées, lorsque l'on se trouve dans une situation erronée, on ne peut pas faire de récupération ultérieure, ou alors au prix de complications assez importantes.
De plus, en abandonnant l'aspect algébrique, on perd toutes les facilités offertes par les types abstraits algébriques. Ceci est dû au fait qu'il est beaucoup plus aisé de raisonner sur des relations d'équivalence (congruences) que de traiter directement des équivalences d'algorithmes.

□ Les travaux fondés sur des algèbres partielles. Dans une algèbre partielle, les opérations peuvent ne rien retourner pour certaines valeurs de leur arguments (les cas d'erreurs). Cette vision de *fonctions partielles* fait l'objet de plusieurs travaux actuels (pas nécessairement dans le cadre du traitement d'exceptions). Le traitement d'exceptions au moyen des algèbres partielles a essentiellement été initié par les travaux du projet CIP, à Munich ; principalement par M. Broy et M. Wirsing (voir par exemple [BW 82]).
Une approche similaire avait déjà été suggérée par Guttag en 1978 (*préconditions* et *cas d'echec*, [Gut 78]), et utilisée dans [Gau 80] pour la spécification des cas d'erreur dans les compilateurs. Mais aucune formalisation sémantique des préconditions et cas d'echec n'est donnée dans [Gut 78].
Bien que l'approche de type "algèbres partielles" soit très complète, elle ne répond pas à tous nos critères concernant le traitement d'exceptions :

- elle ne permet pas de décrire le comportement des cas exceptionnels (car, du fait que les algèbres sont partielles, rien n'est prévu pour les opérations à résultat érroné)
- pour les mêmes raisons, elle ne permet pas de spécifier de récupérations d'exceptions, en particulier il n'est pas possible de spécifier des opérations *non strictes* (telle qu'un *if_ then_ else_* apte à retourner une valeur *Ok* même si l'un de ses arguments est erroné).

□ Enfin une dernière approche consiste à introduire explicitement des valeurs erronées ou exceptionnelles dans les algèbres. C'est cette approche que nous suivrons de préférence, car elle permet alors de spécifier le traitement des cas erronés ou exceptionnels. Elle a principalement été introduite par Goguen, selon deux méthodes différentes [Gog 77, Gog 78].

- Les *erreur-algèbres* de Goguen [Gog 77].
  Au lieu d'introduire une seule valeur erronée par sorte (*UNDEF*), les erreur-algèbres sont scindées en deux sous-ensembles : les valeurs *Ok* et les valeurs erronées. Le formalisme décrit par [Gog 77], puis par [Pla 82], permet de spécifier de manière assez simple les valeurs que l'on veut "Ok" et celles que l'on veut erronées ; par l'intermédiaire "d'*Ok*-opérations" et "d'*Err*-opérations". Il permet aussi de décrire le comportement des valeurs erronées grâce à l'introduction "d'*Err*-axiomes", et d'avoir une propagation implicite des erreurs, augmentant ainsi le lisibilité des spécifications.
  Par contre, la propagation implicite des erreurs et la sémantique des axiomes (*Ok*-axiomes ou *Err*-axiomes) telles qu'elles sont définies ne permettent pas de spécifier de récupérations d'exceptions. De plus, il n'existe pas, en général, d'objet initial associé à une spécification.
  Cependant, les erreur-algèbres introduites par Goguen ont montré qu'il était possible de fournir des spécifications lisibles avec traitement d'exceptions, et ont introduit

plusieurs concepts repris dans plusieurs formalismes ultérieurs, en particulier [Bid 84] et [GDLE 84] que nous décrivons brièvement plus loin.

- *Surcharge et coercion* de [Gog 78].
  Une autre idée de Goguen, dans [Gog 78], est d'utiliser des méthodes de surcharge et coercion pour traiter le problème des erreurs. Pour chaque sorte *s* d'une spécification, on peut associer deux *sous-sortes* : s_{Ok} et s_{Err} . Dés lors, chaque opération de la signature subit une *surcharge*. Par exemple, l'opération *pred* sur les entiers naturels a pour arité :

  $$pred : \quad NAT \quad \rightarrow \quad NAT \, .$$

  On peut lui associer, par un procédé de *complétion*, les opérations suivantes :

  $$pred: \ NAT \rightarrow NAT$$
  $$pred: \ NAT_{Ok} \rightarrow NAT$$
  $$pred: \ NAT_{Err} \rightarrow NAT$$
  $$pred: \ NAT_{Ok} \rightarrow NAT_{Ok}$$
  ...etc...

  De cette façon, on crée plusieurs opérations modélisant les divers cas d'application de l'opération *pred*.
  Cependant, comme l'a montré [BG 83], il est nécessaire de considérer que ces opérations complétées ne sont que des opérations partielles (i.e. qu'elles ne retournent pas toujours une valeurs, c'est le cas de $pred: NAT_{Ok} \rightarrow NAT_{Ok}$ appliquée à *0*). Or nous avons déjà remarqué que les opérations partielles ne résolvent pas complètement notre propos.
  Bien que [Gog 78] ne résolve pas complètement la question du traitement d'exceptions, ce formalisme présente un intérêt majeur car il résoud par contre les questions de surcharge et coercion. De plus, plusieurs formalisations rigoureuses de cette vision sont actuellement développée (par exemple au CRIN à Nancy, dans le cadre d'OBJ2) ; elles permettent de traiter de nombreux cas d'exceptions, et d'opérations partielles. Toutefois, cette approche conduit à des spécifications assez complexes.

Nous allons maintenant décrire brièvement les formalismes de [Bid 84] et [GDLE 84].

3.3. LES E,R-ALGEBRES

Le formalisme des E,R-algèbres (Erreur,Récupération–algèbres), introduit dans [Bid 84], repose sur l'idée qu'ils faut bien distinguer d'une part la déclaration de ce que l'on veut *Ok* et erroné, et d'autre part la spécification des *Ok*-équations et des *Err*-équations. Il offre une solution au problème de *récupération* d'exceptions non résolu par [Gog 77] en donnant d'une part une définition plus souple de la propagation d'erreurs et d'autre part une meilleur sémantique des *Err*-axiomes. De plus, il permet de déclarer d'une manière structurée les cas d'erreur et les cas de récupérations, ce qui clarifie encore les spécifications.

Ceci est obtenu de la manière suivante.

- □ La signature d'une E,R-spécification contient une signature classique <**S**,**Σ**>, et les E,R-algèbres sur cette signature sont, comme dans [Gog 77], partitionnées en deux : une partie *Ok* et une partie *Err* (plus précisément : chaque support de sorte A_s d'une algèbre est partitionné en $A_{s,Ok}$ et $A_{s,Err}$).

□ Les cas erronés sont de plus déclarés dans la signature au moyen de *déclarations d'erreur*, qui se présentent sous forme d'un ensemble de termes avec variables. Les variables sont éventuellement "signées" :

- les variables "*Ok*", notées $x+$, ne peuvent être instanciées que par des valeurs *Ok* d'une E,R-algèbre
- les variables "erronées", notées $x-$, ne peuvent être instanciées que par des valeurs erronées d'une E,R-algèbre

Une **Σ**-algèbre A valide ces déclarations d'erreurs si et seulement si pour toute instanciation *saine* (c'est à dire compatible avec le "signe" des variables), la valeur obtenue est erronée dans A.

□ Une E,R-signature contient enfin une déclaration de *cas de récupération*, de la même manière que pour les cas d'erreur. Une **Σ**-algèbre A valide ces déclarations de récupération si et seulement si pour toute instanciation saine de ces termes, la valeur obtenue est *Ok* dans A.

□ On distingue les *Ok*-équations et les *Err*-équations comme dans [Gog 77]. La structure "*Ok*" est décrite au moyen d'*Ok*-équations. Ces *Ok*-équations peuvent faire intervenir des termes avec variables "signées". Elle ne s'appliquent que si, après instanciation saine, *les deux* membres sont *Ok* dans l'algèbre.

□ Les *Err*-équations se présentent sous la même forme. Elles décrivent la partie erronée de la structure spécifiée. Elles ne s'appliquent que si, après instanciation saine, *les deux* membres sont erronés dans l'algèbre.

Voici par exemple une spécification des entiers bornés pour le formalisme des E,R-algèbres :

S = { *NAT* }

Σ = { *0* , *succ* , *pred* , *Maxint* , *crash* }
où *0*, *succ* et *pred* ont les arités habituelles, *Maxint* et *crash* sont des constantes de sorte *NAT*.

Cas erronés : *e1 :* *succ(Maxint)*
e2 : *crash*
e3 : *pred(0)*

Cas de récupération : *r1 :* *pred(succ(Maxint))*

Ok-équations : *ok1 :* $pred(succ(n+)) = n+$
ok2 : $Maxint = succ^{Maxint}(0)$

Erreur-équations : *err1 :* $pred(0) = crash$
err2 : $succ(x-) = crash$
err3 : $pred(crash) = crash$

La sémantique de cette spécification peut ici être décrite par $[0...Maxint] \cup \{succ(maxint), crash\}$; avec pour partie *Ok* l'intervalle *[0...Maxint]*, pour partie erronée {*succ(maxint),crash*} ; et *pred* ou *succ* appliqués à *crash* retournent *crash*, *pred* appliqué à *0* retourne *crash*, *pred* appliqué à la valeur

erronée *succ(Maxint)* est récupéré en *Maxint*, *succ* appliqué à *Maxint* retourne la valeur erronée *succ(Maxint)*, et *succ* appliqué à *succ(Maxint)* retourne *crash*.

Remarquons que la première *Ok*-équation s'applique en particulier lorsque $n+ = Maxint$.

$$pred(succ(Maxint)) = Maxint$$

Ceci peut paraître contradictoire avec la propagation implicite d'erreur : puisque *succ(Maxint)* est erroné, *pred(succ(Maxint))* devrait l'être aussi. Cependant, ce dernier terme est déclaré en tant que *récupération* ; il n'est donc plus erroné. Plus précisément, [Bid 84] introduit une nouvelle définition de la propagation implicite d'erreurs : *pour toute opération op de la signature, si l'un des* t_i *est erroné alors* $op(t_1, \ldots, t_n)$ *l'est aussi SAUF si* $op(t_1, \ldots, t_n)$ *correspond à un cas de récupération.* Cette nouvelle définition de la propagation permet donc de spécifier des cas de récupération d'erreurs.

Une remarque s'impose cependant : pour l'exemple que nous avons donné, nous avons pu décrire la sémantique au moyen d'une E,R-algèbre particulière qui est l'E,R-algèbre *initiale* associée à la spécification que nous avons décrite. Malheureusement, comme le montrent [Bid 84] puis [Cap 85], cette algèbre initiale n'existe pas pour toute E,R-spécification, ni même pour toute E,R-signature ; et les conditions suffisantes pour qu'il en existe une sont relativement restrictives. Plus généralement, il n'existe pas toujours de congruence minimale associée à une E,R-spécification. Il en résulte donc que l'on ne peut pas définir d'adjoint à gauche au foncteur d'oubli ; et il n'est donc pas possible de définir des notions similaires à la consistance hiérarchique ou à la suffisante complétude. En particulier, il semble difficile de définir une notion d'*enrichissement* suffisamment générale.

3.4. OPERATIONS 'SAFE' ET 'UNSAFE'

Le traitement d'exceptions proposé par [GDLE 84] repose sur un découpage de la signature en opérations "*safe*" et opérations "*unsafe*". Par exemple, dans le cas des entiers naturels *non bornés*, les opérations *0* et *succ* sont *safe* parce que leur application (sur des valeurs *Ok*) n'induit jamais de nouvelle erreur, tandis que l'opération *pred* est *unsafe* parce que son application à la valeur *0* retourne une valeur (négative) erronée.
Les opérations *safe* ne peuvent retourner que des valeurs *Ok* lorsqu'elles sont appliquées à des valeurs *Ok*, tandis que les opérations *unsafe* peuvent retourner des valeurs erronées, même si leurs arguments sont tous *Ok*.
Il en résulte que l'on dispose toujours d'un "noyau" de termes fermés qui sont nécessairement *Ok* : ceux qui ne sont construits qu'avec des opérations *safe*. Par exemple, tous les termes de la forme $succ^n(0)$ sont nécessairement *Ok*.

La syntaxe des équations repose sur une extension de la notion de variables : un ensemble de variables **V** est découpé en variables *v* telles que *ok(v)=True* et *v+* telles que *ok(v+)=False* (correspondant respectivement aux *x+* et *x-* de [Bid 84] ; on ne dispose pas des variables "générales" *x* de [Bid 84]). Mais les équations ne sont pas découpées en équations *Ok* ou *Err* comme dans [Bid 84], il en résulte que la spécification des cas *Ok* et des cas erronés est complètement mélangée.

L'avantage majeur de ce formalisme est l'existence de congruences minimales associées à une spécification. Il en résulte qu'il existe toujours une algèbre initiale, et que les notions d'enrichissement, de suffisante complétude et de consistance hiérarchique peuvent être redéfinies. Par exemple, l'algèbre initiale de la spécification des entiers naturels, avec l'axiome

$$pred(succ(x+)) = x+$$

contiendra pour éléments *Ok* ceux de la forme $succ^n(0)$ (la partie *Ok* est donc isomorphe à **N**), et les éléments erronés seront tous les surtermes de *pred(0)*.
Remarquons de plus qu'il est possible de spécifier des cas de récupération : il suffit par exemple

d'écrire des axiomes contenant des variables erronées d'un coté et des variables *Ok* de l'autre.

Par contre, le pouvoir d'expression de ce formalisme n'est pas suffisant pour traiter les structures bornées. Par exemple, on ne peux pas spécifier les entiers naturels bornés. En effet, dans le cas des entiers naturels bornés, il n'y a pas d'opération *safe* (sauf *0*), car l'opération *succ* retourne une valeur erronée pour la valeur *Ok Maxint*. Il en résulte que l'algèbre initiale de *NAT* ne contiendrait que la constante *0* dans sa partie *Ok*, ce qui n'est certainement pas ce que l'on veut modéliser.
Cette lacune est majeure, elle nuit au développement de primitives de spécification sérieuses avec traitement d'exceptions. Par exemple, pour pouvoir étendre la notion d'implémentation abstraite au traitement d'exceptions de manière crédible, il faut au minimum pouvoir modéliser des implémentations telles que celle des files bornées par des tableaux bornés. L'intérêt du traitement d'exceptions, dans ce cas, étant justement l'étude du passage de bornes entre les tableaux et les files (ainsi que des rapports entre leurs messages d'erreur respectifs, tels que *out-of-range* et *stack-too-large*).

4. RECAPITULATION

En résumé, parmi tous les formalismes que nous venons de citer, le seul formalisme algébrique (sans opérations partielles) qui possède un pouvoir d'expression assez puissant pour notre propos est [Bid 84]. Mais la sémantique n'en est pas assez puissante car elle ne fournit pas de congruences minimales, donc pas d'algèbres initiales et pas de notion d'enrichissement, de suffisante complétude ou de consistance hiérarchique. Le formalisme de [GDLE 84] posséde une sémantique "agréable" (objet initial, congruence minimale associée à toute spécification, donc existence de spécifications hiérarchiques) ; mais par contre il présente des faiblesses de pouvoir d'expression (impossibilité de modéliser des structures bornées, distinction peu claire des traitements "normaux" et des traitements exceptionnels).

De plus, une critique majeure s'applique à tous les formalismes existants de traitement d'exceptions : aucun ne modélise la notion de "message d'erreur". Or il semble clair que, quitte à construire un formalisme de traitement d'exceptions, autant offrir aussi un moyen de diagnostiquer les diverses exceptions.

En conclusion, il apparaît clairement que pour traiter complètement les exceptions dans un cadre algébrique, il faut d'abord introduire un nouveau concept pour modéliser "proprement" la notion de message d'erreur. Un tel concept n'avait encore été proposé dans aucun formalisme de traitement d'exceptions ; il permettra pourtant de spécifier le comportement des cas exceptionnels avec beaucoup plus de souplesse (rapport entres les valeurs erronées, récupération des cas exceptionnels en fonction du type d'exception levée). De plus, on constate que l'idée, introduite par [Bid 84], de déclarer les cas *Ok* et exceptionnels explicitement dans la syntaxe conduit à des spécifications plus facilement lisibles ; ainsi que le l'idée, introduite par [Gog 77], de scinder en deux les ensembles d'équations (une partie traitant les cas *Ok* et une partie traitant les cas exceptionnels).

Nous avons aussi remarqué que tout formalisme de traitement (algébrique) d'exceptions doit être soumis aux contraintes sémantiques de propagation implicite d'exceptions et d'erreurs, et offrir des notions hiérarchiques (telles que la consistance hiérarchique et la suffisante complétude), ceci afin de pouvoir étendre raisonnablement les primitives usuelles de structuration des spécifications.

Chapitre II :

Vision intuitive

de notre formalisme

Dans ce chapitre, nous allons examiner une à une les caractéristiques utiles au traitement d'exceptions (motivées en section 2 du chapitre précédent), et nous allons argumenter les choix faits dans notre formalisme des exception-algèbres pour y répondre. Les caractéristiques principales demandées pour le traitement d'exceptions sont :

- □ L'aptitude à modéliser simplement la notion de *messages d'erreur*.
- □ L'aptitude à *déclarer* ce que l'on veut *Ok* (ou ce que l'on veut exceptionnel) d'une manière structurée, simple, mais assez puissante (cf. les structures bornées).
- □ La possibilité de spécifier quand doit intervenir tel ou tel message d'erreur, c'est à dire "étiqueter" certains termes par un message d'erreur, ou encore *diagnostiquer les divers types d'exceptions*.
- □ La possiblité de décrire le *comportement des valeurs exceptionnelles* : les dépendances entre les diverses valeurs erronées, ou la récupération de certaines valeurs exceptionnelles.
- □ Une *propagation implicite* des exceptions et des erreurs, pour ne pas avoir à spécifier explicitement la propagation des cas exceptionnels (ou des cas erronés), ce qui résulterait en un grand nombre d'axiomes.
- □ Enfin l'existence d'enrichissements, munis de critères tels que la consistance hiérarchique ou la suffisante complétude, d'une manière qui prenne en compte les cas exceptionnels ou erronés. Ceci impose une sémantique où les foncteurs adjoints à gauche aux foncteurs d'oubli existent, donc une sémantique où la notion de congruence minimale existe. En particulier, il doit exister une algèbre initiale.

On voit que les quatre premiers critères sont essentiellement des facilités qui doivent être offertes à l'usager au niveau *syntaxique*. Les deux derniers critères sont plutôt des contraintes *sémantiques* qui doivent être vérifiées afin de rendre assez puissant un formalisme de traitement d'exceptions (i.e. lui donner les même potentialités que le formalisme classique des types abstraits algébriques, en particulier au niveau d'extensions ultérieures telles que l'implémentation abstraite, la paramétrisation, les

spécifications incomplètes ...).

1. L'ASPECT SYNTAXIQUE

1.1. MODELISATION DES MESSAGES D'ERREUR

Dans le formalisme classique des types abstraits algébriques, la syntaxe est donnée par un triplet <**S**,**Σ**,**A**>. La signature <**S**,**Σ**> décrit les éléments de base des algèbres : l'ensemble des sortes **S** décrit le découpage de chaque algèbre (par exemple, un ensemble de booléens, un ensemble d'entiers, un ensemble de piles d'entiers ...), et l'ensemble des opérations **Σ** décrit les fonctions travaillant sur ces supports de sortes. L'ensemble d'axiomes **A** (généralement un ensemble d'équations ou un ensemble d'équations condionnelles positives) spécifie quant-à lui les propriétés que doivent valider les opérations (et les sortes) les unes par rapport aux autres.

Une algèbre modélise donc d'une part un ensemble de valeurs typées (au moyen de ses ensembles supports), et d'autre part l'action des opérations sur ces valeurs (au moyen des fonctions associées aux opérations). Nous devons maintenant lui adjoindre des "diagnostics d'erreur". La remarque que nous voulons faire ici est qu'un message d'erreur ne peut être modélisé valablement ni par une sorte, ni par une opération :

- □ Un message d'erreur ne peut être modélisé par une sorte.
  Par exemple, lorsque l'on applique l'opération *pred* à la valeur *0*, le résultat *pred(0)* est exceptionnel, mais ce n'est pas pour autant qu'il change de sorte. C'est encore un entier naturel (un entier naturel exceptionnel) ; ce fait est spécialement évident lorsque l'on pense à d'éventuels cas de récupération. Par exemple, si l'on veut récupérer le terme *succ(pred(0))* en *0*, il faut bien que l'on puisse appliquer l'opération *succ* à *pred(0)*, donc que *pred(0)* soit de sorte *NAT*. [Comme nous l'avons déjà motivé, nous ne choisissons pas ici l'approche consistant à surcharger l'opération *pred* , afin d'obtenir des spécifications plus simples].

- □ Un message d'erreur ne doit pas être modélisé par une opération.
  Par exemple, le message d'erreur attaché à chaque valeur négative pourrait être "*NEGATIVE-VALUE*". Si l'on cherche à le modéliser par une opération, on devrait ajouter l'opération constante *NEGATIVE-VALUE* (de sorte *NAT*). Dans ce cas, pour étiqueter toutes les valeurs négatives par *NEGATIVE-VALUE*, on devrait poser les axiomes :
  $$pred(0) = NEGATIVE\text{-}VALUE$$
  $$pred(NEGATIVE\text{-}VALUE) = NEGATIVE\text{-}VALUE\ .$$
  Par conséquent, on aurait amalgamé tous les termes de la forme $pred^n(0)$ sur *NEGATIVE-VALUE*. Ce qui nuit au pouvoir d'expression : on pourrait avoir envie d'autoriser une récupération de *succ(pred(0))*, mais faire tomber *pred(pred(0))* dans un "crash" irrécupérable (sans pour autant changer le disgnostic *NEGATIVE-VALUE*). Sans compter que l'on risque de créer alors des inconsistances telles que *succ(pred(pred(0))) = 0*.

Remarquons aussi qu'un même message d'erreur peut parfois étiqueter deux valeurs de sortes différentes. Par exemple, dans le cas de tableaux bornés d'entiers, on peut vouloir associer le message *OUT-OF-RANGE* à des termes comme *t[i]:=x* et *t[i]* (où le rang *i* est hors des bornes du tableau) ; ces deux termes sont respectivement de sortes *ARRAY* et *NAT*.

On constate donc que le message d'erreur associé à certains termes exceptionnels doit être indépendant de la valeur de ce terme. En fait, diagnostiquer des cas exceptionnels relève plutôt d'un

"*étiquetage*" de termes par des messages d'erreur. Nous modélisons ceci dans notre formalisme par le biais d'*étiquettes d'exceptions*. Ces étiquettes d'exception n'étant ni des sortes, ni des opérations, nous allons étendre la notion de signature en lui ajoutant un ensemble "d'étiquettes d'exceptions" (tels que *NEGATIVE-VALUE* ou *OUT-OF-RANGE*). Une exception-signature s'exprimera donc par (**S**,**Σ**,**L**) où **L** est une ensemble d'étiquettes d'exception (**L** pour *exception Labels*, en bon français ...).

1.2. DECLARATION DES VALEURS *Ok*

Il s'agit maintenant de déclarer ce que l'on veut *Ok* et ce que l'on veut exceptionnel. Nous avons vu que la notion de *déclarations* introduite par [Bid 84] conduit à des spécifications plus structurées, où le traitement des cas exceptionnels est clairement séparé du traitement des cas *Ok*. Nous ne suivront pas la même méthode que [Bid 84] pour effectuer ces déclarations, mais nous conserverons l'idée consistant à utiliser une *déclaration* dans la syntaxe (pour caractériser les cas *Ok*).

Remarquons qu'il n'est pas nécessaire de déclarer les cas exceptionnels : il est clair qu'un terme dont la valeur n'est pas *Ok* sera exceptionnel. Par contre, notons que la réciproque n'est pas vraie. Plus précisément : nous disposons déjà de la notion d'étiquette d'exception, et on pourrait penser qu'elle suffit à distinguer les cas *Ok* des cas erronés (un terme qui n'est pas étiqueté serait automatiquement *Ok*). En fait, les étiquettes d'exception ne suffisent pas pour déterminer ni les cas *Ok*, ni les cas erronés. En voici les raisons intuitives :

- Ce n'est pas parce qu'un terme (ou l'un de ses sous-termes) est étiqueté par une étiquette d'exception qu'il est erroné. Ceci est dû au fait que l'on veut pouvoir récupérer certains termes exceptionnels. Par exemple, le terme *succ(pred(0))* contient le sous-terme *pred(0)* qui est étiqueté par *NEGATIVE-VALUE*, mais il n'empêche que l'on peut vouloir (dans certains cas) le récupérer sur la valeur *0*. De la même façon, même si le terme *t[i]* (avec l'indice *i* hors des bornes) est étiqueté par *OUT-OF-RANGE*, on peut fort bien, dans certains cas, vouloir le récupérer (sur *0* par exemple). De tels termes sont alors étiquetés par une étiquette d'exception, mais leur valeur est pourtant *Ok* (par récupération).

- Ce n'est pas parce qu'un terme n'est étiqueté par aucun message d'erreur qu'il doit nécessairement être *Ok*. En effet, rappelons que nous voulons bâtir un formalisme de traitement d'exceptions pour lequel on puisse étendre la plupart des acquis des types abstraits algébriques classiques. En particulier, nous aimerions pouvoir étendre la notion de spécifications incomplètes. L'exemple le plus simple de spécification incomplète est celui du type *ensemble* avec une opération *choose*. Considérons un ensemble *{a,b}* tel que *a* soit *Ok* et *b* soit erroné ; le terme *choose({a,b})* ne peut pas être étiqueté par l'étiquette de *b*, car il est possible que *choose({a,b})* soit égal à *a* . Par défaut on ne peut donc lui associer aucune étiquette d'exception, mais ce n'est pas pour autant qu'il est *Ok*.
  Plus généralement, de même que le formalisme classique n'impose pas d'office qu'une spécification soit complète (par exemple par rapport à une quelconque forme normale), nous n'avons aucune raison d'imposer a priori que nos spécifications soient complètes par rapport aux étiquettes d'exceptions et aux valeurs *Ok*. Dans le même ordre d'idée, contrairement aux formalisme algébriques précedemment développés, nous n'imposerons pas qu'une exception-algèbre soit toujours partitionnée en valeurs *Ok* ou erronées. Il pourra y avoir des valeurs qui ne soient ni *Ok* ni erronées. Par soucis de généralité, mieux vaut laisser ce genre de propriétés à une notion élargie de suffisante complétude plutôt que de "cabler" cette propriété dans la sémantique de départ.

Le seul étiquetage d'exceptions n'est donc pas suffisant pour déterminer quelles sont les valeurs *Ok* et les valeurs erronées.
Par contre, si l'on connait les valeurs *Ok* et les étiquetages d'exception, alors on peut déterminer les valeurs erronées. Il suffit de se servir de la propagation implicite des erreurs, sauf si elles sont récupérées. Les valeurs erronées seront déterminées par la méthode récursive suivante :

- Un terme étiqueté par une étiquette d'exception est erroné, sauf si sa valeur est une valeur *Ok* (auquel cas c'est qu'il a été récupéré).
- Pour tout terme de la forme $t = op(t_1 \cdots t_n)$, si l'un des t_i est erroné alors t est aussi erroné, sauf si sa valeur est *Ok*. Ce qui traduit la propagation implicite des erreurs, sauf en cas de récupération.

On reconnaît là la méthode introduite par [Bid 84].

Par conséquent, connaissant l'étiquetage d'exception des termes, il suffit de déclarer les valeurs *Ok* pour connaître les valeurs erronées. Nous n'utiliserons donc des déclarations que pour spécifier les valeurs *Ok*.
Comme nous l'avons déjà remarqué, il est clairement impossible de déclarer *tous* les termes *Ok*. Il nous faut donc les déclarer sous une forme "standard" (par exemple sous forme normale). La méthode que nous suivront est une déclaration récursive des "formes *Ok*". Intuitivement, il s'agira dans la plupart des exemples d'un sous-ensemble des formes normales : les formes normales dont la valeur est *Ok*.

Voici par exemple comment on pourrait déclarer récursivement l'ensemble des formes *Ok* des entiers naturels bornés :

0 est une forme Ok

Si n est une forme Ok et si n<Maxint , alors succ(n) est une forme Ok

(où *Maxint* est la borne supérieure)
Une autre possibilité est la déclaration suivante, qui présente l'avantage de ne pas utiliser l'opération "<" :

$succ^{Maxint}(0)$ *est une forme Ok*

Si succ(n) est une forme Ok, alors n aussi

1.3. ETIQUETAGE D'EXCEPTIONS

Au point où nous en sommes, nous modélisons les messages d'erreur au moyen d'un ensemble d'*étiquettes d'exception*, et nous caractérisons les valeurs *Ok* au moyen d'une *déclaration de formes Ok*. De manière évidente, le comportement des cas "normaux" (*Ok*) sera spécifié au moyen d'*Ok-axiomes*.
Nous n'avons pas encore expliqué quels sont les rapports entre les valeurs d'une exception-algèbre et notre nouveau concept d'étiquette d'exception ; autrement dit, il nous faut un outil pour étiqueter certaines valeurs des algèbres avec les étiquettes d'exception. Ceci se fait très simplement : au même titre que les rapports entre les opérations de la signature sont spécifiées par des axiomes sur la signature, l'étiquetage se fera au moyens d'*axiomes d'etiquetage*. Ces axiomes spécifient quand une valeur de l'algèbre doit appartenir à l'ensemble des valeurs étiquetées par un élément de **L** donné. Nous utiliserons pour cela des axiomes conditionnels, sous une forme telle que celle-ci :

$$pred(0) \in NEGATIVE\text{-}VALUE$$

$$n \in NEGATIVE\text{-}VALUE \implies pred(n) \in NEGATIVE\text{-}VALUE$$

ou encore, dans le cas des tableaux :

$$i > Maxrange = True \implies t[i]:=x \in OUT\text{-}OF\text{-}RANGE$$

$$i \in NEGATIVE\text{-}VALUE \quad \Rightarrow \quad t[i]:=x \in OUT\text{-}OF\text{-}RANGE$$
$$i > Maxrange = True \quad \Rightarrow \quad t[i] \in OUT\text{-}OF\text{-}RANGE$$
$$i \in NEGATIVE\text{-}VALUE \quad \Rightarrow \quad t[i] \in OUT\text{-}OF\text{-}RANGE$$

On voit que l'étiquetage des exceptions se fait de manière naturelle avec de tels axiomes.

1.4. TRAITEMENT DES VALEURS EXCEPTIONNELLES

Il ne nous reste plus qu'à décrire le comportement des valeurs exceptionnelles (amalgame de plusieurs exceptions ou récupération de certaines valeurs exceptionnelles). Le seul fait d'avoir un étiquetage des valeurs exceptionnelles offre une souplesse d'expression très grande.
Nous spécifierons le traitement des valeurs exceptionnelles au moyen d'axiomes conditionnels, mais ces axiomes seront "généralisés" en ce sens que l'on pourra aussi utiliser des prémisses relatives aux étiquetages. Par exemple, si l'on veut récupérer sur *0* toutes les valeurs issues d'accès "*OUT-OF-RANGE*" à des tableaux bornés, il suffit de spécifier l'axiome suivant :

$$n \in OUT\text{-}OF\text{-}RANGE \quad \Rightarrow \quad n = 0$$

D'un autre côté, si l'on désire amalgamer toutes les valeurs négatives sur une constante "*crash*", il suffit de spécifier :

$$n \in NEGATIVE\text{-}VALUE \quad \Rightarrow \quad n = crash$$

On peut aussi faire un traitement d'exceptions plus fin : on peut par exemple amalgamer sur *crash* les valeurs négatives inférieures ou égales à *–2,* et autoriser la récupération sur *0* du terme exceptionnel *succ(–1)*. Il suffit de spécifier :

$$succ(pred(0)) = 0$$
$$n \in NEGATIVE\text{-}VALUE \;\; et \;\; eq(n,pred(0)) = False \quad \Rightarrow \quad n = crash$$

1.5. RECAPITULATION DE LA SYNTAXE

Nos critères de traitement d'exceptions se traduiront syntaxiquement comme suit :

- □ Les divers diagnostics d'exception sont modélisés par un ensemble d'étiquettes d'exception ajouté dans la signature (**L**, traduisant un ensemble de "messages d'erreur").
- □ La déclaration des valeurs que l'on veut *Ok* se fait de manière simple grâce à une déclaration de "formes *Ok*".
- □ La spécification de la structure "normale" (i.e. *Ok*) des exception-algèbres est faite au moyen d'un ensemble d'*Ok*-axiomes.
- □ Les étiquetages des valeurs exceptionnelles par les éléments de **L** se spécifie simplement au moyen d'axiomes d'étiquetage.
- □ Enfin le traitement des cas exceptionnels est spécifié au moyen d'axiomes conditionnels "généralisés" aptes à prendre en compte les étiquetages d'exception dans leurs premisses.

2. L'ASPECT SEMANTIQUE

2.1. TERMES EXCEPTIONNELS ET VALEURS ERRONEES

Jusqu'à maintenant, nous avons utilisé les mots "exception" et "erreur" sans expliquer réellement quelle est la différence entre les deux. En fait, aucun formalisme précédemment développé ne différencie ces deux concepts ; l'une des idées majeures de notre formalisme est justement qu'il faut bien différencier les *termes exceptionnels* des *valeurs erronées*.

Intuitivement, un terme exceptionnel est un terme auquel on ne peut pas donner une valeur *Ok* via le traitement "normal" de la structure spécifiée. Autrement dit, un terme exceptionnel est un terme qui ne peut pas être amalgamé avec une valeur *Ok* au moyen des *Ok*-axiomes (rappelons que les *Ok*-axiomes décrivent le traitement de la partie "purement *Ok*" des exception-algèbres, et que l'on caractérise les valeurs *Ok* par le biais de formes *Ok*).
Remarquons tout de suite qu'un terme exceptionnel peut néanmoins avoir une valeur *Ok* : il suffit pour cela qu'il soit récupéré ultérieurement (grâce aux axiomes généralisés qui décrivent le traitement des cas exceptionnels).

Par ailleurs, nous avons déjà expliqué intuitivement ce qu'est une valeur erronée (section 1.2) : c'est une valeur étiquetée par un "message d'erreur", ou issue par propagation d'une valeur étiquetée, et qui n'a pas été récupérée.

Voici un exemple simple, mais néanmoins important, qui montre pourquoi il faut distinguer les deux notions de termes exceptionnels et de valeurs erronées.
Supposons que l'on veuille spécifier les entiers naturels bornés. Soit *Maxint* la borne (*Ok*) supérieure. On peut fort bien vouloir récupérer toutes les valeurs plus grandes que *Maxint* sur *Maxint* lui-même c'est à dire poser l'axiome de récupération suivant :

$$succ(Maxint) = Maxint$$

D'un autre côté, les *Ok*-axiomes, qui décrivent la structure "normale" des entiers naturels, vont bien sûr contenir l'axiome suivant :

$$pred(succ(n)) = n\ .$$

La question est donc maintenant de donner une sémantique à ces axiomes.
Si l'on donne une sémantique similaire à toutes celles que nous avons citées dans le chapitre précédent, un *Ok*-axiome s'appliquera (au minimum) chaque fois que les deux membres sont *Ok*. Mais ici, la valeur *pred(succ(Maxint))* est *Ok*, car *succ(Maxint)* est récupéré sur *Maxint*, qui est lui même *Ok* ; il en résulte que *pred(succ(Maxint))* est amalgamé avec *Maxint* via notre *Ok*-axiome. Mais l'axiome de récupération conduit par contre à *succ(Maxint)=Maxint* si bien que l'on obtient l'inconsistance suivante :

$$pred(Maxint) = Maxint\ .$$

En fait, il est clair que l'on ne voudrait pas que l'*Ok*-axiome s'applique dans ce cas. Bien que le terme *pred(succ(Maxint))* possède une valeur *Ok* (parce qu'il a été récupéré), il n'en nécessite pas moins un "traitement d'exception". C'est ici qu'intervient notre distinction : le *terme succ(Maxint)* est exceptionnel, mais sa *valeur* est *Ok* (par récupération).

Le fait que les *termes exceptionnels* n'ont pas nécessairement des *valeurs erronées*, est simple à prendre en compte dans l'algèbre des termes fermés. En effet, il suffit d'appliquer les *Ok*-axiomes *avant* d'appliquer les axiomes généralisés (avant de traiter la récupération). Les *Ok*-axiomes agirons alors en toute ignorance des éventuelles récupérations ultérieures, évitant ainsi les inconsistances telles que celle que nous venons de décrire.
Il peut paraître plus complexe de résoudre ce point dans une exception-algèbre quelconque puisque les amalgames relatifs aux récupérations y sont déjà faits. Il nous faut donc établir une nouvelle notion de *validation* des *Ok*-axiomes. La solution que nous proposons est somme toute assez simple :

pour toute exception-algèbre A , la validation des *Ok*-axiomes est définie grâce à l'algèbre libre des termes avec variables dans A (notée $T_{\Sigma(A)}$), au lieu d'être directement définie dans A . Nous pouvons alors de nouveau faire la distinction entre les *termes exceptionnels* de $T_{\Sigma(A)}$ et leur *valeur* dans A, qui n'est pas nécessairement erronée. Le "transfert" des critères de validations est alors simplement assuré par le morphisme canonique d'*évaluation* entre $T_{\Sigma(A)}$ et A.

2.2. LES PROPAGATIONS IMPLICITES

Comme déjà expliqué, la propagation des exceptions et des erreurs doit être incluse dans la sémantique des exception-algèbres afin d'éviter de longues listes d'axiomes de propagation dans les spécifications.

Nous avons déjà décrit la propagation implicite des valeurs erronées :

- Une étiquette d'exception attaché à une valeur la rend erronée, sauf si cette valeur est *Ok*, auquel cas c'est qu'elle est récupérée.
- L'application de toute opération de la signature propage les erreurs, sauf si le résultat est *Ok*, auquel cas nous sommes dans une situation de récupération.

Il nous faut maintenant décrire la propagation implicite des termes exceptionnels. Comme nous l'avons remarqué, les termes exceptionnels ne doivent pas être traités par les *Ok*-axiomes, même s'ils sont récupérés ultérieurement. Par conséquent, la propagation implicite des termes exceptionnels est plus simple que celle des valeurs erronées : si un terme est exceptionnel, alors tous ses surtermes le sont aussi (sans aucune considération de récupération cette fois). Il est donc difficile de décrire la propagation implicite des termes exceptionnels directement dans une algèbre, puisque les éléments des algèbres sont des *valeurs* et non des *termes*.
Pour notre formalisme, la propagation implicite des termes exceptionnels sera traduite dans la notion de validation des *Ok*-axiomes : on imposera une évaluation des *Ok*-axiomes "par les termes les plus profonds d'abord". Une telle évaluation est en fait équivalente à une propagation implicite des termes exceptionnels. En effet, si un terme est exceptionnel, alors les *Ok*-axiomes ne pourront pas lui fournir une valeur *Ok*. Si l'on considère maintenant un surterme quelconque de ce terme exceptionnel, aucun *Ok*-axiome ne pourra lui être appliqué, puisque l'évaluation devrait *d'abord* fournir une valeur *Ok* à tous ses sous-termes. Nous reviendrons plus précisément sur cet aspect dans le chapitre IV (section 4, second paragraphe de la remarque 3).

2.3. LES OUTILS HIERARCHIQUES

Un formalisme algébrique offrant toutes les facilités de traitement d'exceptions ne suffit pas ; encore faut-il qu'il offre les outils qui font l'intérêt des types abstraits algébriques.

Le premier concept de base des types abstraits algébriques est celui de *congruences*. Ce sont les congruences qui permettent des raisonnements simples sur les propriétés sémantiques (car il est plus simple de manipuler des classes d'équivalences que des équivalences d'algorithmes). Il est donc de première nécessité de disposer d'une notion étendue de congruence prenant en compte le traitement d'exceptions.

Ensuite, viennent les notions *hiérarchiques* (essentiellement la consistance hiérarchique et la suffisante complétude). Nous avons en particulier eu l'occasion de constater leur utilité dans la partie A de cette thèse, à propos de l'implémentation abstraite. Rappelons que ces notions sont définies au

moyen du morphisme d'adjonction associé aux foncteurs d'oubli et de synthèse. Il est donc important de pouvoir définir le foncteur de synthèse qui est adjoint à gauche au foncteur d'oubli.
La construction du foncteur de synthèse repose sur la notion de *congruence minimale* (cf. [Ber 86]). Nous devons donc bâtir la sémantique des exception-algèbres de telle sorte qu'il existe toujours une congruence minimale associée à une spécification. Les notions de consistance hiérarchique et suffisante complétude peuvent alors être redéfinies de manière à prendre en compte le traitement d'exception.

L'existence d'une congruence minimale aura un corollaire immédiat : pour toute exception-spécification, il existe une exception-algèbre initiale associée. De même, l'existence de congruences minimales permettra d'étendre la plupart des primitives de structuration des spécifications au cas avec traitement d'exceptions. En particulier, nous montrerons que l'on étend notre notion d'implémentation abstraite sans difficulté.

Chapitre III :

Exception-spécifications

Ce chapitre décrit la syntaxe de notre formalisme de types abstraits algébriques avec traitement d'exceptions. Suivant les choix motivés dans le chapitre précédent (chapitre II, section 1), une *exception-spécification* sera définie par :

$$\boxed{\textbf{SPEC} \;=\; < \textbf{S} , \boldsymbol{\Sigma} , \textbf{L} \;,\; \textbf{Ok-Frm} \;,\; \textbf{Ok-Ax} \;,\; \textbf{Lbl-Ax} \;,\; \textbf{Gen-Ax} >}$$

où :

1. $< \mathbf{S} , \boldsymbol{\Sigma} , \mathbf{L} >$ est une *exception-signature*.
2. **Ok-Frm** est une *déclaration de formes Ok* ("Ok-forms").
3. **Ok-Ax** est un ensemble d'*Ok-axiomes*.
4. **Lbl-Ax** est un ensemble d'*axiomes d'étiquetage* ("labelling-axioms").
5. **Gen-Ax** est un ensemble d'*axiomes généralisés* ("generalized-axioms").

Nous allons examiner successivement ces cinq points, et développer au fur et à mesure l'exemple des entiers naturels bornés. D'autres exemples d'exception-spécifications sont donnés en annexe 3.

1. EXCEPTION-SIGNATURES

Une *exception-signature* est une signature classique munie d'un ensemble d'étiquettes d'exception :

Définition 1 :
Une *exception-signature* $\boldsymbol{\Sigma}$-*exc* = $<\mathbf{S},\boldsymbol{\Sigma},\mathbf{L}>$ est définie par :

- □ **S** est un ensemble fini de *sortes*
- □ $\boldsymbol{\Sigma}$ est une ensemble fini d'*opérations* à arité dans **S** (c'est à dire qu'un mot non-vide sur **S** est associé à chaque nom d'opération de $\boldsymbol{\Sigma}$)
- □ **L** est un ensemble fini d'*étiquettes d'exception*. Pour des raisons que nous expliciterons ultérieurement, on impose qu'aucune étiquette de **L** ne soit notée "*Ok*" ou "*err*".

Exemple 1 :
L'exception-signature des *entiers naturels bornés* peut être spécifiée comme suit :

- □ **S** = { *BOOL* , *NAT* }
- □ **Σ** = { *true, false, 0, Maxint, succ_* , _ <_ , *pred_* , _ +_ , _ −_ , _ ×_ , _ *div_* }
  avec les arités usuelles évidentes
- □ **L** = { *NEGATIVE-NUMBER , TOO-LARGE-NUMBER , DIV-BY-0-ERROR* }
  L'étiquette *NEGATIVE-NUMBER* étiquettera clairement les valeurs négatives, *TOO-LARGE-NUMBER* celles strictement plus grandes que la borne supérieure *Maxint*, et *DIV-BY-0-ERROR* étiquettera les cas de division par *0*.

La seule différence entre une signature classique et une exception-signature est l'ensemble des étiquettes d'exception, qui modélisent la notion de messages d'erreur. Les étiquettes d'exception ne sont ni des sortes ni des opérations de la signature. Au niveau de l'exception-signature, on ne spécifie aucune condition sur les étiquettes d'exception, sinon qu'aucune d'elles ne peut être égale à *Ok* ou à *err* . Ceci est simplement dû au fait que l'on se servira de l'étiquette "*Ok*" pour les valeurs *Ok* d'une exception-algèbre, et de la notation "*err*" pour les valeurs erronées d'une exception-algèbre.

2. LES FORMES *Ok*

Cette partie de la syntaxe sert à déclarer quelles sont les valeurs que l'on veut *Ok*. Par exemple pour déclarer les formes *Ok* relatives aux entiers naturels bornés, on est conduit de manière naturelle à utiliser une méthode récursive :

$$0 \text{ est une forme Ok}$$

$$\left.\begin{array}{r} Si\ \ n\ \ est\ une\ forme\ Ok \\ et\ si\ \ n<Maxint\ \ est\ vrai \end{array}\right\}\ alors\ \ succ(n)\ \ est\ une\ forme\ Ok$$

De cette façon, les formes *Ok* des naturels bornés seront les termes de la forme $succ^i(0)$ où $0 \le i \le Maxint$.
On peut ainsi caractériser assez facilement les valeurs que l'on veut *Ok* dans une algèbre (souvent sous forme canonique, par exemple les termes de la forme $succ^n(0)$).

Définition 2 :
Etant donnée une exception-signature **Σ**-*exc* = <**S**,**Σ**,**L**> , une *déclaration de formes Ok*, notée **Ok-Frm**, est un ensemble fini de déclarations élémentaires de la forme suivante :

$$\left.\begin{array}{r} t_1 \in \text{Ok-Frm} \ \wedge \cdots \wedge\ t_m \in \text{Ok-Frm} \\ \wedge\ \ x_1 \in l_1\ \wedge \cdots \wedge\ x_n \in l_n \\ \wedge\ \ v_1 = w_1\ \wedge \cdots \wedge\ v_p = w_p \end{array}\right\} \quad \Rightarrow \quad t \in \text{Ok-Frm}$$

où les t_i, x_j, v_k, w_k, et t sont des **Σ**-termes avec variables ([1]) ; les l_j sont des étiquettes appartenant à $\mathbf{L} \cup \{Ok\}$; évidemment, v_j et w_j doivent être de même sorte pour chaque j. (m, n ou p peuvent être égaux à *0*).

(1) nos variables sont des variables au sens courant, elles ne sont pas "signées", contrairement à [Bid 84] ou [GDLE 84].

Cette définition traduit formellement des caractérisations récursives comme celle décrite précédemment. On peut utiliser des prémisses portant sur les étiquetages. En pratique, les prémisses de la forme "$x_k \in l_k$" seront peu utilisées, sauf lorsque l_k égale *Ok*. Ici, remarquons que l'étiquette *Ok* n'a pas encore été citée. C'est une étiquette "prédéfinie". En fait, son interprétation sémantique intuitive est claire : $x \in Ok$ sera vrai si et seulement si la valeur de x est *Ok* dans l'exception-algèbre considérée. Nous reviendrons évidemment sur ce point lorsque nous définirons la sémantique associée à une telle déclaration.

Exemples 2 :
On peut déclarer l'ensemble des formes *Ok* de *NAT* comme suit :

$$\begin{array}{rcl} & & 0 \in \text{Ok-Frm} \\ n \in \text{Ok-Frm} \ \wedge\ n < Maxint = True & \Rightarrow & succ(n) \in \text{Ok-Frm} \end{array}$$

Une remarque cependant : dans la déclaration précédente, nous faisons usage de l'opération _ <_ . Mais on pourrait fort bien ne pas désirer spécifier cette opération dans une spécification des entiers bornés.
Il existe un moyen assez simple d'éviter l'usage d'un prédicat booléen pour déclarer les formes *Ok* : il suffit de déclarer une caractérisation récursive "à l'envers" :

$$\begin{array}{rcl} & & succ^{Maxint}(0) \in \text{Ok-Frm} \\ succ(n) \in \text{Ok-Frm} & \Rightarrow & n \in \text{Ok-Frm} \end{array}$$

Les formes *Ok* sont encore les termes de la forme $succ^i(0)$ tels que $0 \le i \le Maxint$.

Maintenant que nous disposons d'un moyen pour caractériser les valeurs *Ok* dans une exception-spécification, nous décrivons les *Ok*-axiomes.

3. LES OK-AXIOMES

Les *Ok*-axiomes sont utilisés pour spécifier les cas purement *Ok*. "Purement" signifie "sans prendre en compte la récupération" ; nous avons montré au chapitre précédent pourquoi il est nécessaire d'exclure les cas exceptionnels du traitement des *Ok*-axiomes (même s'ils sont récupérés). Nos *Ok*-axiomes sont de simples axiomes conditionnels positifs :

Définition 3 :
Etant donnée une exception-signature $\mathbf{\Sigma}\text{-}exc = \langle \mathbf{S}, \mathbf{\Sigma}, \mathbf{L} \rangle$, on note **Ok-Ax** un ensemble d'*Ok-axiomes* de la forme suivante :

$$[\, v_1 = w_1 \ \wedge \cdots \wedge\ v_n = w_n \,] \quad \Rightarrow \quad v_{n+1} = w_{n+1}$$

où les v_i et w_i sont des $\mathbf{\Sigma}$-termes avec variables, v_i et w_i de même sorte pour chaque i. (n peut être égal à *0*).

[En fait, on peut aussi ajouter des prémisses de la forme "$x_j \in l_j$" sans nuire aux résultats que nous énoncerons ; mais ceci est peu utile dans les exemples, puisque **Ok-Ax** ne traite que les cas "normaux"].

Par exemple, les *Ok*-axiomes relatifs aux entiers naturels bornés, avec l'exception-signature donnée en exemple 1, sont classiques :

$$
\begin{array}{rcl}
Maxint & = & succ^{Maxint}(0) \\
n < 0 & = & False \\
0 < succ(m) & = & True \\
succ(n) < succ(m) & = & n < m \\
pred(succ(n)) & = & n \\
n + 0 & = & n \\
n + succ(m) & = & succ(n) + m \\
n - 0 & = & n \\
n - succ(m) & = & pred(n) - m \\
n \times 0 & = & 0 \\
n \times succ(m) & = & (n \times m) + n \\
r < m = True \quad \Rightarrow \quad ((n \times m)+r)\ div\ m & = & n
\end{array}
$$

Ouvrons une petite parenthèse : noter que même si cette spécification ne fournit pas un système de réécriture pour calculer la division euclidienne, elle n'en est pas moins algébriquement correcte (elle définit *div* de manière suffisamment complète). Si l'on préfère, on peut bien sûr spécifier la division comme suit :

$$
\begin{array}{lcll}
n < m = True & \Rightarrow & n\ div\ m & = & 0 \\
n < m = False & \Rightarrow & n\ div\ m & = & succ((n-m)\ div\ m)
\end{array}
$$

Naturellement, prouver que ces deux spécifications de la division ont la même sémantique relève de méthodes similaires à l'implémentation abstraite. <fin de la parenthèse>
Remarquons aussi que le second *Ok*-axiome "*n<0 = False*" est parfaitement valide malgré l'opération *pred*, car si *n* est négatif alors il est exceptionnel, donc cet *Ok*-axiome ne s'applique pas.

Maintenant que nous avons les outils syntaxiques pour caractériser les valeurs *Ok* (via les formes *Ok*), ainsi que ceux pour spécifier le comportement des valeurs *Ok*, nous allons passer au traitement des cas exceptionnels. Commençons par les étiquetages d'exceptions.

4. L'ETIQUETAGE D'EXCEPTIONS

Lorsqu'une opération est appliquée à des valeurs *Ok* mais que le résultat est erroné ou exceptionnel, il faut attacher une étiquette d'exception appropriée au terme résultant. Par exemple, on veut attacher l'étiquette d'exception *NEGATIVE-NUMBER* à un terme tel que *pred(0)*, ou l'étiquette *TOO-LARGE-NUMBER* à *succ(Maxint)*, ou encore l'étiquette *DIV-BY-0-ERROR* à (*n div 0*). Ceci est spécifié grâce aux *axiomes d'étiquetage* :

Définition 4 :
Etant donnée une exception-signature $\mathbf{\Sigma}$-*exc* = <**S**,**Σ**,**L**> , l'ensemble des *axiomes d'étiquetage*, noté **Lbl-Ax** ("Labelling-Axioms"), est un ensemble fini d'axiomes comme suit :

$$[\, t_1 \in l_1 \wedge \cdots \wedge t_n \in l_n \ \wedge\ v_1 = w_1 \wedge \cdots \wedge v_m = w_m \,] \quad \Rightarrow \quad t \in l$$

où t_i, v_j, w_j et t sont des $\mathbf{\Sigma}$-termes avec variables, les l_i et l sont des éléments de $\mathbf{L} \cup \{Ok\}$. (v_j et w_j doivent être de même sorte pour chaque j, les l_i ne sont pas nécessairement distincts, *n* ou *m* peuvent être égaux à 0).
Intuitivement, l'étiquette supplémentaire "*Ok*" dénote évidemment les valeurs *Ok* d'une exception-algèbre. Nous donnerons plus de détails au sujet de l'étiquette *Ok* dans le chapitre suivant (sur la sémantique des exception-algèbres), il suffit de savoir pour l'instant qu'il étiquette toutes les valeurs

Ok d'une exception-algèbre.

En pratique, on va utiliser l'étiquetage d'exceptions pour associer un "diagnostic" à chaque terme issu de l'application d'une opération à des valeurs *Ok*, mais retournant une valeur exceptionnelle. Ceci n'est pas sans rappeler les préconditions d'application introduites par Guttag [Gut 78]. En fait, il ne s'agit pas ici de déclarer des conditions sous lesquelles une opération peut s'appliquer ou non. Il s'agit simplement d'attacher une étiquette d'exception à certaines valeurs. On peut attacher une étiquette d'exception à n'importe quel terme, par exemple *pred(pred(pred(0)))* répondra à l'étiquette *NEGATIVE-NUMBER*, bien que l'opération *pred* de la racine ne soit pas appliquée à une valeur *Ok*.

Exemple 3 :
L'étiquetage d'exceptions associé aux entiers naturels bornés peut être spécifié comme suit :

$$
\begin{array}{rcl}
 & & pred(0) \in NEGATIVE\text{-}NUMBER \\
n \in NEGATIVE\text{-}NUMBER & \Rightarrow & pred(n) \in NEGATIVE\text{-}NUMBER \\
n < m = True & \Rightarrow & (n - m) \in NEGATIVE\text{-}NUMBER \\
 & & succ(Maxint) \in TOO\text{-}LARGE\text{-}NUMBER \\
n \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & succ(n) \in TOO\text{-}LARGE\text{-}NUMBER \\
n \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & (n + 0) \in TOO\text{-}LARGE\text{-}NUMBER \\
succ(n)+m \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n+succ(m) \in TOO\text{-}LARGE\text{-}NUMBER \\
n \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & (n \times 1) \in TOO\text{-}LARGE\text{-}NUMBER \\
(n \times m + n) \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & (n \times succ(m)) \in TOO\text{-}LARGE\text{-}NUMBER \\
 & & (n \; div \; 0) \in DIV\text{-}BY\text{-}0\text{-}ERROR
\end{array}
$$

Ici, deux remarques s'imposent. Elles font malheureusement un peu appel à l'aspect sémantique de notre formalisme, mais nous espérons que les choses sont suffisamment intuitives pour pouvoir faire ces remarques dès maintenant.

Remarques 1 :

- Remarquons que les étiquettes d'exception ne doivent pas se propager implicitement. Par exemple, *pred(0)* est étiqueté par *NEGATIVE-NUMBER*, mais il n'y a clairement aucune raison pour propager implicitement cette étiquette à un terme tel que *succ(pred(0))*. C'est pour cette raison que nous spécifions explicitement des axiomes tels que
  $$n \in NEGATIVE\text{-}NUMBER \Rightarrow pred(n) \in NEGATIVE\text{-}NUMBER$$
- Remarquons aussi que les *Ok*-axiomes ne s'appliquent pas sur toutes les valeurs de l'algèbre (ils ne s'appliquent que sur les valeurs *Ok*). En conséquence, si $n+m > Maxint$, alors les *Ok*-axiomes n'amalgameront pas un terme tel que $succ^n(0) + succ^m(0)$ sur la forme normale $succ^{n+m}(0)$ c'est pour cette raison que nous devons spécifier un axiome d'étiquetage tel que :
  $$succ(n)+m \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow n+succ(m) \in TOO\text{-}LARGE\text{-}NUMBER$$

Munis des étiquetages d'exceptions, nous pouvons maintenant spécifier le traitement proprement dit des termes exceptionnels (amalgame de plusieurs valeurs erronées, ou récupération de certains termes exceptionnels).

5. LES AXIOMES GENERALISES

Les axiomes généralisés sont des axiomes conditionnels qui peuvent contenir des conditions relatives à l'étiquetage d'exception dans leurs prémisses. Cette généralisation nous offre un grande souplesse de spécification du traitement des valeurs exceptionnelles, puisque l'on va pouvoir traiter les exceptions de manière différenciée, en fonction du "diagnostic" qui leur est associé.

Définition 5 :
Etant donnée une exception-signature $\mathbf{\Sigma}$-*exc* = <**S**,**Σ**,**L**> , on note **Gen-Ax** un ensemble fini d'*axiomes généralisés* de la forme suivante :

$$[\, t_1 \in l_1 \wedge \cdots \wedge t_n \in l_n \;\wedge\; v_1 = w_1 \wedge \cdots \wedge v_n = w_n \,] \;\Rightarrow\; v_{n+1} = w_{n+1}$$

où les t_i, v_j et w_j sont des $\mathbf{\Sigma}$-termes avec variables, et les l_i sont des éléments de $\mathbf{L} \cup \{Ok\}$. Naturellement, v_j et w_j doivent être de même sorte pour chaque j, les l_i ne sont pas nécessairement distincts, n ou m peuvent être égaux à 0 ; et l'étiquette *Ok* étiquette les valeurs *Ok* d'une exception-algèbre.

Les axiomes généralisés offrent de nombreuses possibilités de traitement d'exceptions différents dans les entiers naturels bornés. A titre d'exemple, on peut choisir le traitement suivant :

- □ Ne faire aucune récupération des valeurs négatives, et au contraire les amalgamer sur une constante (disons *crash*).
- □ Amalgamer aussi le résultat de toute division par *0* sur *crash*.
- □ Récupérer tous les termes qui rencontrent une valeur trop grande dans leur "histoire" (mais aucune valeur négative), pourvu que leur valeur finale soit comprise entre *0* et *Maxint*.
- □ Associer sa forme normale à tout terme rencontrant une valeur trop grande dans son histoire (mais aucune valeur négative), même si cette forme est plus grande que *Maxint*.

Exemple 4 :
Ceci peut se spécifier, par exemple, de la manière suivante, en supposant avoir ajouté une constante *crash* dans la signature :

$$
\begin{array}{rcrcl}
n \in NEGATIVE\text{-}NUMBER & \Rightarrow & n & = & crash \\
 & & n\ div\ 0 & = & crash \\
 & & succ(crash) & = & crash \\
 & & pred(crash) & = & crash \\
 & & crash + n & = & crash \\
 & & crash - n & = & crash \\
 & & n - crash & = & crash \\
 & & crash \times n & = & crash \\
 & & crash\ div\ n & = & crash \\
 & & n\ div\ crash & = & crash \\
 & & n + m & = & m + n \\
 & & n \times m & = & m \times n \\
n \in TOO\text{-}LARGE\text{-}NUMBER \wedge m \in Ok & \Rightarrow & n < m & = & False \\
n \in Ok \wedge m \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n < m & = & True \\
\left.\begin{array}{r} succ(n) \in TOO\text{-}LARGE\text{-}NUMBER \\ succ(m) \in TOO\text{-}LARGE\text{-}NUMBER \end{array}\right\} & \Rightarrow & succ(n) < succ(m) & = & n < m \\
succ(n) \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & pred(succ(n)) & = & n \\
n \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n + 0 & = & n \\
n + succ(m) \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n + succ(m) & = & succ(n) + m \\
n \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n - 0 & = & n \\
n \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n - succ(m) & = & pred(n) - m \\
n \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n \times 0 & = & 0 \\
n \times succ(m) \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n \times succ(m) & = & (n \times m) + n \\
r < m = True & \Rightarrow & ((n \times m) + r)\ div\ m & = & n
\end{array}
$$

Cet ensemble d'axiomes généralisés peut paraître long ; en fait ceci traduit seulement le traitement cas par cas de chacune des situations exceptionnelles. Si ces axiomes sont nombreux, c'est seulement parce que le traitement d'exceptions que l'on fait est particulier à chaque cas d'exception. Si l'on spécifiait des structures plus simples, les axiomes généralisés seraient eux aussi plus simples. Si la plupart des formalismes de traitement d'exceptions que nous avons cités au chapitre I ne présentent pas d'ensembles d'axiomes si longs dans leurs exemples, c'est parce qu'ils n'offrent pas un traitement d'exceptions aussi différencié que le notre. L'annexe 3 offre des exemples d'exception-spécifications plus simples, des entiers bornés et des entiers non-bornés, où aucune récupération n'est spécifiée. On constate alors que ces spécifications sont très courtes et simples.
Remarquons cependant que certains axiomes font double emploi avec les *Ok*-axiomes. Par exemple, des axiomes tels que :

$$
\begin{array}{rcl}
n \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n + 0 = n \\
n + succ(m) \in TOO\text{-}LARGE\text{-}NUMBER & \Rightarrow & n + succ(m) = succ(n) + m
\end{array}
$$

pourraient être simplement écrits sous la forme suivante dans **Gen-Ax** :

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

dès lors, du fait que l'on a supprimé les prémisses, ces axiomes s'appliquent à toutes les valeurs des exception-algèbres, y compris les valeurs *Ok*, si bien que l'on peut les omettre dans **Ok-Ax**. Nous reverrons ceci de plus près dans la chapitre suivant sur la sémantique associée aux exception-spécifications. On peut en effet difficilement entrer dans de tels détails sans décrire d'abord la sémantique exacte de nos axiomes.

Nous disposons maintenant de tous les éléments pour définir totalement notre syntaxe : les exception-spécifications.

Définition 6 :
Une *exception-spécification* est un n-uplet :

$$\mathbf{SPEC} = < \mathbf{S}, \mathbf{\Sigma}, \mathbf{L}, \mathbf{Ok\text{-}Frm}, \mathbf{Ok\text{-}Ax}, \mathbf{Lbl\text{-}Ax}, \mathbf{Gen\text{-}Ax} >$$

où :

- □ <**S**,**Σ**,**L**> est une exception-signature (définition 1)
- □ **Ok-Frm** est une déclaration de formes *Ok* sur **Σ**-*exc* (définition 2)
- □ **Ok-Ax** est un ensemble fini d'*Ok*-axiomes sur **Σ**-*exc* (définition 3)
- □ **Lbl-Ax** est un ensemble fini d'axiomes d'étiquetage sur **Σ**-*exc* (définition 4)
- □ **Gen-Ax** est un ensemble fini d'axiomes généralisés sur **Σ**-*exc* (définition 5).

Le chapitre suivant décrit la sémantique associée aux exception-spécifications : les exception-algèbres et les notions de validation associées à une exception-spécification.

Chapitre IV :

Exception-Algèbres

Ce chapitre décrit la sémantique associée aux exception-spécifications. Les modèles répondant à une exception-spécification s'appellent les *exception-algèbres*. Pour les décrire, nous suivons la démarche habituelle :

- □ Nous commençons par définir les exception-algèbres sur une exception-signature **Σ**-*exc* .
- □ Parmi les **Σ**-*exc*-algèbres, nous définissons celles qui *valident* les déclarations de formes *Ok* (**Ok-Frm**).
- □ De même, nous définissons la validation des *Ok*-axiomes (**Ok-Ax**)
- □ Puis nous définissons la validation des axiomes d'étiquetage (**Lbl-Ax**)
- □ Enfin nous définissons la validation des axiomes généralisés (**Gen-Ax**).

Une **Σ**-*exc*-algèbre validera la spécification **SPEC** si et seulement si elle valide **Ok-Frm**, **Ok-Ax**, **Lbl-Ax** et **Gen-Ax**.

1. LES Σ-*exc*-ALGEBRES

Les exception-algèbres doivent contenir un certain nombre d'informations "en plus" de celles qui sont classiquement incluses dans les algèbres :

- □ Parmi toutes les valeurs de l'algèbre, il faut connaître celles qui sont *Ok*.
- □ Pour chaque étiquette d'exception, il faut connaître quelles sont les valeurs qui réponde à cette étiquette.

Il en résulte qu'une exception-algèbre doit être munie :

- □ d'un sous-ensemble contenant les valeurs *Ok* de cette algèbre,
- □ et pour chaque étiquette d'exception, d'un sous-ensemble contenant toutes les valeurs étiquetées par elle.

C'est exactement ce que traduit la définition suivante.

Définition 7 :
Soit $\mathbf{\Sigma}$-*exc* = <**S**,**Σ**,**L**> une exception-signature. Une *exception-algèbre* sur $\mathbf{\Sigma}$-*exc*, aussi appelée une $\mathbf{\Sigma}$-*exc-algèbre*, est une $\mathbf{\Sigma}$-algèbre classique munie d'une famille de sous-ensembles indéxée par $\mathbf{L} \cup \{Ok\}$.
Ce qui signifie précisément qu'une $\mathbf{\Sigma}$-*exc*-algèbre A est définie par :

- Le *support* de A est un ensemble (hétérogène), noté *A*, partitionné en une famille de sous-ensembles indéxée par **S** : $\{ A_s \,/\, s \in \mathbf{S} \}$. L'ensemble *A* est la réunion disjointe des A_s, où *s* parcours **S**.

- Pour chaque opération élément de $\mathbf{\Sigma}$:
  $$op : \; s_1 \cdots s_n \;\rightarrow\; s$$
  l'algèbre A est munie d'une fonction :
  $$op_A : \; A_{s_1} \times \cdots \times A_{s_n} \;\rightarrow\; A_s \; .$$
  Le plus souvent, par abus de langage, on note encore *op* la fonction op_A associée à l'opération *op* dans *A*.

- Enfin, pour toute étiquette $l \in \mathbf{L} \cup \{Ok\}$, l'algèbre A est munie d'un sous-ensemble quelconque A_l de A . Ces sous-ensembles A_l sont hétérogènes : il peuvent intersecter plusieurs sortes ; de plus ils ne sont pas nécessairement disjoints deux à deux, et leur réunion n'est pas nécessairement égale à A.
  Les éléments de A_{Ok} sont appelés *les valeurs Ok* de *A*.
  Pour toute étiquette d'exception l de **L**, on dit que les éléments de A_l sont *étiquetés par l*.

On notera A = $(A,\{A_{l_i}\})$ une $\mathbf{\Sigma}$-*exc*-algèbre ; notation qui sous-entend qu'une $\mathbf{\Sigma}$-*exc*-algèbre A est une $\mathbf{\Sigma}$-algèbre classique *A*, munie d'une famille de sous-ensembles $\{A_{l_i}\}$ indéxée par $\mathbf{L} \cup \{Ok\}$.

Remarquons qu'à ce niveau de la spécification, rien n'est imposé sur les ensembles A_{l_i} , ils peuvent être absolument quelconques.

Exemples 5 :
Voici quelques exemples de $\mathbf{\Sigma}$-*exc*-algèbres sur l'exception-signature de *NAT* décrite en exemple 1 (chapitre précédent, section 1) :

- L'exception-algèbre des termes fermés, $T_{\mathbf{\Sigma}}$, munie de :
  $$T_{\mathbf{\Sigma},Ok} = T_{\mathbf{\Sigma},NEGATIVE\text{-}NUMBER} = T_{\mathbf{\Sigma},TOO\text{-}LARGE\text{-}VALUE} = T_{\mathbf{\Sigma},DIV\text{-}BY\text{-}0\text{-}ERROR} = \varnothing$$

- L'exception-algèbre des entiers relatifs **Z**, avec $z \; div \; 0 = Maxint$ pour tout *z*, et munie de :
  $$\mathbf{Z}_{Ok} \;=\; [0, Maxint]$$
  $$\mathbf{Z}_{NEGATIVE\text{-}NUMBER} \;=\;]-\infty, 0[$$
  $$\mathbf{Z}_{TOO\text{-}LARGE\text{-}NUMBER} \;=\;]Maxint, +\infty[$$
  $$\mathbf{Z}_{DIV\text{-}BY\text{-}0\text{-}ERROR} \;=\; \{Maxint\}$$
  Intuitivement, dans cette algèbre, les opérations travaillent de la manière habituelle, mais les résultats sont erronés s'ils sont hors de *[0,Maxint]*. De plus, toute division par *0* est récupérée sur *Maxint*.

- L'exception-algèbre $\mathbf{N} \cup \{crash\}$ avec : $pred(0) = crash = n \; div \; 0$, toute opération appliquée à *crash* retourne *crash*, et munie de :
  $$\mathbf{N}_{Ok} \;=\; [0, Maxint]$$
  $$\mathbf{N}_{NEGATIVE\text{-}NUMBER} \;=\; \{crash\}$$
  $$\mathbf{N}_{TOO\text{-}LARGE\text{-}NUMBER} \;=\;]Maxint, +\infty[$$

$$\mathbf{N}_{DIV\text{-}BY\text{-}0\text{-}ERROR} = \{crash\}$$

Nous verrons que cette exception-algèbre est en fait l'algèbre initiale associée à la spécification développée dans les exemples du chapitre précédent.

Nous avons précisé que les ensembles A_{l_i} sont des sous-ensembles quelconques ; en particulier, leurs intersections ne sont pas nécessairement vides. C'est le cas du dernier exemple que nous avons donné : la valeur *crash* est à la fois étiquetée par *NEGATIVE-NUMBER* et par *DIV-BY-0-ERROR*. C'est aussi le cas de l'avant dernier exemple, *Maxint* est à la fois étiqueté par *Ok* et par *DIV-BY-0-ERROR*.
D'autre part, les ensembles A_{l_i} peuvent intersecter plusieurs sortes distinctes. C'est le cas des tableaux, lorsque l'on attache la même étiquette d'exception à l'accès hors des bornes ou à l'affectation hors des bornes : des termes tels que *t[i]* ou *t[i]:=x* (avec *i* hors des bornes) recevront la même étiquette *OUT-OF-RANGE*, bien que l'un soit de sorte prédéfinie et l'autre de sorte tableau.

2. LES VALEURS ERRONEES

La définition d'une exception-algèbre fournit l'ensemble des valeurs *Ok* de l'algèbre ainsi que les ensembles étiquetés par des étiquettes d'exception, mais elle ne décrit pas l'ensemble des valeurs erronées. Comme nous l'avons déjà remarqué (chapitre II, section 1.2), l'ensemble des valeurs erronées d'une exception-algèbre n'est pas nécessairement le complémentaire de l'ensemble des valeurs *Ok*.
De la même façon que la théorie classique des types abstraits algébriques n'impose pas a priori que les spécifications soient complètement spécifiées, nous ne voulons pas imposer d'office que la répartition des valeurs *Ok* ou erronées soit toujours complète (i.e. il peut y avoir des valeurs qui ne soient ni *Ok* ni erronées dans une exception-algèbre).

Pour définir les valeurs erronées d'une exception-algèbre, il semble raisonnable de suivre l'idée suivante :

- Une valeur étiquetée par une étiquette d'exception est erronée, sauf si cette valeur est *Ok*, auquel cas on dira qu'elle est récupérée.
- Les valeurs erronées se propagent implicitement, sauf si elle font l'objet d'une récupération, c'est à dire si l'on rencontre une valeur *Ok* lors de la propagation.
- Une valeur qui ne résulte pas d'une valeur étiquetée par une étiquette d'exception (y compris par propagation implicite d'erreurs), et qui n'est pas *Ok*, n'est ni *Ok* ni erronée (elle est en somme incomplètement spécifiée).

La définition suivante traduit exactement cette idée :

Définition 8 :
Soit $A = (A,\{A_{l_i}\})$ une exception-algèbre. L'ensemble des *valeurs erronées* de A, noté A_{err}, est par définition le plus petit sous-ensemble (hétérogène) de *A* vérifiant les deux propriétés suivantes :

- Pour toute étiquette d'exception l élément de **L**, A_{err} contient $(A_l - A_{Ok})$.
- Pour toute opération $op(\_,\ldots,\_)$ élément de **Σ**, et pour tous éléments $u_1 \cdots u_n$ de *A* (de sortes compatibles avec l'arité de *op*) :
  si l'un des u_i est élément de A_{err} et si $op(u_1,\ldots,u_n)$ n'est pas élément de A_{Ok}, alors

$op(u_1, \ldots, u_n)$ est encore élément de A_{err}.

La première propriété signifie qu'une valeur étiquetée par une étiquette d'exception est erronée, sauf si elle a été récupérée. La seconde signifie que les erreurs se propagent implicitement, sauf en cas de récupération.

Pour que cette définition soit valide, il faut bien sûr prouver que ce plus petit ensemble A_{err} existe. C'est particulièrement simple :

- La famille de tous les sous-ensembles de A vérifiant ces deux propriétés n'est pas vide : elle contient au moins A lui même.
- L'intersection de tous les sous-ensembles de A vérifiant ces deux propriétés vérifie encore ces deux propriétés :
  - Puisque chacun des ensembles de la famille contient les $(A_{l_i} - A_{Ok})$, leur intersection aussi.
  - si l'un des u_i est élément de tous les ensembles de la famille (i.e. est élément de leur intersection), et si $op(u_1, \ldots, u_n)$ n'est pas élément de A_{Ok}, alors $op(u_1, \ldots, u_n)$ appartient à tous les ensembles de la famille, donc à leur intersection.

Exemple 6 :
Dans le cas de l'exception-algèbre $\mathbf{N} \cup \{crash\}$ décrite en exemple 5, $\mathbf{N}_{err}$ est égal à $\{crash\} \cup]Maxint, +\infty[$. Si l'on considère par exemple la valeur *succ(Maxint)*, elle est étiquetée par *TOO-LARGE-NUMBER*, et du fait qu'elle n'est pas récupérée (elle n'est pas élément de $\mathbf{N}_{Ok}$), elle est erronée. Par contre, si l'on considère la valeur associée au terme *pred(succ(Maxint))*, elle serait susceptible d'être erronée (par propagation implicite d'erreurs), mais du fait que sa valeur est *Ok* (récupérée sur *Maxint*), elle n'est pas élément de $\mathbf{N}_{err}$.
L'exemple 5 fournit aussi un cas d'exception-algèbre où les valeurs sont incomplètement spécifiées : dans l'algèbre des termes fermés T_{Σ}, avec tous les T_{Σ, l_i} vides, aucun terme n'est complètement spécifié, puisqu'aucun n'est *Ok* ou étiqueté par une étiquette d'exception.

Cette construction des valeurs erronées d'une exception-algèbre nous sera utile dans le chapitre VII, pour définir la suffisante complétude en tenant compte du traitement d'exceptions.

Nota :
Rappelons que dans la définition 1 du chapitre précédent, on impose qu'aucune étiquette de **L** ne soit notée "*Ok*" ou "*err*". On comprend maintenant pourquoi : ces deux notations sont utilisées pour A_{Ok} et A_{err} ; d'où un risque de surcharge sur ces étiquettes.

3. LES FORMES *Ok*

Le but de cette section est de définir la validation de **Ok-Frm**. Pour ce faire, nous avons remarqué qu'il fallait travailler sur des *termes*, et non directement sur les valeurs d'une exception-algèbre (car il faut distinguer les *termes exceptionnels* des *valeurs erronées*, pour éviter des inconsistances induites par la récupération, cf. chapitre II, section 2.1.).
Nous ne pouvons donc pas travailler directement sur les exception-algèbres car elles ne modélisent pas des termes, mais des valeurs (des classes d'équivalence). Une solution simple consisterait à toujours travailler sur les termes fermés, mais ceci n'est pas satisfaisant car nous voulons pouvoir aussi

traiter les notions de validations dans les exception-algèbres non finiment générées. Nous proposons une solution qui permet de prendre en compte toutes les exception-algèbres, même non finiment générées : pour toute exception-algèbre A, il suffit de travailler dans l'algèbre $T_{\Sigma(A)}$. Commençons par rappeler la définition de $T_{\Sigma(A)}$; il est fondamental de bien la comprendre pour la suite.

Définition 9 :
Soit (**S**,**Σ**) une signature classique. Soit $V = \{V_s\}_{s \in \mathbf{S}}$ un ensemble hétérogène indexé par **S**. L'*algèbre libre des* **Σ***-termes avec variables dans V*, notée $T_{\Sigma(V)}$, est récursivement définie par :

- □ Pour toute sorte $s \in \mathbf{S}$, le support de $T_{\Sigma(V),s}$ contient V_s, et contient toutes les constantes de **Σ** de sorte s.
- □ Pour toute opération non constante de **Σ**, $op(\_,...,\_) \in \mathbf{\Sigma}$, et pour tous termes $t_1 \cdots t_n$ éléments de $T_{\Sigma(V)}$ (de sortes compatibles avec l'arité de *op*), le terme $op(t_1, \ldots, t_n)$ est encore un élément de $T_{\Sigma(V)}$.

L'exemple le plus courant est le cas où *V* est un ensemble de variables. Mais remarquons que cette définition ne suppose pas que *V* soit fini ; on peut donc en particulier l'appliquer au cas où *V* est en fait une algèbre (classique) *A*. Dans ce cas, *A* et $T_{\Sigma(A)}$ sont toutes deux des **Σ**-algèbres, et il existe un morphisme canonique

$$\mathit{éval} : T_{\Sigma(A)} \rightarrow A$$

qui fait correspondre à chaque terme de $T_{\Sigma(A)}$ sa valeur calculée dans *A*. Le morphisme *éval* est donc défini comme suit :

- □ Remarquons tout d'abord que les éléments de *A* sont des *constantes* de $T_{\Sigma(A)}$, par conséquent, pour tout élément *a* de *A* (considéré comme une constante de $T_{\Sigma(A)}$), *éval(a)* est canoniquement égal à *a* (considéré comme une valeur de *A*).
- □ Puis, pour toute opération *op* de **Σ**, on a :
$$\mathit{éval}(op(t_1, \ldots, t_n)) = op_A(\mathit{éval}(t_1), \ldots, \mathit{éval}(t_n)) .$$

Exemple 7 :
Soit **Σ** = { *0* , *succ_* , *pred_* } l'ensemble des opérations des entiers relatifs (*INT*). Considérons l'algèbre **Z**=]...,-2,-1,0,1,2,...[, l'algèbre $T_{\Sigma(\mathbf{Z})}$ est alors égale à l'ensemble des termes avec "variables" dans]...,-2,-1,0,1,2,...[; et le morphisme *éval* est l'application de $T_{\Sigma(\mathbf{Z})}$ dans **Z** qui fait correspondre la *valeur* –4 à des *termes* tels que *pred(–3), pred(pred(–2)), succ(pred(–4))*, ou encore la valeur 1 au terme *pred(2)* ...etc.

Remarquons que le morphisme *éval* est intrinsèquement défini par la structure de **Σ**-algèbre de *A*. aucune équation ou information supplémentaire n'est requise pour définir *éval*.

Nous pouvons maintenant définir les formes *Ok* comme des *termes* de $T_{\Sigma(A)}$.

Définition 10 :
Soit **Ok-Frm** une déclaration de formes *Ok* sur l'exception-signature **Σ***-exc*. Soit $A = (A,\{A_{l_i}\})$ une exception-algèbre. L'ensemble des *formes Ok* de $T_{\Sigma(A)}$, noté $Ok\text{-}Frm_A$, est le plus petit sous-ensemble (hétérogène) de $T_{\Sigma(A)}$ vérifiant les deux propriétés suivantes :

- □ $Ok\text{-}Frm_A$ contient A_{Ok} (rappelons que les éléments de *A* sont des *constantes* de $T_{\Sigma(A)}$, en particulier les éléments de A_{Ok} sont des constantes de $T_{\Sigma(A)}$).

□ Pour toute déclaration élémentaire de **Ok-Frm** :

$$\left.\begin{array}{l} t_1 \in \text{Ok-Frm} \wedge \cdots \wedge t_m \in \text{Ok-Frm} \\ \wedge\ x_1 \in l_1 \wedge \cdots \wedge x_n \in l_n \\ \wedge\ v_1 = w_1 \wedge \cdots \wedge v_p = w_p \end{array}\right\} \Rightarrow \quad t \in \text{Ok-Frm}$$

et pour toute substitution σ à valeur dans $T_{\Sigma(A)}$, on a :
si $\sigma(t_i) \in \textit{Ok-Frm}_A$ pour tout $i=1..m$, et $\textit{éval}[\sigma(x_j)] \in A_{l_j}$ pour tout $j=1..n$, et $\textit{éval}[\sigma(v_k)] = \textit{éval}[\sigma(w_k)]$ pour tout $k=1..p$, alors $\sigma(t)$ est élément de $\textit{Ok-Frm}_A$.

La seconde propriété traduit simplement le fait que l'ensemble des formes *Ok* de $T_{\Sigma(A)}$ doit valider les déclarations élémentaires de **Ok-Frm**.
La première, quant-à elle, traduit le fait que tout élément *Ok* de A doit être une forme *Ok* lorsqu'il est considéré comme une constante de $T_{\Sigma(A)}$. Il est en effet bien naturel que l'on veuille que les valeurs *Ok* de A deviennent des formes *Ok* de $T_{\Sigma(A)}$ puisque ce ne sont que des constantes de $T_{\Sigma(A)}$.
Par exemple, si A=**N** est l'algèbre des entiers naturels, on veut bien sûr que le terme *succ(succ(succ(0)))* soit une forme *Ok*, mais on veut aussi que la constante 3 en soit une. Ceci est justement obtenu par la première condition.

Remarquons que l'existence de $\textit{Ok-Frm}_A$ est claire car, d'une part, la famille de tous les sous-ensembles de $T_{\Sigma(A)}$ vérifiant ces deux propriétés n'est pas vide (elle contient $T_{\Sigma(A)}$), et d'autre part, l'intersection de tous les sous-ensembles de cette famille vérifie encore les deux propriétés (immédiat).

Nous sommes maintenant à même de définir la *validation* de **Ok-Frm** :

Définition 11 :
Soit **Ok-Frm** une déclaration de formes *Ok* sur l'exception-signature **Σ**-*exc*. Une exception-algèbre $A = (A,\{A_{l_i}\})$ *valide* **Ok-Frm** si et seulement si :

$$\textit{éval}(\textit{Ok-Frm}_A) \subset A_{Ok}.$$

Ceci signifie que les formes *Ok* de $T_{\Sigma(A)}$ doivent avoir une valeur *Ok* dans A, après évaluation.
Remarquons que l'inclusion inverse est toujours satisfaite car $\textit{Ok-Frm}_A$ contient toujours A_{Ok} (par définition).

Exemple 8 :
Les formes *Ok* de $T_{\Sigma(\mathbf{N})}$, pour l'exception-algèbre **N**∪*{crash}* décrite en exemple 5, sont les termes de la forme $succ^i(n)$ tels que $i+n \leq Maxint$. Leur évaluation donne la valeur $i+n$ dans **N**, qui est comprise entre *0* et *Maxint*, si bien que **N**∪*{crash}* valide **Ok-Frm**.
Il en est de même pour l'exception-algèbre **Z** décrite dans ce même exemple 5.
Par contre, l'exception-algèbre des termes fermés (toujours décrite en exemple 5), avec $T_{\Sigma,Ok}$ vide, ne valide pas **Ok-Frm**. En effet, l'évaluation des formes *Ok* de $T_{\Sigma(T_\Sigma)}$ correspond aux termes de la forme $succ^i(0)$, avec $0 \leq i \leq Maxint$, or $T_{\Sigma,Ok}$ est vide.

Nous allons maintenant définir la validation des *Ok*-axiomes en utilisant le sous-ensemble $\textit{Ok-Frm}_A$ de $T_{\Sigma(A)}$.

4. VALIDATION DE Ok-Ax

Les *Ok*-axiomes permettent de spécifier le comportement des cas non-exceptionnels. Remarquons que nous n'avons pas encore caractérisé tous les termes *Ok* ou exceptionnels de $T_{\Sigma(A)}$ en ce point de

la sémantique. En effet, pour l'exemple des entiers naturels bornés, les formes *Ok* sont les termes de la forme $succ^i(n)$ tels que $0 \leq i+n \leq Maxint$. Rien ne permet d'affirmer en ce point que des termes tels que *0+0* ou *pred(Maxint)* sont des termes *Ok*.

Le rôle sémantique des *Ok*-axiomes est donc double : d'une part ils assurent les amalgames "*Ok*" de la structure spécifiée, d'autre part ils permettent de caractériser les termes *Ok* de $T_{\Sigma(A)}$. Ces deux aspects de la sémantique des *Ok*-axiomes peuvent sembler paradoxaux :

- □ D'une part nous voulons que les *Ok*-axiomes caractérisent les termes *Ok* de $T_{\Sigma(A)}$: se sont ceux qui obtiennent une valeur *Ok* via **Ok-Ax**.
- □ D'autre part nous voulons que les *Ok*-axiomes ne s'appliquent qu'aux termes *Ok*. Une idée simple serait qu'une occurrence d'*Ok*-axiome ne s'applique que si les deux membres de l'égalité sont des termes *Ok* ; mais ceci supposerait d'avoir caractérisé *au préalable* tous les termes *Ok*.

En fait, la méthode que nous allons suivre évite cet écueil. C'est la suivante :

- □ Nous connaissons déjà un ensemble de termes qui sont assurément *Ok* dans $T_{\Sigma(A)}$: ce sont les formes *Ok* (*Ok-*Frm_A). Cet ensemble nous munit donc d'un "noyau" de termes *Ok*.
- □ Si l'on considère maintenant une occurrence d'*Ok*-axiome de la forme $\alpha = t$, où α est une forme *Ok* ($\in$*Ok-*Frm_A) ; si *t* n'est pas exceptionnel, il semble raisonnable de dire que *t* est un terme *Ok*, et que son *Ok*-évaluation est la forme *Ok* α .
- □ Nous venons ainsi de "*propager les termes Ok*" d'un cran ; nous pouvons continuer inductivement. Si l'on considère une nouvelle occurrence *d'Ok*-axiome de la forme $t = t'$, où *t* est le terme *Ok* construit précédemment, et si *t'* n'est pas exceptionnel, on peut de la même façon propager la qualité "*Ok*" au terme *t'* ; et son *Ok*-évaluation sera aussi la forme *Ok* α , via *t*. Et ainsi de suite ...

La seule imprécision dans la méthode que nous venons de décrire est : comment caractériser si un terme *t* de $T_{\Sigma(A)}$ est *exceptionnel* ou non ?
Par exemple, considérons l'*Ok*-axiome

$$pred(succ(n)) = n$$

et supposons que l'algèbre A vérifie : *succ(Maxint)=Maxint* (par récupération) ; notre propos est de traduire le fait que le terme *succ(Maxint)*, et donc sa propagation *pred(succ(Maxint))*, est exceptionnel (pour que l'occurrence *pred(succ(Maxint))=Maxint* ne s'applique pas). Or, comme on l'avait déjà remarqué, aussi bien *succ(Maxint)* que *pred(succ(Maxint))* ont des valeurs *Ok*. Il nous faut en fait traduire l'idée suivante : puisque *succ(Maxint)* ne possède pas de forme *Ok* associée *grâce aux Ok-axiomes*, il est exceptionnel ; et en particulier sa "propagation" *pred(succ(Maxint))* est aussi exceptionnelle.
La méthode pour traduire ceci est simple : il suffit d'imposer une *Ok*-évaluation par "les sous-termes les plus profonds d'abord".

Cette contrainte sur la sémantique des *Ok*-axiomes peut s'exprimer au moyen d'une congruence (au sens classique). Puisqu'il est nécessaire que cette congruence s'applique sur des *termes* et non sur les *valeurs* de l'algèbre, une "*Ok*-congruence" se construit dans $T_{\Sigma(A)}$ et non dans *A*. Le morphisme *éval* nous permettra de porter vers A la notion de *validation* des *Ok*-axiomes, à partir de cette *Ok*-congruence sur $T_{\Sigma(A)}$.

Proposition 1 :
Soit **Ok-Frm** une déclaration de formes *Ok* sur une exception-signature **Σ**-*exc*. Soit **Ok-Ax** un ensemble d'*Ok*-axiomes sur **Σ**-*exc*, et soit A une **Σ**-*exc*-algèbre.
Il existe une plus petite congruence sur $T_{\Sigma(A)}$, notée $\equiv_{Ok}$, vérifiant la condition "SI..ALORS.." suivante :

Pour toute substitution σ à valeur dans $T_{\Sigma(A)}$, et pour tout *Ok*-axiome de **Ok-Ax**

$$[\, v_1 = w_1 \ \wedge \cdots \wedge \ v_n = w_n \,] \quad \Rightarrow \quad v = w$$

(ou $w = v$ car nos *Ok*-axiomes ne sont pas orientés), en posant

$$\sigma(v) = op(t_1, \ldots, t_m) \ (^2)$$

SI les trois conditions suivantes sont vérifiées :

- □ $éval[\sigma(v_i)] = éval[\sigma(w_i)]$ pour tout $i=1..n$
- □ il existe des formes *Ok* $\alpha_1 \cdots \alpha_m$ ($\in Ok\text{-}Frm_A$) telles que $t_j \equiv_{Ok} \alpha_j$ pour tout $j=1..m$
- □ il existe une forme *Ok* α ($\in Ok\text{-}Frm_A$) telle que $\sigma(w) \equiv_{Ok} \alpha$

ALORS $\sigma(v) = \sigma(w)$.

Avant de donner la preuve de cette proposition, commençons par en donner l'explication intuitive :

Remarque 2 :
Les trois conditions incluses dans le "**SI**" traduisent les faits suivants :

- □ La première condition est simplement la validation des prémisses de l'*Ok*-axiome (il faut que la *valeur* de v_i soit égale à la *valeur* de w_i dans A).
- □ La seconde reflète une évaluation par "les termes les plus profonds d'abord". En effet, "$t_j \equiv_{Ok} \alpha_j$" signifie exactement que l'*Ok*-évaluation de tous les sous-termes stricts de $\sigma(v)$ aboutit (c'est à dire qu'elle retourne une forme *Ok*). C'est cette condition qui permet de limiter l'action des *Ok*-axiomes aux termes non-exceptionnels.
- □ La dernière condition enfin traduit le fait que l'évaluation d'un des deux membres de l'égalité a déjà abouti (ici $\sigma(w)$, mais on peut inverser les rôles de *v* et *w* car nos *Ok*-axiomes ne sont pas orientés). C'est cette dernière condition qui exprime la "*propagation des termes Ok*" par les *Ok*-axiomes, à partir des formes *Ok*.

Preuve de la proposition 1 :
Pour montrer qu'il existe une plus petite congruence sur $T_{\Sigma(A)}$ satisfaisant cette propriété "SI..ALORS", nous allons considérer la famille **C** de toutes les congruences sur $T_{\Sigma(A)}$ satisfaisant "SI..ALORS", montrer qu'elle n'est pas vide, puis montrer que l'intersection (i.e. la conjonction) de toutes les congruences de **C** est encore un élément de **C** (i.e. satisfait encore SI..ALORS).

Pour montrer que **C** n'est pas vide, il suffit d'exhiber une congruence sur $T_{\Sigma(A)}$ satisfaisant SI..ALORS. Il est immédiat que la congruence triviale sur $T_{\Sigma(A)}$ est élément de **C** (la congruence triviale étant celle qui amalgame entre eux tous les termes de même sorte).

(2) *m* peut être égal à *0*.

Considérons maintenant la congruence $\equiv_{Ok}$ définie par

$$t \equiv_{Ok} t' \iff \forall \equiv \in \mathbf{C},\ t \equiv t'.$$

Supposons que $\equiv_{Ok}$, σ et l'*Ok*-axiome $([v_1 = w_1 \wedge \cdots \wedge v_n = w_n] \Rightarrow v = w)$ satisfassent les trois conditions incluses dans le "SI". Nous voulons prouver la conclusion de "ALORS", c'est à dire que $\sigma(v) \equiv_{Ok} \sigma(w)$.
Sachant que $\equiv_{Ok}$ est par définition la conjonction de toutes les congruences $\equiv$ de **C**, il suffit de prouver que $\sigma(v) \equiv \sigma(w)$ pour toute congruence $\equiv$ de **C**.
Sachant de plus que, par définition de **C**, toutes les congruences de **C** vérifient la propriété SI..ALORS, il suffit de prouver que toutes les congruences $\equiv$ de **C** satisfont les trois conditions incluses dans "SI". Rappelons que notre hypothèse est que $\equiv_{Ok}$ satisfait ces trois conditions.

- □ La première condition est indépendante du choix de la congruence sur $T_{\mathbf{\Sigma}(A)}$ car le morphisme *éval* est intrinsèque à l'algèbre *A*. Par conséquent, si cette condition est satisfaite une fois (ici pour $\equiv_{Ok}$), alors elle l'est indépendament de toutes congruences, en particulier pour celles de **C**.

- □ En ce qui concerne la seconde condition, le fait que $\alpha_1 \cdots \alpha_m$ soient congrus (respectivement) à $t_1 \cdots t_m$ modulo $\equiv_{Ok}$ implique qu'il en est de même modulo n'importe quelle congruence $\equiv$ de **C**. Ceci resulte simplement du fait que $\equiv_{Ok}$ est la conjonction de toutes les congruences de **C**.

- □ Enfin la dernière condition tombe sous le même argument : si $\sigma(w) \equiv_{Ok} \alpha$ alors $\sigma(w) \equiv \alpha$ pour toute congruence $\equiv$ de **C**, car $\equiv_{Ok}$ est la conjonction de toutes les congruences éléments de **C**.

Ceci clôt la preuve de notre proposition. □

Munis de cette *Ok*-congruence sur $T_{\mathbf{\Sigma}(A)}$, nous pouvons maintenant définir la *validation des Ok-axiomes*. Pour qu'une exception-algèbre valide les *Ok*-axiomes, il faut clairement que deux termes de $T_{\mathbf{\Sigma}(A)}$ qui sont amalgamés via $\equiv_{Ok}$ possèdent des valeurs égales après évaluation sur les valeurs de A. C'est exactement ce que traduit la définition suivante :

Définition 12 :
Soit **Ok-Frm** une déclaration de formes *Ok* sur l'exception-signature **Σ**-*exc*. Soit **Ok-Ax** un ensemble d'*Ok*-axiomes sur **Σ**-*exc*. Une **Σ**-*exc*-algèbre A = $(A,\{A_{l_i}\})$ *valide* **Ok-Ax** si et seulement si :

$$\forall\, t \in T_{\mathbf{\Sigma}(A)},\ \forall\, t' \in T_{\mathbf{\Sigma}(A)},\quad t \equiv_{Ok} t' \Rightarrow \mathit{éval}(t) = \mathit{éval}(t')$$

Nota : (important)
La congruence $\equiv_{\mathit{éval}}$ sur $T_{\mathbf{\Sigma}(A)}$, associée au morphisme *éval, ne vérifie pas nécessairement la propriété* SI..ALORS. La seule chose qu'impose la validation des *Ok*-axiomes est : $\equiv_{Ok} \subset \equiv_{\mathit{éval}}$.

Voici un exemple d'*Ok*-évaluations suivant la congruence $\equiv_{Ok}$:

Exemple 9 :
Rappelons l'*Ok*-axiome concernant les opérations *0*, *succ* et *pred*, pour les entiers naturels bornés :

$$\mathit{pred(succ(n))} = n$$

Supposons que l'on veuille évaluer le terme $\mathit{succ}(\mathit{pred}(\mathit{succ}^{\mathit{Maxint}}(0)))$. Nous devons d'abord tester si tous ses sous-termes stricts possèdent une valeur *Ok*, c'est à dire possèdent une forme *Ok* dans leur classe modulo $\equiv_{Ok}$; c'est la seconde condition de notre proposition (la première est immédiate car l'axiome que l'on veut appliquer a une prémisse vide).
Le terme $\mathit{succ}^{\mathit{Maxint}}(0)$ est déjà une forme *Ok*, il n'y a donc aucune évaluation nécessaire pour lui.
Le terme $\mathit{pred}(\mathit{succ}^{\mathit{Maxint}}(0))$ se réécrit en $\mathit{succ}^{\mathit{Maxint}-1}(0)$ via notre *Ok*-axiome car $\mathit{succ}^{\mathit{Maxint}-1}(0)$ est

une forme *Ok* (troisième condition de la proposition), et $succ^{Maxint}(0)$ est un terme *Ok* (seconde condition). $pred(succ^{Maxint}(0))$ est donc un terme *Ok* (congru à $succ^{Maxint-1}(0)$ modulo $\equiv_{Ok}$).
Il ne reste plus qu'à évaluer le terme $succ(succ^{Maxint-1}(0))$, qui ne nécessite en fait aucune évaluation car c'est une forme *Ok* ($succ^{Maxint}(0)$). Par conséquent, on obtient le résultat suivant :

$$succ(pred(succ^{Maxint}(0))) \equiv_{Ok} succ^{Maxint}(0) .$$

Par contre, si l'on cherche à évaluer le terme $pred(succ(succ^{Maxint}(0)))$, on doit d'abord évaluer le sous-terme $succ(succ^{Maxint}(0))$. Or ce terme ne possède aucune forme *Ok* (il est plus grand que *Maxint*). Il en résulte que l'on ne peut pas appliquer notre *Ok*-axiome, et par conséquent : *le terme* $pred(succ(succ^{Maxint}(0)))$ *n'est pas congru à* $succ^{Maxint}(0)$ *modulo* $\equiv_{Ok}$.
En fait, les axiomes de récupération que nous avons spécifiés au chapitre précédent (exemple 4) vont *récupérer* le terme $pred(succ(succ^{Maxint}(0)))$ sur sa valeur *Ok* $succ^{Maxint}(0)$. Mais il n'en reste pas moins que le terme $pred(succ(succ^{Maxint}(0)))$ est *exceptionnel*, et par conséquent ce n'est en aucun cas aux *Ok*-axiomes d'évaluer sa valeur finale. La classe de ce terme modulo $\equiv_{Ok}$ est réduite au singleton $\{pred(succ(succ^{Maxint}(0)))\}$.

Ici, trois remarques sont importantes.

Remarques 3 :

- □ On constate que la "qualité" d'un terme (i.e. s'il est *Ok* ou exceptionnel) est déterminée uniquement par son résultat, et celui de tous ses sous-termes. Un terme est *Ok* si et seulement si le résultat de ce terme, et celui de chacun de ses sous-termes (évalué au moyen des *Ok*-axiomes *seulement*) appartient aux formes *Ok* déclarées dans **Ok-Frm**. Autrement dit, le fait qu'une opération *op*, appliquée à des termes "*Ok*", retourne une valeur exceptionnelle ou non-exceptionnelle est déterminé par son *codomaine*.
  On peut cependant limiter explicitement les domaines des opérations lorsque cela est utile. Il suffit pour cela d'utiliser les prémisses des *Ok*-axiomes. Par exemple, si l'on veut spécifier une structure d'entiers naturels bornés telle que *Maxint>10*, mais où l'addition ne peut retourner une valeur *Ok* que si le résultat est plus petit que 10, il suffit de spécifier :

$$n<11 = True \implies \begin{array}{rcl} n + 0 & = & n \\ n + succ(m) & = & succ(n) + m \end{array}$$

  On interdit ainsi l'évaluation de tout terme de la forme $t_1 + t_2$ dont le résultat est plus grand que 10 en "bloquant" le cas de base.

- □ L'évaluation "par les termes les plus profonds d'abord" décrite par la proposition 1 correspond exactement à une *propagation implicite* des termes exceptionnels. En effet, si un terme *t* est exceptionnel (i.e. n'est congru à aucune forme *Ok* modulo $\equiv_{Ok}$), alors tout terme contenant *t* parmi ses sous-termes est aussi exceptionnel (car il faudrait évaluer *t* avant).
  Ainsi, on peut définir les termes *Ok* (i.e. non-exceptionnels) comme étant les termes *t* de $T_{\Sigma(A)}$ tels qu'il existe une forme *Ok* $\alpha \in Ok\text{-}Frm_A$ vérifiant $t \equiv_{Ok} \alpha$. Et un terme *t* de $T_{\Sigma(A)}$ sera *exceptionnel* s'il n'existe pas de forme *Ok* α telle que $t \equiv_{Ok} \alpha$. On constate dès lors que les termes exceptionnels se propagent implicitement : si un terme est exceptionnel, alors tous ses surtermes le sont aussi.

- □ Dans beaucoup d'exemples, un "*if_ then_ else_* " non strict est très utile. Or la propagation implicite des termes exceptionnels ne permet pas de spécifier d'opérations non strictes au niveau des *Ok*-axiomes.
  En fait, spécifier une opération non stricte relève de la notion de récupération, et par conséquent elle doit être spécifiée au moyen d'axiomes de récupération (pour nous, les

axiomes généralisés). Nous verrons plus loin qu'il n'est pas difficile de spécifier des opérations non strictes grâce aux axiomes généralisés.
Il existe de plus un moyen fort simple d'obtenir la sémantique d'un "*if_ then_ else_* " non strict dès le niveau des *Ok*-axiomes. Les *Ok*-axiomes étant conditionnels, il suffit de remplacer chaque axiome de la forme

$$v \;=\; \text{if } B \textit{ then } w_1 \textit{ else } w_2$$

par :

$$B = True \;\Rightarrow\; v = w_1$$
$$B = False \;\Rightarrow\; v = w_2$$

Il nous reste à définir la validation de l'étiquetage d'exceptions (**Lbl-Ax**) et la validation des axiomes généralisés (**Gen-Ax**). Ces deux notions de validation seront particulièrement simples, car elles se définissent directement dans l'exception-algèbre A , et non dans $T_{\Sigma(A)}$.

5. VALIDATION DE Lbl-Ax

Les axiomes d'étiquetage servent à attacher des étiquettes d'exception à certaines valeurs d'une exception-algèbre. Ce qui signifie que **Lbl-Ax** sert à spécifier des propriétés relatives aux sous-ensembles A_{l_i} d'une exception-algèbre A=(A,$\{A_{l_i}\}$) . Définir la validation des axiomes d'étiquetage est donc très simple.

Définition 13 :
Soit **Lbl-Ax** un ensemble d'axiomes d'étiquetage sur l'exception-signature **Σ**-*exc*. Une **Σ**-*exc*-algèbre A = (A,$\{A_{l_i}\}$) *valide* **Lbl-Ax** si et seulement si pour tout axiome de **Lbl-Ax**

$$[\, t_1 \in l_1 \wedge \cdots \wedge t_n \in l_n \;\wedge\; v_1 = w_1 \wedge \cdots \wedge v_m = w_m \,] \;\Rightarrow\; t \in l$$

et pour toute substitution σ à valeur dans A, on a :
si $\sigma(t_i)$ est élément de A_{l_i} pour tout $i=1..n$, et $\sigma(v_j) = \sigma(w_j)$ pour tout $j=1..m$, alors $\sigma(t)$ est élément de A_l .

Cette définition traduit simplement que pour toute occurrence d'axiome d'étiquetage, si les prémisses sont vérifiés dans A, alors la conclusion doit l'être aussi.

Exemples 10 :
Reprenons les exemples donnés en exemple 5 (section 1), avec l'exception-spécification des entiers naturels bornés que nous avons développée durant le chapitre III (étiquetage d'exceptions de l'exemple 3) :

$$\begin{aligned}
&&&pred(0) \in NEGATIVE\text{-}NUMBER\\
n \in NEGATIVE\text{-}NUMBER &\;\Rightarrow\;& &pred(n) \in NEGATIVE\text{-}NUMBER\\
n < m \;=\; True &\;\Rightarrow\;& &(n - m) \in NEGATIVE\text{-}NUMBER\\
&&&succ(Maxint) \in TOO\text{-}LARGE\text{-}NUMBER\\
n \in TOO\text{-}LARGE\text{-}NUMBER &\;\Rightarrow\;& &succ(n) \in TOO\text{-}LARGE\text{-}NUMBER\\
n \in TOO\text{-}LARGE\text{-}NUMBER &\;\Rightarrow\;& &(n + 0) \in TOO\text{-}LARGE\text{-}NUMBER\\
succ(n)+m \in TOO\text{-}LARGE\text{-}NUMBER &\;\Rightarrow\;& &n+succ(m) \in TOO\text{-}LARGE\text{-}NUMBER\\
n \in TOO\text{-}LARGE\text{-}NUMBER &\;\Rightarrow\;& &(n \times 1) \in TOO\text{-}LARGE\text{-}NUMBER\\
(n\times m + n) \in TOO\text{-}LARGE\text{-}NUMBER &\;\Rightarrow\;& &(n \times succ(m)) \in TOO\text{-}LARGE\text{-}NUMBER\\
&&&(n \text{ div } 0) \in DIV\text{-}BY\text{-}0\text{-}ERROR
\end{aligned}$$

□ L'exception-algèbre des termes fermés, T_{Σ}, munie de

$$T_{\Sigma,Ok} = T_{\Sigma,NEGATIVE\text{-}NUMBER} = T_{\Sigma,TOO\text{-}LARGE\text{-}VALUE} = T_{\Sigma,DIV\text{-}BY\text{-}0\text{-}ERROR} = \emptyset$$

ne valide évidemment pas **Lbl-Ax** ; par exemple le terme fermé *pred(0)* n'est pas un élément de $T_{\Sigma,NEGATIVE\text{-}NUMBER}$.

□ L'exception algèbre des entiers relatifs **Z**, avec *z div 0 = Maxint* pour tout *z*, et munie de :

$$\mathbf{Z}_{Ok} = [0, Maxint]$$
$$\mathbf{Z}_{NEGATIVE\text{-}NUMBER} =]-\infty, 0[$$
$$\mathbf{Z}_{TOO\text{-}LARGE\text{-}NUMBER} =]Maxint, +\infty[$$
$$\mathbf{Z}_{DIV\text{-}BY\text{-}0\text{-}ERROR} = \{Maxint\}$$

valide clairement **Lbl-Ax**. On constate en effet que chacun des axiomes d'étiquetage précédents est vrai dans cette algèbre.

□ De même, l'exception-algèbre $\mathbf{N} \cup \{crash\}$ avec : *pred(0) = crash = n div 0* , toute opération appliquée à *crash* retourne *crash*, et munie de :

$$\mathbf{N}_{Ok} = [0, Maxint]$$
$$\mathbf{N}_{NEGATIVE\text{-}NUMBER} = \{crash\}$$
$$\mathbf{N}_{TOO\text{-}LARGE\text{-}NUMBER} =]Maxint, +\infty[$$
$$\mathbf{N}_{DIV\text{-}BY\text{-}0\text{-}ERROR} = \{crash\}$$

valide clairement chacun des axiomes d'étiquetage précédents.

Remarque 4 :
Rappelons que **Lbl-Ax** sert seulement à imposer l'étiquetage de certaines valeurs d'une exception-algèbre. Il ne crée aucune valeur erronée. Les sous-ensembles A_l , avec $l \in \mathbf{L}$, ne sont pas nécessairement disjoints de A_{Ok} .
Si **Lbl-Ax** contient un axiome qui étiquette une valeur *Ok*, par exemple un axiome de la forme "$0 \in UNE\text{-}ETIQUETTE$", alors la valeur *Ok* en question (ici *0*) est *récupérée* aussitôt qu'elle a été étiquetée, de manière immédiate. Ceci arrive assez souvent ; c'est en particulier le cas du second exemple (**Z**) : la valeur *Maxint* est une valeur *Ok* de **Z**, mais elle est pourtant étiquetée par *DIV-BY-0-ERROR*. En fait, cela traduit intuitivement le fait qu'il existe un terme étiqueté par *DIV-BY-0-ERROR* mais ayant été récupéré sur la valeur *Maxint* (dans notre exemple, tous les termes de la forme *z div 0* sont récupérés sur *Maxint*).
En un certain sens, on pourrait dire que l'étiquette *Ok* est "prioritaire" sur toutes les autres (c'est la récupération d'exceptions).

6. VALIDATION DE Gen-Ax

Les axiomes généralisés servent à traiter les valeurs exceptionnelles. Ils peuvent amalgamer plusieurs valeurs erronées entre elles, ou récupérer certaines valeurs exceptionnelles sur des valeurs *Ok*. On constate donc que les axiomes généralisés doivent pouvoir manipuler *toutes* les valeurs d'une exception-algèbre (aussi bien les valeurs erronées que les valeurs *Ok* ou celles incomplètement spécifiées).
Ainsi, les axiomes généralisés ont une sémantique qui couvre toute l'algèbre. Définir la validation des axiomes généralisés est donc très simple :

Définition 14 :
Soit **Gen-Ax** un ensemble d'axiomes généralisés sur l'exception-signature **Σ**-*exc*. Une **Σ**-*exc*-algèbre $A = (A, \{A_{l_i}\})$ *valide* **Gen-Ax** si et seulement si pour tout axiome de **Gen-Ax**

$$[\, t_1 \in l_1 \wedge \cdots \wedge t_n \in l_n \ \wedge \ v_1 = w_1 \wedge \cdots \wedge v_n = w_n \,] \quad \Rightarrow \quad v = w$$

et pour toute substitution σ à valeur dans *A*, on a :
si $\sigma(t_i)$ est élément de A_{l_i} pour tout $i=1..n$, et $\sigma(v_j) = \sigma(w_j)$ pour tout $j=1..m$, alors $\sigma(v)$ est égal à $\sigma(w)$ dans A.

Cette définition traduit simplement que pour toute occurrence d'axiome généralisé, si les prémisses sont vérifiés dans A, alors la conclusion doit l'être aussi.

Exemple 11 :
Avec les axiomes généralisés donnés en exemple 4 :

$n \in NEGATIVE\text{-}NUMBER$	$\Rightarrow$	n	=	$crash$
		$n\ div\ 0$	=	$crash$
		$succ(crash)$	=	$crash$
		$pred(crash)$	=	$crash$
		$crash + n$	=	$crash$
		$crash - n$	=	$crash$
		$n - crash$	=	$crash$
		$crash \times n$	=	$crash$
		$crash\ div\ n$	=	$crash$
		$n\ div\ crash$	=	$crash$
		$n + m$	=	$m + n$
		$n \times m$	=	$m \times n$
$n \in TOO\text{-}LARGE\text{-}NUMBER \wedge m \in Ok$	$\Rightarrow$	$n < m$	=	$False$
$n \in Ok \wedge m \in TOO\text{-}LARGE\text{-}NUMBER$	$\Rightarrow$	$n < m$	=	$True$
$succ(n) \in TOO\text{-}LARGE\text{-}NUMBER$ $succ(m) \in TOO\text{-}LARGE\text{-}NUMBER$	$\Rightarrow$	$succ(n) < succ(m)$	=	$n < m$
$succ(n) \in TOO\text{-}LARGE\text{-}NUMBER$	$\Rightarrow$	$pred(succ(n))$	=	n
$n \in TOO\text{-}LARGE\text{-}NUMBER$	$\Rightarrow$	$n + 0$	=	n
$n + succ(m) \in TOO\text{-}LARGE\text{-}NUMBER$	$\Rightarrow$	$n + succ(m)$	=	$succ(n) + m$
$n \in TOO\text{-}LARGE\text{-}NUMBER$	$\Rightarrow$	$n - 0$	=	n
$n \in TOO\text{-}LARGE\text{-}NUMBER$	$\Rightarrow$	$n - succ(m)$	=	$pred(n) - m$
$n \in TOO\text{-}LARGE\text{-}NUMBER$	$\Rightarrow$	$n \times 0$	=	0
$n \times succ(m) \in TOO\text{-}LARGE\text{-}NUMBER$	$\Rightarrow$	$n \times succ(m)$	=	$(n \times m) + n$
$r < m = True$	$\Rightarrow$	$((n \times m) + r)\ div\ m$	=	n

On constate clairement que :

- □ Notre exemple des termes fermés ne valide pas **Gen-Ax**, par exemple parce que les termes $n+m$ et $m+n$ sont différents chaque fois que l'on va instancier *m* et *n* par des termes distincts.
- □ Notre exemple des entiers relatifs (avec $z\ div\ 0 = Maxint$) ne valide pas non plus **Gen-Ax** ; par exemple parce que *pred(pred(0))* et *pred(0)* sont distincts dans **Z**, et ne peuvent donc pas être tous deux égaux à *crash*.
- □ Par contre, notre exemple $\mathbf{N} \cup \{crash\}$ valide clairement ces axiomes généralisés. Il en résulte que c'est le seul parmi ces trois exemples qui valide notre exception-spécification (**SPEC**) des entiers naturels bornés. C'est une **SPEC**-algèbre.

Les **SPEC**-algèbres étant définies comme suit :

Définition 15 :
Soit **SPEC** = <**Σ**-*exc*,**Ok-Frm**,**Ok-Ax**,**Lbl-Ax**,**Gen-Ax**> une exception-spécification. Une **Σ**-*exc*-algèbre A est une **SPEC**-algèbre si et seulement si elle valide **Ok-Frm**, **Ok-Ax**, **Lbl-Ax** et **Gen-Ax**.

Revenons à la sémantique des axiomes généralisés. Une petite remarque s'impose : la sémantique des axiomes généralisés est définie sur *toutes* les valeurs d'une exception-algèbre. Elle peut aussi s'appliquer si les deux membres de l'égalité sont *Ok*. Il faut donc manipuler les axiomes généralisés en prenant bien garde de ne pas créer d'inconsistances. Ce sont les prémisses de chaque axiome généralisé qui doivent limiter son application aux cas réellement souhaités ; si on laisse les prémisses vides (comme l'axiome de commutativité de l'addition dans notre exemple), alors l'axiome ne s'appliquera pas seulement aux valeurs exceptionnelles ou erronées, mais aussi aux valeurs *Ok*.
Cette remarque est parfois utile pour limiter le nombre d'axiomes d'une exception-spécification. Par exemple, les axiomes définissant l'opération "+"

$$\begin{aligned} n + 0 &= n \\ n + succ(m) &= succ(n) + m \end{aligned}$$

sont vrais partout (aussi bien sur les valeurs *Ok* que les valeurs erronées, même si *m* ou *n* est égal à *crash*). On peut donc les spécifier dans **Gen-Ax** et les omettre dans **Ok-Ax**.
On pourra effectuer cette simplification chaque fois qu'une opération peut être *complètement* définie uniquement à l'aide des axiomes généralisés : il est inutile de la redéfinir au niveau des *Ok*-axiomes. On gagne ainsi sur la taille des spécifications (mais il n'est pas certain que l'on gagne en lisibilité ...).

Nous allons maintenant développer l'aspect catégoriel des exception-algèbres, et ses rapports avec la notion de congruence.

Chapitre V :

La catégorie

des exception-algèbres

---

1. DEFINITIONS ET RESULTATS ELEMENTAIRES

Le chapitre précédent nous a permis de définir les exception-algèbres sur une exception-signature **Σ**-*exc*, et les exception-algèbres qui valident une exception-spécification **SPEC**. Nous connaissons donc déjà la classe des **Σ**-*exc*-algèbres et la sous-classe des **SPEC**-algèbres. Pour définir la *catégorie* des **Σ**-*exc*-algèbres et celle des **SPEC**-algèbres, il suffit de définir les *exception-morphismes*.
Cette section donne les définitions des catégories Alg(**Σ**-*exc*) , Alg(**SPEC**) , Gen(**Σ**-*exc*) et Gen(**SPEC**) ; ainsi que quelques propriétés immédiates.

Définition 16 :
Soient A=(*A*,{A_{l_i}}) et B=(*B*,{B_{l_i}}) deux exception-algèbres sur la signature **Σ**-*exc*=<**S**,**Σ**,**L**> . Un *exception-morphisme*, *h*: A → B , est un **Σ**-morphisme classique qui préserve les étiquetages. Plus précisément, *h* est une application de *A* dans *B* telle que :

- □ Pour toute sorte *s* de **S**, $h(A_s)$ est inclus dans B_s .
- □ Pour toute opération *op* de **Σ**, on a :
$$h[op(x_1, \ldots, x_n)] = op(h[x_1], \ldots, h[x_n]) \quad \text{(dans } B\text{)}.$$
- □ Pour toute étiquette *l* élément de **L**∪{*Ok*} , $h(A_l)$ est inclus dans B_l .

Cette définition est particulièrement simple ; elle exprime qu'un exception-morphisme est un morphisme classique (i.e. préservant les sortes, et compatible avec les opérations), qui de plus ne "perd" aucun étiquetage (si une valeurs est étiquetée dans l'algèbre de départ, alors son image est aussi étiquetée, avec la même étiquette).

Nota :
Dans le chapitre précédent, nous avons défini l'ensemble des valeurs erronées d'une exception-algèbre A, noté A_{err} . Rappelons que "*err*" *n'est nullement une étiquette*. Il n'est pas élément de **L**∪{*Ok*}.

Soulignons que les valeurs erronées ne sont pas compatibles avec les exception-morphismes. L'image d'une valeur erronée par un exception-morphisme n'est pas nécéssairement erronée. En d'autres termes, les exception-morphismes peuvent porter des récupérations.
On peut par exemple considérer l'exception-algèbre A=**N**, avec $A_{Ok} = [0, Maxint]$ et $A_{TOO\text{-}LARGE\text{-}NUMBER} =]Maxint, \infty[$; et l'exception-algèbre B=$B_{Ok} = [0, Maxint]$, avec $B_{TOO\text{-}LARGE\text{-}NUMBER}=\{Maxint\}$. Il existe alors un exception-morphisme de A dans B, qui récupère $]Maxint,\infty[$ sur *Maxint*. Les valeurs erronées de A sont pourtant $A_{err} =]Maxint, \infty[$ tandis que B_{err} est vide (*Maxint* n'y est pas erroné, c'est un "lieu de récupération").

Il est immédiat que l'identité sur une exception-algèbre est un exception-morphisme. Il est immédiat aussi que la composition de deux exception-morphismes est encore un exception-morphisme. Par conséquent, nous pouvons considérer la catégorie des exception-algèbres, munie de ces morphismes.

Notations 1 :
Etant donnée une exception-signature **Σ**-*exc*, on note Alg(**Σ**-*exc*) la catégorie dont les objets sont les **Σ**-*exc*-algèbres, et les morphismes sont les exception-morphismes.
Etant donnée une exception-spécification **SPEC** au dessus de la signature **Σ**-*exc*, on note Alg(**SPEC**) la sous-catégorie pleine de Alg(**Σ**-*exc*) dont les objets sont les **Σ**-*exc*-algèbres qui valident **SPEC**.
[Sous-catégorie *pleine* : si A et B sont deux **SPEC**-algèbres alors tout **Σ**-*exc*-morphisme de A dans B est encore un **SPEC**-morphisme, i.e. $\mathrm{Hom}_{\mathbf{SPEC}}(A, B) = \mathrm{Hom}_{\mathbf{\Sigma}\text{-}exc}(A, B)$].

Notre premier résultat est immédiat :

Lemme 1 :
Etant donnée une exception-signature **Σ**-*exc*, l'algèbre des termes fermés $T_{\mathbf{\Sigma}}$ munie de

$$T_{\mathbf{\Sigma},l_i} = \varnothing \quad \text{pour toute étiquette } l_i \in \mathbf{L} \cup \{Ok\}$$

est initiale dans Alg(**Σ**-*exc*) . On note $T_{\mathbf{\Sigma}\text{-}exc}$ cette exception-algèbre, et on la nomme *l'exception-algèbre des termes fermés*.

Preuve :
Absolument évidente : puisque l'algèbre des termes fermés est initiale parmi les **Σ**-algèbres classiques, il existe un unique **Σ**-morphisme (classique), *init*, entre $T_{\mathbf{\Sigma}\text{-}exc}$ et toute exception-algèbre A de Alg(**Σ**-*exc*). Il suffit donc de vérifier que ce **Σ**-morphisme est un exception-morphisme ; ce qui est clair puisque pour toute étiquette *l* de **L**$\cup\{Ok\}$, $init(T_{\mathbf{\Sigma},l})=\varnothing$ est toujours inclus dans A_l . □

Le lemme précédent nous permet de définir les exception-algèbres *finiment générées* :

Notations 2 :
Etant donnée une exception-signature **Σ**-*exc*, une exception-algèbre A est dite *finiment générée* si et seulement si l'exception-morphisme

$$init : \quad T_{\mathbf{\Sigma}\text{-}exc} \quad \rightarrow \quad A$$

est surjectif.
On note Gen(**Σ**-*exc*) la sous-catégorie pleine de Alg(**Σ**-*exc*) dont les objets sont les **Σ**-*exc*-algèbres finiment générées.
Etant donnée une exception-spécification **SPEC** au dessus de la signature **Σ**-*exc*, on note Gen(**SPEC**) la sous-catégorie pleine de Alg(**SPEC**) dont les objets sont les **SPEC**-algèbres finiment générées.

L'identité sur $T_{\mathbf{\Sigma}\text{-}exc}$ étant évidemment surjective, l'algèbre initiale $T_{\mathbf{\Sigma}\text{-}exc}$ est elle même finiment générée.
D'autre part, le résultat suivant est tout aussi immédiat que le lemme 1 :

Lemme 2 :
Soit **S** l'exception-algèbre "*triviale*", contenant un unique élément dans chaque sorte, les opérations de **Σ** travaillant donc comme l'impose leur arité, et vérifiant :

$$\mathbf{S}_l = \mathbf{S} \quad \text{pour toute étiquette } l \in \mathbf{L} \cup \{Ok\} .$$

Cette algèbre est terminale dans Alg(**Σ**-*exc*) et Alg(**SPEC**).
De plus, si la signature **Σ** est *sensible* ([3]) alors **S** est aussi terminale dans Gen(**Σ**-*exc*) et Gen(**SPEC**).
Dans le cas où la signature **Σ** n'est pas sensible, Gen(**Σ**-*exc*) et Gen(**SPEC**) possèdent tout de même une exception-algèbre terminale : il suffit de considérer l'exception-algèbre τ telle que τ_s égale $\{s\}$ s'il existe un terme fermé de sorte s, et τ_s égale $\emptyset$ sinon (avec toujours $\tau_l = \tau$ pour tout $l \in \mathbf{L} \cup \{Ok\}$).

Preuve :
Le fait que **S** (ou τ) soit une algèbre terminale est bien connu dans le cas classique. Il suffit dont de vérifier que le **Σ**-morphisme terminal

$$term : \quad A \quad \rightarrow \quad \mathbf{S} \quad (ou\ \tau)$$

est un exception-morphisme (c'est à dire préserve les étiquettes). C'est immédiat.
De plus **S** (ou τ si **Σ** n'est pas sensible) est clairement finiment générée, donc elle est aussi terminale dans Gen(**Σ**-*exc*).
Enfin, il est clair que quelque soit l'exception-spécification **SPEC**, **S** (ou τ) valide cette spécification.
Il en résulte que **S** (ou τ) est élément de Alg(**SPEC**) et Gen(**SPEC**), et y est donc terminale. □

Ces résultats triviaux sont récapitulés comme suit :

Théorème 1 :
Etant données une exception-signature **Σ**-*exc*, et une exception-spécification **SPEC** au dessus de **Σ**-*exc*, on a :

- □ l'exception-algèbre des termes fermés, $T_{\mathbf{\Sigma}\text{-}exc}$, est initiale dans Alg(**Σ**-*exc*) et Gen(**Σ**-*exc*)
- □ les catégories Alg(**Σ**-*exc*), Gen(**Σ**-*exc*), Alg(**SPEC**) et Gen(**SPEC**) possèdent toujours une exception-algèbre terminale.

Ce théorème ne dit rien à propos d'une exception-algèbre initiale dans Alg(**SPEC**) ou Gen(**SPEC**). Nous allons prouver que Alg(**SPEC**) et Gen(**SPEC**) possèdent une algèbre initiale. Pour ce faire, nous allons décrire les rapports entre les exception-morphismes et les *congruences*.

2. CONGRUENCES MINIMALES ET OBJETS INITIAUX

Le théorème que nous prouvons dans cette section est un résultat technique essentiel de notre théorie. Il permettra de prouver que Alg(**SPEC**) et Gen(**SPEC**) possèdent des algèbres initiales, de définir un foncteur adjoint à gauche au foncteur d'oubli (pour une présentation), et par là même de redéfinir les notions classiques de suffisante complétude et de consistance hiérarchique.

Nous commençons par remarquer que les exception-morphismes définissent un préordre dans les exception-algèbres.

(3) i.e. s'il existe au moins un terme fermé dans chaque sorte

2.1. MORPHISMES ET PREORDRE

Définition 17 :
Soient A et B deux exception-algèbres sur une signature **Σ**-*exc*.
On dira que $A \leq B$ si et seulement si il existe un exception-morphisme de A dans B.

Remarque 5 :
Cette définition est parfaitement valide car cette propriété est réflexive (l'identité est un exception-morphisme), et transitive (la composition de deux exception-morphismes est encore un exception-morphisme). La relation "≤" est donc bien un *préordre* dans Alg(**Σ**-*exc*) (et aussi dans Alg(**SPEC**)).
[La suite de cette remarque est anecdotique, elle ne sera pas utilisée pour la suite de l'exposé].
C'est même un *ordre* dans Gen(**Σ**-*exc*) et Gen(**SPEC**), car elle est alors antisymétrique. En effet, la "bonne" définition d'une catégorie est de considérer qu'un objet est défini *à isomorphisme près* (c'est à dire que deux algèbres isomorphes sont en fait égales dans Alg(...) ou Gen(...)). Pour prouver que "≤" est antisymétrique dans Gen(...), il suffit de prouver que s'il existe μ et ν :

$$\begin{aligned} \mu &: A \rightarrow B \\ \nu &: B \rightarrow A \end{aligned}$$

alors μ ou ν est en fait un isomorphisme.
Nous allons même prouver que : $\nu \circ \mu = Identité$

$$\begin{array}{ccccc} A & \xrightarrow{-\mu\rightarrow} & B & \xrightarrow{-\nu\rightarrow} & A \\ \Uparrow & & \Uparrow & & \Uparrow \\ init_A & & init_B & & init_A \\ | & & | & & | \\ T_{\Sigma\text{-}exc} & = & T_{\Sigma\text{-}exc} & = & T_{\Sigma\text{-}exc} \end{array}$$

Du fait que $T_{\Sigma\text{-}exc}$ est initial dans Gen(**Σ**-*exc*), le diagramme précédent est nécéssairement commutatif ($init_A$ est unique). D'autre part, du fait que $init_A$ est surjectif (car A est finiment générée), on a nécéssairement $\nu(\mu(a)) = a$ pour tout élément *a* de A. c.q.f.d.
Là où notre remarque devient intéressante, c'est que "≤" n'est pas un ordre dans Alg(...). Il n'est pas antisymétrique. Il suffit pour le voir de considérer l'exemple d'une signature avec une seule sorte et une seule opération (constante) de cette sorte : *c*.
On constate alors que l'algèbre initiale $\{c\}$ est liée à une algèbre non finiment générée $\{c, \gamma\}$ par le morphisme d'inclusion, et que cette algèbre non finiment générée $\{c, \gamma\}$ se rétracte sur $\{c\}$ par le morphisme $r(c) = r(\gamma) = c$. Pourtant, ces deux algèbres ne sont pas isomorphes.
La relation "≤" est donc un *ordre* dans Gen(...), mais seulement un *préordre* dans Alg(...). Naturellement, cette remarque vaut aussi pour le cas classique (i.e. sans traitement d'exceptions).

Cette remarque faite, reprenons le cours de notre exposé.

2.2. CONGRUENCES MINIMALES

Voici le résultat technique sur lequel repose la plupart de nos résultats ultérieurs :

Théorème 2 :
Soit **SPEC** une exception-spécification sur l'exception-signature **Σ**-*exc*. Soit X une **Σ**-*exc*-algèbre quelconque.
Soit R une relation binaire sur X, compatible avec les sortes de X ; c'est à dire que R est un sous-ensemble quelconque de $\bigcup_{s \in \mathbf{S}} (X_s \times X_s)$.
Il existe une plus petite **SPEC**-algèbre Y vérifiant :

$X \leq Y$, et si xRy alors x est égal à y dans Y.

Ce théorème peut clairement s'écrire aussi :

Théorème 2*bis* :
Soit **SPEC** une exception-spécification sur l'exception-signature **Σ**-*exc*. Soit X une **Σ**-*exc*-algèbre quelconque. Soit R une relation binaire sur X, compatible avec les sortes de X.
Il existe une plus petite congruence $\equiv$ sur X , et il existe une plus petite famille de sous ensembles $\{Y_{l_i}\}$ de $Y = (X/\equiv)$ (indéxée par $\mathbf{L} \cup \{Ok\}$), telles que :

$\equiv$ contient R , et $Y=(Y,\{Y_{l_i}\})$ est une **SPEC**-algèbre

Nous démontrons ce théorème sous sa première forme. La démarche suivie est très classique :
on considère la famille (**F**) de toutes les **SPEC**-algèbres Z compatibles avec R et plus grandes que X . On montre que **F** n'est pas vide ; puis on construit une **Σ**-algèbre Y qui est nécessairement plus petite que toutes les **Σ**-algèbres sous-jacentes (Z) de **F**, ainsi que des sous-ensembles Y_{l_i} inclus dans tous les Z_{l_i} sous-jacents. Il suffit de montrer alors que $Y=(Y,\{Y_{l_i}\})$ est une **SPEC**-algèbre élément de **F**.

Soit donc **F** la famille de toutes les **SPEC**-algèbres $Z=(Z,\{Z_{l_i}\})$ munies d'un exception-morphisme

$$\mu : X \rightarrow Z$$

(en fait, **F** est un ensemble de *couples* $<Z,\mu>$).

F n'est pas vide : elle contient au moins l'exception-algèbre triviale **S** , munie de l'exception-morphisme trivial (qui associe sa sorte à tout élément de X).

Considérons maintenant la **Σ**-*exc*-algèbre $Y=(Y,\{Y_{l_i}\})$ construite comme suit :

- □ Y est le quotient de X défini par :
  $\mu_Y(x) = \mu_Y(y)$ dans Y si et seulement si $\mu(x) = \mu(y)$ pour tout couple $<Z,\mu>$ de **F**.
  Par cette définition de Y et de μ_Y, il est clair que Y est une **Σ**-algèbre et que μ_Y est un **Σ**-morphisme (classique), car la compatibilité avec les opérations de **Σ** est immédiate.

- □ Pour toute étiquette $l_i \in \mathbf{L} \cup \{Ok\}$, et pour tout élément x de X, $\mu_Y(x)$ est élément de Y_{l_i} si et seulement si $\mu(x) \in Z_{l_i}$ pour tout couple $<Z,\mu>$ de **F**.
  Par cette définition des Y_{l_i}, il est immédiat que μ_Y est un exception-morphisme (i.e. préserve les étiquettes de $\mathbf{L} \cup \{Ok\}$), car tous les morphismes μ de **F** préservent les étiquettes.

A supposer que Y soit une **SPEC**-algèbre, c'est clairement un élément de **F**, car l'exception-morphisme μ_Y impose que Y soit supérieure à X, et le fait que tous les $<Z,\mu>$ de **F** soient compatibles avec la relation binaire R impose que $<Y,\mu_Y>$ l'est aussi.
De plus, dans cette hypothèse, $<Y,\mu_Y>$ est nécessairement le plus petit élément de **F**, car, par construction de Y, il existe toujours un exception-morphisme ν de Y dans Z tel que $\nu \; o \; \mu_Y = \mu$.

$$X \quad -\mu_Y\rightarrow \quad Y \quad -\nu\rightarrow \quad Z$$

Par conséquent, il suffit de prouver que Y est une **SPEC**-algèbre pour obtenir que **F** possède un plus petit élément, et par suite clore notre preuve du théorème 2. Nous allons donc maintenant nous attacher à prouver que Y valide **Ok-Frm**, **Ok-Ax**, **Lbl-Ax** et **Gen-Ax**. Plusieurs lemmes sont nécessaires pour atteindre ce but.
Dans toute la suite de cette preuve, on notera $\bar{\nu}$ l'unique **Σ**-morphisme déduit de ν :

$$\bar{\nu} : T_{\mathbf{\Sigma}(Y)} \rightarrow T_{\mathbf{\Sigma}(Z)}$$

Lemme 3 :
Pour tout $<Z,\mu>$ dans **F**, $\bar{\nu}(Ok\text{-}Frm_Y)$ est inclus dans $Ok\text{-}Frm_Z$.

Preuve :
Soit $\overline{Ok\text{-}Frm_Y}$ l'ensemble de tous les termes t de $T_{\Sigma(Y)}$ tels que $\bar{\nu}(t)$ est élément de $Ok\text{-}Frm_Z$ pour tout Z dans **F** :

$$\overline{Ok\text{-}Frm_Y} \;=\; \bigcap_{Z\in\mathbf{F}} \bar{\nu}^{-1}(Ok\text{-}Frm_Z)\ .$$

Le lemme 3 équivaut clairement à :

$$Ok\text{-}Frm_Y \;\subset\; \overline{Ok\text{-}Frm_Y}\ .$$

Or $Ok\text{-}Frm_Y$ est le plus petit sous-ensemble de $T_{\Sigma(Y)}$ satisfaisant les 2 propriétés de la définition 10 (chap. IV, section 3) ; donc il suffit de prouver que $\overline{Ok\text{-}Frm_Y}$ satisfait ces 2 propriétés pour prouver l'inclusion.

□ $\overline{Ok\text{-}Frm_Y}$ contient Y_{Ok} :
ceci résulte du fait que ν est un exception-morphisme, par conséquent $\nu(Y_{Ok})$ est inclus dans Z_{Ok} pour tout Z dans **F**, donc $\bar{\nu}(Y_{Ok}) \subset Z_{Ok}$ (en tant que sous-ensembles de $T_{\Sigma(Z)}$). Si bien que le fait que Z_{Ok} soit toujours inclus dans $Ok\text{-}Frm_Z$ implique que $\bar{\nu}^{-1}(Ok\text{-}Frm_Z)$ contient Y_{Ok} pour tout Z dans **F**.

□ $\overline{Ok\text{-}Frm_Y}$ valide chaque occurrence de déclaration élémentaire de **Ok-Frm**

$$\left.\begin{array}{r} t_1 \in \text{Ok-Frm} \;\wedge\cdots\wedge\; t_m \in \text{Ok-Frm} \\ \wedge\; x_1 \in l_1 \;\wedge\cdots\wedge\; x_n \in l_n \\ \wedge\; v_1 = w_1 \;\wedge\cdots\wedge\; v_p = w_p \end{array}\right\} \;\Rightarrow\; t \in \text{Ok-Frm}$$

car :

- si $\sigma(t_i) \in \overline{Ok\text{-}Frm_Y}$ alors $\bar{\nu}(\sigma(t_i))$ est élément de $Ok\text{-}Frm_Z$ pour tout Z dans **F** (par définition de $\overline{Ok\text{-}Frm_Y}$),
- si $éval_Y[\sigma(x_j)] \in Y_l$ alors on a $éval_Z[\bar{\nu}(\sigma(x_j))] \in Z_l$ (par définitions de Y, ν, $\bar{\nu}$ et *éval*),
- et si $éval_Y[\sigma(v_j)] = éval_Y[\sigma(w_j)]$, alors
  $éval_Z[\bar{\nu}(\sigma(v_j))] = éval_Z[\bar{\nu}(\sigma(w_j))]$ (par définition de Y).

Par conséquent, le fait que $Ok\text{-}Frm_Z$ valide **Ok-Frm** implique que $\bar{\nu}(\sigma(t))$ est élément de $Ok\text{-}Frm_Z$ pour tout Z dans **F**, et donc que $\sigma(t)$ est élément de $\overline{Ok\text{-}Frm_Y}$ (par définition de $\overline{Ok\text{-}Frm_Y}$). Ce qui démontre que $\overline{Ok\text{-}Frm_Y}$ valide la déclaration en question.

Nous avons ainsi prouvé que $\overline{Ok\text{-}Frm_Y}$ satisfait les propriétés de la définition 10 dans $T_{\Sigma(Y)}$. Donc $\overline{Ok\text{-}Frm_Y}$ contient $Ok\text{-}Frm_Y$; ce qui clôt la preuve de notre lemme. □

Il en résulte que Y valide **Ok-Frm** :

Lemme 4 :
Y valide **Ok-Frm**.

Preuve :
Le fait que Y valide **Ok-Frm** signifie que $éval_Y(Ok\text{-}Frm_Y) \subset Y_{Ok}$ (définition 11). Ce qui résulte ici du fait que $éval_Z(Ok\text{-}Frm_Z) \subset Z_{Ok}$ pour tout Z dans **F**, de la définition de Y_{Ok} :

$x \in Y_{Ok}$ si et seulement si $\nu(x) \in Z_{Ok}$ pour tout Z dans **F**

et du fait que $\nu \;o\; éval_Y \;=\; éval_Z \;o\; \bar{\nu}$. □

Prouvons maintenant que Y valide les *Ok*-axiomes :

Lemme 5 :
La **Σ**-*exc*-algèbre Y valide **Ok-Ax**.

Preuve :
Soit $\equiv_{Ok,Y}$ l'*Ok*-congruence minimale dans $T_{\Sigma(Y)}$ associée à **Ok-Ax** (proposition 1 du chapitre précédent). Nous voulons prouver que chaque fois que $t \equiv_{Ok,Y} t'$ dans $T_{\Sigma(Y)}$, on a : $éval_Y(t) = éval_Y(t')$. Par définition de Y, ceci revient à prouver que $\nu(éval_Y(t)) = \nu(éval_Y(t'))$ pour toute algèbre Z dans **F**.
Nous avons déjà remarqué que $\nu \; o \; éval_Y = éval_Z \; o \; \bar{\nu}$. Il suffit donc de prouver que chaque fois que $t \equiv_{Ok,Y} t'$ dans $T_{\Sigma(Y)}$, on a : $éval_Z(\bar{\nu}(t)) = éval_Z(\bar{\nu}(t'))$.
Sachant que toutes les algèbres Z de **F** valident **Ok-Ax**, on sait que si $\bar{\nu}(t) \equiv_{Ok,Z} \bar{\nu}(t')$ dans $T_{\Sigma(Z)}$, alors $éval_Z(\bar{\nu}(t)) = éval_Z(\bar{\nu}(t'))$. Par conséquent, il suffit de prouver que l'image par $\bar{\nu}$ de $\equiv_{Ok,Y}$ est incluse dans $\equiv_{Ok,Z}$.
Sachant enfin que $\equiv_{Ok,Y}$ est la plus petite congruence de $T_{\Sigma(Y)}$ satisfaisant la condition "SI..ALORS" de la proposition 1 du chapitre précédent ; il suffit de démontrer que la congruence $\bar{\nu}^{-1}(\equiv_{Ok,Z})$ satisfait cette condition "SI..ALORS".
Ceci résulte finalement du lemme 3.
En effet, supposons que, pour une occurrence d'*Ok*-axiome

$$[\, v_1 = w_1 \;\wedge\; \cdots \;\wedge\; v_n = w_n \,] \quad \Longrightarrow \quad t = t'$$

(avec $t = op(t_1, \ldots, t_m)$), les trois conditions du "SI" soient vérifiées :

- □ $éval_Y(v_i) = éval_Y(w_i)$ pour tout $i=1..n$
- □ il existe $\alpha_1 \cdots \alpha_m$ ($\in Ok\text{-}Frm_Y$) tels que $t_j \;\; \bar{\nu}^{-1}(\equiv_{Ok,Z}) \;\; \alpha_j$ pour tout $j=1..m$.
- □ il existe α élément de $Ok\text{-}Frm_Y$ tel que $t' \;\; \bar{\nu}^{-1}(\equiv_{Ok,Z}) \;\; \alpha$

On en déduit que, pour toute algèbre Z dans **F** :

- □ $éval_Z(\bar{\nu}(v_i)) = éval_Z(\bar{\nu}(w_i))$ pour tout i
  [Ceci par définition même de Y]
- □ il existe $\beta_1 \cdots \beta_m$ ($\in Ok\text{-}Frm_Z$) tels que $\bar{\nu}(t_j) \equiv_{Ok,Z} \beta_j$ pour tout j .
  [Il suffit de prendre $\beta_j = \bar{\nu}(\alpha_j)$; le lemme 3 nous assure que les β_j sont des éléments de $Ok\text{-}Frm_Z$]
- □ il existe β élément de $Ok\text{-}Frm_Z$ tel que $\bar{\nu}(t') \equiv_{Ok,Z} \beta$
  [Il suffit de prendre $\beta = \bar{\nu}(\alpha)$; le lemme 3 nous assure que β est élément de $Ok\text{-}Frm_Z$]

Maintenant, le fait que toute algèbre Z de **F** valide **Ok-Ax** nous assure que $\equiv_{Ok,Z}$ satisfait la propriété "SI..ALORS" ; si bien que :

$$\bar{\nu}(t) \;\equiv_{Ok,Z}\; \bar{\nu}(t') \quad \text{i.e.} \quad t \;\; \bar{\nu}^{-1}(\equiv_{Ok,Z}) \;\; t' .$$

C'est exactement ce que l'on voulait prouver : la congruence $\bar{\nu}^{-1}(\equiv_{Ok,Z})$ satisfait la propriété "SI..ALORS" ; et par conséquent elle contient $\equiv_{Ok,Y}$. Ce qui clôt la preuve de ce lemme. □

Pour terminer, la validation de **Lbl-Ax** et **Gen-Ax** est très simple à prouver :

Lemme 6 :
La **Σ**-*exc*-algèbre Y valide **Lbl-Ax** et **Gen-Ax**.

Preuve :
Les axiomes de **Lbl-Ax** ou **Gen-Ax** sont de la forme :

$$[\, t_1 \in l_1 \wedge \cdots \wedge t_n \in l_n \ \wedge \ v_1 = w_1 \wedge \cdots \wedge v_m = w_m \,] \ \Rightarrow \ \cdots$$

où la conclusion est soit une appartenance à une étiquette, soit une égalité.
On veut prouver que pour chaque occurrence d'un tel axiome dans Y, si les prémisses sont vérifiées alors la conclusion est vérifiée aussi. Or, si les prémisses sont vérifiées dans Y, alors elles le sont dans toute algèbre Z de **F**, par définition même de Y. Il en résulte que la conclusion est vérifiée dans toute algèbre Z de **F** (car les algèbres de **F** valident **Lbl-Ax** et **Gen-Ax**). Et encore par définition de Y, on en déduit que la conclusion est vérifiée dans Y.
Ce qui prouve ce lemme. □

En récapitulant, nous venons de démontrer que Y est une **SPEC**-algèbre, et que par conséquent elle appartient à la famille **F**. Par construction même de Y, c'est nécessairement le plus petit élément de **F** ; donc **F** possède un plus petit élément ; ce qui clôt la démonstration des théorèmes 2 et 2*bis*.

Remarque 6 :
La **SPEC**-algèbre Y possédant les propriétés de minimalité décrites dans le théorème 2 est clairement unique à isomorphisme près.
Ceci est dû au fait que Y est nécéssairement finiment générée au dessus de X (sinon, on obtiendrait une algèbre strictement plus petite en ne prenant que la partie finiment générée de Y). Du fait qu'elle est finiment générée au dessus de X, le morphisme μ de X dans Y est toujours surjectif. Dès lors, si Y et Y' sont deux telles algèbres minimale, il existe deux morphismes : l'un de Y vers Y', et l'autre de Y' vers Y ; et leur composée est nécéssairement un isomorphisme sur Y (respectivement sur Y'). Donc Y est isomorphe à Y'.

2.3. OBJETS INITIAUX

Le théorème 2 implique que Alg(**SPEC**) est Gen(**SPEC**) possèdent un objet initial. Il suffit de choisir $X=T_{\mathbf{\Sigma}\text{-}exc}$ dans l'énoncé du théorème 2, avec la relation binaire vide : $R=\varnothing$.

Notation 3 :
On note $T_{\mathbf{SPEC}}$ la plus petite **SPEC**-algèbre telle que $T_{\mathbf{\Sigma}\text{-}exc} \leq T_{\mathbf{SPEC}}$.
Cette **SPEC**-algèbre existe, d'après le théorème 2.

$T_{\mathbf{SPEC}}$ est en fait l'algèbre initiale de Alg(**SPEC**) et Gen(**SPEC**) :

Théorème 3 :
La **SPEC**-algèbre $T_{\mathbf{SPEC}}$ est initiale dans Alg(**SPEC**) et Gen(**SPEC**) .

Preuve :
Du fait que la relation binaire R choisie pour définir $T_{\mathbf{SPEC}}$ est vide, $T_{\mathbf{SPEC}}$ est en fait la plus petite **Σ**-*exc*-algèbre plus grande que $T_{\mathbf{\Sigma}\text{-}exc}$ qui valide **SPEC**.
Du fait que $T_{\mathbf{\Sigma}\text{-}exc}$ est initiale dans Alg(**Σ**-*exc*), $T_{\mathbf{\Sigma}\text{-}exc}$ est en fait plus petite que *toute* **Σ**-*exc*-algèbre. Par conséquent, $T_{\mathbf{SPEC}}$ est la plus petite **SPEC**-algèbre. Ce qui signifie que pour toute **SPEC**-algèbre A , il existe un exception-morphisme de $T_{\mathbf{SPEC}}$ dans A.
Il reste à prouver que ce morphisme est unique. Ceci résulte du fait que tout morphisme entre $T_{\mathbf{SPEC}}$ et A résulte de la factorisation de l'unique morphisme initial entre $T_{\mathbf{\Sigma}\text{-}exc}$ et A.
Nous avons donc prouvé que $T_{\mathbf{SPEC}}$ est initial dans Alg(**SPEC**).
Il reste à prouver que $T_{\mathbf{SPEC}}$ est initial dans Gen(**SPEC**) ; c'est à dire que $T_{\mathbf{SPEC}}$ est finiment généré. Mais ceci est immédiat, car dans la démonstration du théorème 2, l'exception-algèbre Y que nous avons construite est un quotient de X ; ce qui se traduit ici par le fait que $T_{\mathbf{SPEC}}$ est un quotient de

$T_{\Sigma\text{-}exc}$, et est donc finiment généré. □

Exemple 12 :
Revenons à la spécification des entiers naturels bornés développée durant le chapitre III ("*Exception-spécifications*"). Sans la recopier ici, rappelons tout de même les faits suivants.
Elle contient la sorte *BOOL* (booléens), avec les opérations *True* et *False* usuelles, et la sorte *NAT* (naturels), avec les opérations *0*, *Maxint*, *crash*, *succ*, *pred*, $<$, $+$, $-$, $\times$ et *div*. Les étiquettes d'exception sont *NEGATIVE-VALUE*, *TOO-LARGE-NUMBER* et *DIV-BY-0-ERROR*.
Les déclarations formes *Ok* caractérisent les valeurs comprises entre *0* et *Maxint*. Les *Ok*-axiomes sont les axiomes habituels décrivant les opérations sur les entiers naturels. Les axiomes d'étiquetage associent *NEGATIVE-VALUE* aux valeurs négatives, *TOO-LARGE-NUMBER* aux termes retournant des valeurs plus grandes que *Maxint*, et *DIV-BY-0-ERROR* à tous ceux de la forme *n div 0* . Enfin les axiomes généralisés récupèrent tous les termes exceptionnels contenant une valeur trop grande dans leur histoire mais dont le résultat final retourne dans les bornes *0..Maxint*, ceci en associant sa forme normale à tout calcul d'opération dont le résultat est plus grand que *Maxint*. Enfin, tous les résultats négatifs sont amalgamés avec la constante *crash*, toute division par 0 retourne *crash*, et toute opération appliquée à *crash* retourne *crash*.
Rappelons que l'exception-algèbre $\mathbf{N}\cup\{crash\}$ décrite en exemple 5, chapitre IV, est définie par : *pred(0)* = *n div 0* = *crash* , toute opération appliquée à *crash* retourne *crash* , et :

$$\mathbf{N}_{Ok} = [0, Maxint]$$
$$\mathbf{N}_{NEGATIVE\text{-}NUMBER} = \{crash\}$$
$$\mathbf{N}_{TOO\text{-}LARGE\text{-}NUMBER} =]Maxint, +\infty[$$
$$\mathbf{N}_{DIV\text{-}BY\text{-}0\text{-}ERROR} = \{crash\}$$

Nous avons prouvé (chapitre IV) que cette exception-algèbre valide la spécification des entiers naturels bornés ; c'est une **SPEC**-algèbre. En fait, elle est initiale dans Alg(**SPEC**).
En effet, les axiomes généralisés et les *Ok*-axiomes impliquent que, d'une part, tous les termes rencontrant une valeur négative dans leur histoire sont amalgamés avec *crash*, et d'autre part, tous les autres termes sont amalgamés avec leur forme normale $succ^i(0)$ (qu'elle soit *Ok* ou non). Il en résulte qu'il existe toujours un **Σ**-morphisme de $\mathbf{N}\cup\{crash\}$ dans une **SPEC**-algèbre quelconque. Il suffit donc de prouver que les sous-ensembles étiquetés de $\mathbf{N}\cup\{crash\}$ sont minimaux (afin que ces **Σ**-morphismes soient des **Σ**-*exc*-morphismes). Mais ceci est clairement déduit des axiomes d'étiquetage de **Lbl-Ax**, et de **Ok-Frm**.

Chapitre VI :

Spécifications hiérarchiques

Dans le formalisme des types abstraits algébriques, on écrit des spécifications *hiérarchiques* grâce à la notion de *présentations*. Etant donnée une spécification déjà faite (dite *prédéfinie*), on peut la compléter afin d'obtenir une spécification plus riche. Ce "morceau de spécification" ajouté à la spécification prédéfinie ($\mathbf{SPEC_1}$) pour en obtenir une autre plus grande ($\mathbf{SPEC_2}$) est appelé une *présentation* au-dessus de $\mathbf{SPEC_1}$.
Du point de vue sémantique, les rapports entre les $\mathbf{SPEC_1}$-algèbres et les $\mathbf{SPEC_2}$-algèbres sont traités grâce à deux foncteurs :

- □ l'un va de Alg($\mathbf{SPEC_2}$) vers Alg($\mathbf{SPEC_1}$), c'est le *foncteur d'oubli*
- □ l'autre va de Alg($\mathbf{SPEC_1}$) vers Alg($\mathbf{SPEC_2}$), c'est le *foncteur de synthèse*.

Ces deux foncteurs ne sont pas inverses l'un de l'autre (...ce serait trop simple !). Il sont en fait liés par une propriété fonctorielle que l'on appelle l'*adjonction* (voir [Ber 86]).

Au cours de ce chapitre, nous allons d'abord nous intéresser au foncteur d'oubli : dans le cadre des exception-signatures, puis dans le cadre général des exception-spécifications. Ensuite, nous allons définir le foncteur de synthèse associé, et démontrer qu'il est adjoint à gauche au foncteur d'oubli.

1. LE FONCTEUR D'OUBLI

1.1. ENRICHISSEMENT DE SIGNATURES

Avant de traiter le foncteur d'oubli de Alg($\mathbf{SPEC_2}$) dans Alg($\mathbf{SPEC_1}$), nous allons d'abord le définir au niveau de deux signatures $\mathbf{\Sigma}\text{-}exc_1 \subset \mathbf{\Sigma}\text{-}exc_2$.

Définition 18 :
Soient $\mathbf{\Sigma}\text{-}exc_1 =< \mathbf{S_1}, \mathbf{\Sigma_1}, \mathbf{L_1} >$ et $\mathbf{\Sigma}\text{-}exc_2 =< \mathbf{S_2}, \mathbf{\Sigma_2}, \mathbf{L_2} >$ deux exception-signatures telles que $\mathbf{\Sigma}\text{-}exc_1 \subset \mathbf{\Sigma}\text{-}exc_2$. Le *foncteur d'oubli* U de Alg($\mathbf{\Sigma}\text{-}exc_2$) dans Alg($\mathbf{\Sigma}\text{-}exc_1$) est défini comme suit :

- □ Pour toute $\mathbf{\Sigma}\text{-}exc_2$-algèbre $B=(B,\{B_{l_i}\})$, $U(B)$ est la $\mathbf{\Sigma}\text{-}exc_1$-algèbre $A=(A,\{A_{l_i}\})$ suivante :
  - A est le sous-ensemble de B correspondant aux sortes de $\mathbf{S_1}$ (c'est-à-dire que l'on enlève les supports de B associés aux sortes de $\mathbf{S_2 - S_1}$)

- pour chaque étiquette $l \in \mathbf{L_1} \cup \{Ok\}$, A_l est égal au sous-ensemble de B_l associé aux sortes de $\mathbf{S_1}$, c'est-à-dire que A_l est égal à $B_l \cap A$.
- les opérations de $\mathbf{\Sigma_1}$ s'appliquent dans *A* exactement comme dans *B*.

Il est immédiat que A est une $\mathbf{\Sigma}$-exc_1-algèbre.

□ Pour tout $\mathbf{\Sigma}$-exc_2-morphisme $\mu: B \to B'$, $U(\mu)$ est le $\mathbf{\Sigma}$-exc_1-morphisme μ, restreint à $U(B)$ et corestreint à $U(B')$.
$U(\mu)$ est clairement un $\mathbf{\Sigma}$-exc_1-morphisme, ceci résulte directement des définitions des exception-morphismes, de $U(B)$ et de $U(B')$.

Voici un exemple simple de foncteur d'oubli :

Exemple 13 :
Soit la signature suivante de *BOOL* : $\mathbf{\Sigma}$-$exc_1 = <\{BOOL\},\{True,False\},\emptyset>$; et soit la signature suivante de *NAT* :
$\mathbf{\Sigma}$-$exc_2 = \mathbf{\Sigma}$-$exc_1 + <\{NAT\},\{0,succ,pred,eq?\},\{NEGATIVE\text{-}VALUE,TOO\text{-}LARGE\text{-}NUMBER\}>$
Considérons la $\mathbf{\Sigma}$-exc_2-algèbre B = $(\mathbf{N} \cup \{crash\}) \cup \{True,False,Maybe\}$ avec

$$B_{Ok} = [0, Maxint] \; o'u' \{True, False\}$$
$$B_{NEGATIVE\text{-}VALUE} = \{crash\}$$
$$B_{TOO\text{-}LARGE\text{-}NUMBER} =]Maxint, +\infty[$$

et :

pred(0) est égal à *crash*
toute opération à valeur entiere appliquée à *crash* retourne *crash*
eq?(crash,crash) égale *True*
eq?(crash,n) et *eq?(n,crash)* égalent *Maybe* ($n \neq crash$)
les autres opérations agissant comme usuellement .

On constate que B est une $\mathbf{\Sigma}$-exc_2-algèbre finiment générée (car même la valeur *Maybe* est atteinte par un terme fermé : *eq?(crash,0)* par exemple).
L'oubli de cette $\mathbf{\Sigma}$-exc_2-algèbre B sur la signature booléenne $\mathbf{\Sigma}$-exc_1 est la $\mathbf{\Sigma}$-exc_1-algèbre $U(B) = A = \{True,False,Maybe\}$ avec $A_{Ok} = \{True,False\}$. On constate que A n'est pas $\mathbf{\Sigma}$-exc_1-finiment générée, bien que B soit $\mathbf{\Sigma}$-exc_2-finiment générée (évidemment, le même phénomène peut se rencontrer dans le formalisme classique (ADJ)).
Considérons maintenant la $\mathbf{\Sigma}$-exc_2-algèbre B' dont la description est presque la même que celle de B, sauf que B' ne contient pas la valeur booléenne *Maybe*, et *eq?(crash,n)=eq?(crash,n)=False* chaque fois que *n* n'est pas égal à *crash*.
Il existe clairement un $\mathbf{\Sigma}$-exc_2-morphisme $\mu: B \to B'$ qui envoie *Maybe* sur *False* (i.e. qui *récupère* la valeur erronée *Maybe* de B en la valeur *Ok False* de B').
Le $\mathbf{\Sigma}$-exc_1-morphisme $U(\mu)$ est alors simplement le morphisme de $\{True,False,Maybe\}$ dans $\{True,False\}$ tel que $U(\mu)(False) = U(\mu)(Maybe) = False$.

Nous venons donc de définir le foncteur d'oubli de Alg($\mathbf{\Sigma}$-exc_2) dans Alg($\mathbf{\Sigma}$-exc_1). Comme dans le formalisme classique (i.e. sans traitement d'exceptions) nous remarquons que ce foncteur ne peut pas être restreint aux catégories Gen(...) . Plus exactement, on peut considérer le foncteur d'oubli de Gen($\mathbf{\Sigma}$-exc_2) dans Alg($\mathbf{\Sigma}$-exc_1), mais on ne peut pas le corestreindre à Gen($\mathbf{\Sigma}$-exc_1).

Rappelons qu'étant donnée une exception-spécification **SPEC** sur une signature $\mathbf{\Sigma}$-*exc*, Alg(**SPEC**) est une sous-catégorie de Alg($\mathbf{\Sigma}$-*exc*). Nous allons maintenant étudier la restriction du foncteur d'oubli aux catégories Alg(**SPEC**).

1.2. LES PRESENTATIONS

Plus généralement que l'enrichissement d'exception-signatures, on peut considérer deux exception-spécifications $\mathbf{SPEC_1}$ et $\mathbf{SPEC_2}$ telles que $\mathbf{SPEC_1}$ est incluse (membre à membre) dans $\mathbf{SPEC_2}$ (en particulier $\mathbf{\Sigma}$-exc_1 est incluse dans $\mathbf{\Sigma}$-exc_2). Le "morceau de spécification" ajouté à $\mathbf{SPEC_1}$ pour obtenir $\mathbf{SPEC_2}$ est appelé une *présentation*.

Définition 19 :
Soit $\mathbf{SPEC_1}$ une exception-spécification. Une *présentation* au-dessus de $\mathbf{SPEC_1}$ est un n-uplet

$$\mathbf{PRES} = < \mathbf{S} , \mathbf{\Sigma} , \mathbf{L} , \mathbf{Ok\text{-}Frm} , \mathbf{Ok\text{-}Ax} , \mathbf{Lbl\text{-}Ax} , \mathbf{Gen\text{-}Ax} >$$

tel que :

$$\mathbf{S_1} \cap \mathbf{S} \quad \text{est vide}$$
$$\mathbf{\Sigma_1} \cap \mathbf{\Sigma} \quad \text{est vide}$$
$$\mathbf{SPEC_2} = \mathbf{SPEC_1} + \mathbf{PRES} \quad \text{est encore une exception-spécification.}$$

Cette définition est la même que dans le cas classique. La spécification $\mathbf{SPEC_1}$ est souvent appelée la spécification *prédéfinie* ; les sortes, opérations et étiquettes de **PRES** sont souvent appelées les sortes, opérations et étiquettes *d'intérêt*.

Du fait que Alg($\mathbf{SPEC_2}$) est une sous-catégorie de Alg($\mathbf{\Sigma}$-exc_2), on peut considérer la restriction du foncteur d'oubli :

$$U : \text{Alg}(\mathbf{SPEC_2}) \rightarrow \text{Alg}(\mathbf{\Sigma}\text{-}exc_1) .$$

Naturellement, ce qui nous intéresse, c'est de corestreindre U à Alg($\mathbf{SPEC_1}$).

Théorème 4 :
Pour toute $\mathbf{SPEC_2}$-algèbre B , la $\mathbf{\Sigma}$-exc_1-algèbre A=U(B) valide $\mathbf{SPEC_1}$.

Preuve :
La validation de $\mathbf{Ok\text{-}Frm_1}$ résulte immédiatement du fait que $Ok\text{-}Frm_A$ est inclus dans $Ok\text{-}Frm_B$ car $\mathbf{Ok\text{-}Frm_1}$ est inclus dans $\mathbf{Ok\text{-}Frm_2}$. De même, la validation de $\mathbf{Lbl\text{-}Ax_1}$ et $\mathbf{Gen\text{-}Ax_1}$ résulte immédiatement du fait qu'ils sont inclus dans $\mathbf{Lbl\text{-}Ax_2}$ et $\mathbf{Gen\text{-}Ax_2}$ respectivement. Il suffit donc de prouver la validation de $\mathbf{Ok\text{-}Ax_1}$.
Du fait que $Ok\text{-}Frm_A \subset Ok\text{-}Frm_B$ il découle facilement que la congruence $\equiv_{Ok,B} \cap (T_{\mathbf{\Sigma_1}(A)} \times T_{\mathbf{\Sigma_1}(A)})$ satisfait la propriété "SI..ALORS" de la proposition 1.
Il en résulte que $\equiv_{Ok,A}$ est incluse dans $\equiv_{Ok,B}$. Or la validation des *Ok*-axiomes par B signifie que le morphisme *éval* est compatible avec $\equiv_{Ok,B}$, d'où il résulte qu'il est en particulier compatible avec $\equiv_{Ok,A}$; donc A=U(B) valide les Ok-axiomes.
Ce qui clôt la preuve du théorème. □

Nous pouvons donc considérer le foncteur d'oubli de Alg($\mathbf{SPEC_2}$) dans Alg($\mathbf{SPEC_1}$). Rappelons que l'oubli d'une $\mathbf{SPEC_2}$-algèbre finiment générée n'est pas nécessairement $\mathbf{SPEC_1}$-finiment générée.

Voici un exemple de présentation, celui des tableaux bornés. On pourra en trouver d'autres dans l'annexe 3.

Exemple 14 :
Soit $\mathbf{SPEC_1}$ une spécification des entiers naturels bornés et des booléens, avec les opérations "<" et "*eq?*" . On peut spécifier les *tableaux bornés* d'entiers naturels bornés comme suit (on supposera que la borne des tableaux, *Maxrange*, est plus petite que celle des entiers, *Maxint*) :

S = { *ARRAY* }

Σ = { *Maxrange* , *create* , _ [_]:=_ , _ [_] } (avec les arités usuelles)

L = { *NEGATIVE-RANGE* , *RANGE-TOO-LARGE* , *NOT-INITIALIZED* }

Ok-Frm :

$$\left.\begin{array}{r} t \in \text{Ok-Frm} \wedge n \in \text{Ok-Frm} \\ \wedge\ i \in \text{Ok-Frm} \wedge i < succ(Maxrange) = True \end{array}\right\} \Rightarrow \quad t[i]:=n \in \text{Ok-Frm}$$

$$create \in \text{Ok-Frm}$$

Ok-Ax :

$$\begin{array}{lrcl} eq?(i,j) = False & \Rightarrow & (t[i]:=n)[j]:=m & = & (t[j]:=m)[i]:=n \\ & & (t[i]:=n)[i]:=m & = & t[i]:=m \\ & & (t[i]:=n)[i] & = & n \\ & & Maxrange & = & succ^{Maxrange}(0) \quad (^4) \end{array}$$

Lbl-Ax :

$$\begin{array}{rcl} & & \text{create}[i] \in \text{NOT-INITIALIZED} \\ t[i] \in NOT\text{-}INITIALIZED \wedge eq?(i,j) = False & \Rightarrow & (t[j]:=n)[i] \in NOT\text{-}INITIALIZED \\ Maxrange < i = True & \Rightarrow & t[i] \in RANGE\text{-}TOO\text{-}LARGE \\ Maxrange < i = True & \Rightarrow & t[i]:=n \in RANGE\text{-}TOO\text{-}LARGE \\ i \in NEGATIVE\text{-}VALUE & \Rightarrow & t[i] \in NEGATIVE\text{-}RANGE \\ i \in NEGATIVE\text{-}VALUE & \Rightarrow & t[i]:=n \in NEGATIVE\text{-}RANGE \end{array}$$

Gen-Ax : ... récupérations, ad libitum ...
Par exemple, si l'on veut récupérer l'accès à un tableau erroné, pourvu que la dernière valeur affectée au rang en question soit *Ok*, on peut spécifier **Gen-Ax** comme suit :

$$\begin{array}{rcl} eq?(i,j) = False & \Rightarrow & (t[i]:=n)[j] = t[j] \\ n \in Ok \wedge 0 \le j \le Maxrange = True & \Rightarrow & (t[j]:=n)[j] = n \end{array}$$

Si l'on préfère récupérer directement tout tableau ayant reçu une valeur erronée à un certain rang, mais ensuite une valeur *Ok* au même rang, il suffit de spécifier :

$$n \in Ok \quad \Rightarrow \quad (t[i]:=x)[i]:=n \;=\; t[i]:=n$$

2. LE FONCTEUR DE SYNTHESE

2.1. DEFINITION

Le foncteur d'oubli permet, partant d'une **SPEC$_2$**-algèbre, de ne conserver que la partie de sa structure concernant la signature prédéfinie. Le foncteur de synthèse permet au contraire, partant d'une algèbre prédéfinie, de la "compléter" de manière minimale pour obtenir une **SPEC$_2$**-algèbre.
Le foncteur de synthèse est défini d'une manière très similaire au cas classique.

(4) l'exposant "*Maxrange*" est considéré de manière méta dans le second membre de cette équation, il signifie l'application de l'opération *succ Maxrange* fois.

Définition 20 :
Soit **PRES** une présentation au-dessus de l'exception-spécification prédéfinie $\mathbf{SPEC_1}$. Soit $\mathbf{SPEC_2} = \mathbf{SPEC_1} + \mathbf{PRES}$.
Le *foncteur de synthèse* de Alg($\mathbf{SPEC_1}$) dans Alg($\mathbf{SPEC_2}$), noté F, est défini comme suit :

□ Pour toute $\mathbf{SPEC_1}$-algèbre A , on peut considérer la $\mathbf{\Sigma_2}$-algèbre $T_{\mathbf{\Sigma_2}(A)}$ comme une $\mathbf{\Sigma}$-exc_2-algèbre en posant :

$$T_{\mathbf{\Sigma_2}(A),l_i} = A_{l_i} \quad \text{pour toute étiquette } l_i \in \mathbf{L_1} \cup \{Ok\}$$
$$T_{\mathbf{\Sigma_2}(A),l_i} = \varnothing \quad \text{pour toute étiquette } l_i \in \mathbf{L_2} - \mathbf{L_1}$$

(rappelons que A, et par conséquent chacun des A_{l_i}, est un sous-ensemble de $T_{\mathbf{\Sigma_1}(A)}$ lui même inclus dans $T_{\mathbf{\Sigma_2}(A)}$)
D'autre part, puisque $T_{\mathbf{\Sigma_1}(A)}$ est un sous-ensemble de $T_{\mathbf{\Sigma_2}(A)}$; on peut considérer la relation binaire R sur $T_{\mathbf{\Sigma_2}(A)}$ définie par :

$$t \; R \; t' \text{ si et seulement si } t \text{ et } t' \in T_{\mathbf{\Sigma_1}(A)} \text{ et } \acute{e}val_A(t) = \acute{e}val_A(t')$$

(où $\acute{e}val_A$ est le morphisme d'évaluation de $T_{\mathbf{\Sigma_1}(A)}$ dans A).
Par définition, F(A) est la plus petite $\mathbf{SPEC_2}$-algèbre compatible avec R telle que $T_{\mathbf{\Sigma_2}(A)} \leq F(\mathrm{A})$; cette $\mathbf{\Sigma}$-exc_2-algèbre existe d'après le théorème 2 du chapitre V précédent.
Ceci signifie que F(A) est égale au quotient de $T_{\mathbf{\Sigma_2}(A)}$ par la plus petite congruence $\equiv_{R,\mathbf{SPEC_2}}$ contenant R et telle que $(T_{\mathbf{\Sigma_2}(A)} / \equiv_{R,\mathbf{SPEC_2}})$, munie des sous-ensembles étiquetés minimaux, valide $\mathbf{SPEC_2}$.

□ Pour tout $\mathbf{\Sigma}$-exc_1-morphisme entre deux $\mathbf{SPEC_1}$-algèbres

$$\nu : \mathrm{A} \rightarrow \mathrm{A'}$$

$F(\nu)$ est l'unique $\mathbf{\Sigma}$-exc_2-morphisme de F(A) dans F(A'), déduit du $\mathbf{\Sigma}$-exc_2-morphisme $\bar{\nu}$:

$$\bar{\nu} : T_{\mathbf{\Sigma_2}(A)} \rightarrow T_{\mathbf{\Sigma_2}(A')}$$

($F(\nu)$ existe car F(A) est *le plus petit* quotient de $T_{\mathbf{\Sigma_2}(A)}$ tel que F(A) soit une $\mathbf{SPEC_2}$-algèbre, et F(A') est une $\mathbf{SPEC_2}$-algèbre).

Exemple 15 :
Soit la (*NAT+BOOL*)-algèbre A = ({*crash*}∪**N**)∪{*True*,*False*} .
Si l'on considère la présentation des tableaux bornés donnée dans l'exemple 14 précédent, la (*NAT+BOOL+ARRAY*)-algèbre F(A) est décrite comme suit :

□ Les tableaux *Ok* sont les tableaux au sens usuels, n'ayant reçu que des affectations d'éléments *Ok* entre les bornes *0...Maxint*.
(Si l'on considère le deuxième jeu d'axiomes généralisés de l'exemple 14, alors les tableaux *Ok* contiennent aussi ceux ayant subi dans leur histoire une affectation erronée mais écrasée ensuite par une valeur *Ok*).

□ Les tableaux erronés sont :

- tous ceux ayant subi une affectation de valeur erronée (par propagation d'erreurs)
- tous ceux ayant reçu une affectation à un rang négatif, auquel cas le sous-terme correspondant à cette affectation répond à l'étiquette d'exception *NEGATIVE-RANGE*
- tous ceux ayant reçu une affectation à un rang supérieur à *Maxrange*, auquel cas le sous-terme correspondant à cette affectation répond à l'étiquette *RANGE-TOO-LARGE*

□ Des valeurs erronées ont été ajoutées aux sortes prédéfinies :

- tout accès à un tableau erroné retourne une nouvelle valeur erronée de sorte *NAT* (à moduler en fonction du type de récupération spécifié par les axiomes généralisés)
- tout accès hors des bornes d'un tableau retourne une nouvelle valeur, qui répond à l'étiquette d'exception *NEGATIVE-RANGE* ou *RANGE-TOO-LARGE* selon le cas
- de même tout accès à un rang non initialisé d'un tableau retourne une nouvelle valeur erronée, étiquetée par *NOT-INITIALIZED*
- ces nouvelles valeurs de sorte *NAT* induisent à leur tour de nouvelles valeurs booléennes erronées (via les opérations "*eq?*" et "<")

Bien sûr, on aurait pu choisir les axiomes généralisés de manière à amalgamer toutes les nouvelles valeurs erronées de sortes *NAT* sur la constante *crash*, ou au contraire les récupérer sur 0 ...etc ... si bien que l'on n'ajoute alors aucune nouvelle valeur prédéfinie erronée. Notons cependant que même sans cela, la présentation *ARRAY* est intuitivement suffisamment complète, car notre présentation spécifie suffisamment d'étiquettes d'exception pour que les nouvelles valeurs entières ou booléennes soient erronées (nous voyons donc sur cet exemple qu'il faudra redéfinir la notion de suffisante complétude).

2.2. ADJONCTION

Nous démontrons ici que les foncteurs d'oubli et de synthèse sont adjoints : le foncteur de synthèse est adjoint à gauche au foncteur d'oubli (et par conséquent, le foncteur d'oubli est adjoint à droite au foncteur de synthèse).

Théorème 5 :
Soit **PRES** une présentation au-dessus de l'exception-spécification prédéfinie $\mathbf{SPEC_1}$. Soit $\mathbf{SPEC_2} = \mathbf{SPEC_1} + \mathbf{PRES}$.
Le foncteur de synthèse F: Alg($\mathbf{SPEC_1}$) $\rightarrow$ Alg($\mathbf{SPEC_2}$) est adjoint à gauche au foncteur d'oubli U: Alg($\mathbf{SPEC_2}$) $\rightarrow$ Alg($\mathbf{SPEC_1}$) .

Preuve :
Rappelons que ceci signifie que pour toute $\mathbf{SPEC_1}$-algèbre A , et pour toute $\mathbf{SPEC_2}$-algèbre B , $\mathrm{Hom}_{\mathbf{SPEC_1}}(\mathrm{A}, U(\mathrm{B}))$ est en *bijection naturelle* avec $\mathrm{Hom}_{\mathbf{SPEC_2}}(F(\mathrm{A}), \mathrm{B})$.
On notera *AG* (Adjonction Gauche) l'application qui va dans le sens :

$$AG : \ \mathrm{Hom}_{\mathbf{SPEC_1}}(\mathrm{A}, U(\mathrm{B})) \ \rightarrow \ \mathrm{Hom}_{\mathbf{SPEC_2}}(F(\mathrm{A}), \mathrm{B})$$

et *AD* (Adjonction Droite) celle qui va dans l'autre sens. Rappelons encore que *bijection naturelle* signifie 2 conditions :

□ naturelle en A :
pour tout $\mathbf{SPEC_1}$-morphisme $\mu : \mathrm{A} \rightarrow U(\mathrm{B})$, et pour tout $\mathbf{SPEC_1}$-morphisme $\alpha: \mathrm{A} \rightarrow \mathrm{A}'$, on a :

$$AG(\mu \, o \, \alpha) \ = \ AG(\mu) \, o \, F(\alpha)$$

□ naturelle en B :
pour tout $\mathbf{SPEC_2}$-morphisme $\nu : F(\mathrm{A}) \rightarrow \mathrm{B}$, et pour tout $\mathbf{SPEC_2}$-morphisme $\beta: \mathrm{B} \rightarrow \mathrm{B}'$, on a :

$$AD(\beta \, o \, \nu) \ = \ U(\beta) \, o \, AD(\nu)$$

(Pour plus de précisions, se reporter à [McL 71]).

Nous allons construire cette bijection en trois étapes :

$$
\begin{array}{lcll}
\{\ \mathrm{A} & \rightarrow & U(\mathrm{B})\ \} & \\
 & \Uparrow & & \\
 & \textit{étape 1} & & \\
 & \downarrow & & \\
\{\ T_{\mathbf{\Sigma_1}(A)} = U(T_{\mathbf{\Sigma_2}(A)}) & \rightarrow & U(\mathrm{B})\ \} & \textit{(compatible avec R)} \\
 & \Uparrow & & \\
 & \textit{étape 2} & & \\
 & \downarrow & & \\
\{\ T_{\mathbf{\Sigma_2}(A)} & \rightarrow & \mathrm{B}\ \} & \textit{(compatible avec R)} \\
 & \Uparrow & & \\
 & \textit{étape 3} & & \\
 & \downarrow & & \\
\{\ F(\mathrm{A}) = (T_{\mathbf{\Sigma_2}(A)} / \equiv_{R,\mathbf{SPEC_2}}) & \rightarrow & \mathrm{B}\ \} &
\end{array}
$$

□ $\mathrm{Hom}_{\mathbf{SPEC_1}}(\mathrm{A}, U(\mathrm{B}))$ est en bijection avec l'ensemble des $\mathbf{\Sigma}$-exc_1-morphismes de $T_{\mathbf{\Sigma_1}(A)}$ dans U(B) compatibles avec la relation R sur $T_{\mathbf{\Sigma_1}(A)}$ (il s'agit de la relation R de la définition 20). Ceci résulte du fait que $T_{\mathbf{\Sigma_1}(A)}$ est la $\mathbf{\Sigma_1}$-algèbre libre au dessus de A, et que les sous-ensembles étiquetés de $T_{\mathbf{\Sigma_1}(A)}$ sont minimaux.
Cette bijection est naturelle en A : il est bien connu que les constructions libres sont naturelles, or nous nous sommes contentés de passer de A à $T_{\mathbf{\Sigma_1}(A)}$.
De plus elle est naturelle en B de manière immédiate, puisque l'on n'a pas modifié la composante de B dans cette étape.

□ L'ensemble des $\mathbf{\Sigma}$-exc_1-morphismes de $T_{\mathbf{\Sigma_1}(A)}$ dans U(B) compatibles avec la relation R est en bijection avec l'ensemble des $\mathbf{\Sigma}$-exc_2-morphismes de $T_{\mathbf{\Sigma_2}(A)}$ dans B compatibles avec la relation R.
Ceci résulte du fait que, puisque $T_{\mathbf{\Sigma_2}(A)}$ est la plus petite $\mathbf{\Sigma}$-exc_2-algèbre contenant $T_{\mathbf{\Sigma_1}(A)}$, on peut déduire un unique morphisme de $T_{\mathbf{\Sigma_2}(A)}$ dans B à partir de chaque $\mathbf{\Sigma}$-exc_1-morphisme de $T_{\mathbf{\Sigma_1}(A)}$ dans U(B). L'application inverse est simplement obtenue en faisant un oubli, car $U(T_{\mathbf{\Sigma_2}(A)}) = T_{\mathbf{\Sigma_1}(A)}$.
Le fait que cette bijection soit naturelle en A est clair car le passage de $T_{\mathbf{\Sigma_1}(A)}$ à $T_{\mathbf{\Sigma_2}(A)}$ est une inclusion ; elle est naturelle en B pour la même raison (le passage de U(B) à B est une inclusion).

□ Enfin, l'ensemble des $\mathbf{\Sigma}$-exc_2-morphismes de $T_{\mathbf{\Sigma_2}(A)}$ dans B compatibles avec le relation R est en bijection avec l'ensemble des $\mathbf{\Sigma}$-exc_2-morphismes de F(A) dans B simplement parce que F(A) est par définition le quotient de $T_{\mathbf{\Sigma_2}(A)}$ par la plus petite $\mathbf{SPEC_2}$-congruence compatible avec R, et que B est une $\mathbf{SPEC_2}$-algèbre.
Cette bijection est naturelle en B, puisque l'on n'a pas modifié la composante B dans cette dernière étape.
Elle est aussi naturelle en A parce que le diagramme suivant est toujours commutatif :

$$\begin{array}{ccc} F(\mathrm{A}) & -AG(\mu) \rightarrow & F(\mathrm{A'}) \\ \Uparrow & & \Uparrow \\ /\equiv_{R,\mathbf{SPEC_2}} & & /\equiv_{R,\mathbf{SPEC_2}} \\ | & & | \\ T_{\mathbf{\Sigma_2}(A)} & -\bar{\mu} \rightarrow & T_{\mathbf{\Sigma_2}(A')} \end{array}$$

La commutativité de ce diagramme résulte de la minimalité de la congruence de gauche et de la compatibilité de celle de droite avec $\mathbf{SPEC_2}$.

Ceci clôt la démonstration de notre théorème. □

Nota :
Une démonstration beaucoup plus courte de ce théorème se trouve dans [BBC 85]. Elle utilise un lemme puissant de la théorie des catégories : le lemme de Yoneda ([McL 71], III.2 ou IV.1). Ici, la place ne nous étant pas (trop) comptée, mieux valait en faire une démonstration “à la main”, car elle nous permet de comprendre dans le détail comment est bâtie l’adjonction entre U et F : il s’agit essentiellement d’une construction libre (de A à $T_{\mathbf{\Sigma_2}(A)}$) suivie d’un quotient par une congruence minimale compatible à la fois avec A et avec la spécification $\mathbf{SPEC_2}$. Le lemme de Yoneda, par sa puissance, laisse ce fait dans l’ombre ; et cache du même coup les motivations intuitives qui ont conduit à ce résultat. De plus, il utilise certains concepts spécifiques de la théorie des catégories que nous préférons écarter dans la mesure où nous ne les utilisons pas ailleurs.

Remarque 7 :
Le théorème précédent nous assure que pour toute $\mathbf{SPEC_1}$-algèbre A, et pour toute $\mathbf{SPEC_2}$-algèbre B, $\mathrm{Hom}_{\mathbf{SPEC_1}}(\mathrm{A}, U(\mathrm{B}))$ est en bijection naturelle avec $\mathrm{Hom}_{\mathbf{SPEC_2}}(F(\mathrm{A}), \mathrm{B})$.
En fait, pour les types abstraits algébriques avec une vision initiale, un seul cas très restreint de ce résultat nous intéresse (voir [Ber 86]) : c’est la cas où A est l’algèbre initiale $T_{\mathbf{SPEC_1}}$, et où B=$F(T_{\mathbf{SPEC_1}})$. Dans ce cas, le théorème précédent nous assure que $\mathrm{Hom}_{\mathbf{SPEC_1}}(T_{\mathbf{SPEC_1}}, U(F(T_{\mathbf{SPEC_1}})))$ est en bijection naturelle avec $\mathrm{Hom}_{\mathbf{SPEC_2}}(F(T_{\mathbf{SPEC_1}}), F(T_{\mathbf{SPEC_1}}))$. D’autre part, le fait que F soit adjoint à gauche au foncteur d’oubli, nous assure que $F(T_{\mathbf{SPEC_1}})$ est égal à l’algèbre initiale de Alg($\mathbf{SPEC_2}$), $T_{\mathbf{SPEC_2}}$. Si bien que l’on obtient : $\mathrm{Hom}_{\mathbf{SPEC_1}}(T_{\mathbf{SPEC_1}}, U(T_{\mathbf{SPEC_2}}))$ est en bijection naturelle avec $\mathrm{Hom}_{\mathbf{SPEC_2}}(T_{\mathbf{SPEC_2}}, T_{\mathbf{SPEC_2}})$.
Là encore, seulement un résultat restreint de ce fait nous intéresse : $\mathrm{Hom}_{\mathbf{SPEC_2}}(T_{\mathbf{SPEC_2}}, T_{\mathbf{SPEC_2}})$ contient en particulier le morphisme identité sur $T_{\mathbf{SPEC_2}}$, qui est donc associé de manière unique avec un morphisme de $T_{\mathbf{SPEC_1}}$ dans $U(T_{\mathbf{SPEC_2}})$. Ce morphisme s’appelle le *morphisme d’adjonction* associé à $T_{\mathbf{SPEC_1}}$. On le note habituellement $I_{T_{\mathbf{SPEC_1}}}$:

$$I_{T_{\mathbf{SPEC_1}}} : \ T_{\mathbf{SPEC_1}} \ \rightarrow \ U(T_{\mathbf{SPEC_2}}) \, .$$

Voici intuitivement pourquoi ce morphisme est important :
lorsque le foncteur de synthèse F agit sur l’algèbre $T_{\mathbf{SPEC_1}}$, il effectue intuitivement la construction minimale au-dessus de $T_{\mathbf{SPEC_1}}$ qui permette d’enrichir $T_{\mathbf{SPEC_1}}$ en accord avec la présentation **PRES**. Il se trouve tout naturellement que cet enrichissement est égal à l’algèbre initiale de la spécification $\mathbf{SPEC_2} = \mathbf{SPEC_1} + \mathbf{PRES}$. Lorsque l’on veut “retrouver” la structure prédéfinie après que la présentation **PRES** ait agi, on ne peut le faire que via le foncteur d’oubli U. On obtient donc la $\mathbf{SPEC_1}$-algèbre $U(T_{\mathbf{SPEC_2}})$, et non la $\mathbf{SPEC_1}$-algèbre $T_{\mathbf{SPEC_1}}$. Cette algèbre $U(T_{\mathbf{SPEC_2}})$ n’est pas nécessairement isomorphe à $T_{\mathbf{SPEC_1}}$: il se peut que la présentation **PRES** ait “abimé” $T_{\mathbf{SPEC_1}}$. C’est là que le morphisme d’adjonction de $T_{\mathbf{SPEC_1}}$ dans $U(T_{\mathbf{SPEC_2}})$ prend toute sa valeur : c’est lui qui permet de mesurer les modifications subies par $T_{\mathbf{SPEC_1}}$ au cours de cette manipulation.
On comprends ainsi l’importance d’une sémantique possédant une foncteur de synthèse adjoint à gauche au foncteur d’oubli dans tout formalisme des types abstraits algébriques. C’est l’adjonction qui permet de construire ce morphisme $I_{T_{\mathbf{SPEC_1}}}$; et par voie de conséquence, c’est elle qui permet de

donner une sémantique *hiérarchique* utilisable.
Nous ferons donc largement usage de ce morphisme d'adjonction $I_{T_{\mathbf{SPEC}_1}}$ dans le chapitre suivant.

Chapitre VII :

Consistance hiérarchique,

Suffisante complétude

Lorsque l'on spécifie une présentation au-dessus d'une spécification prédéfinie, on est souvent amené à vérifier que cette présentation "n'abime pas" la structure prédéfinie. Cette vérification peut se modéliser au moyen de deux concepts : la *consistance hiérarchique* et le *suffisante complétude*. Nous avons déjà remarqué combien ces notions sont utiles pour prouver, par exemple, qu'une implémentation abstraite est correcte. Dans ce chapitre, nous montrons comment on peut redéfinir la consistance hiérarchique et la suffisante complétude dans le cadre des exception-algèbres. Nous ferons largement usage de ces notions dans la partie suivante sur l'implémentation abstraite en présence d'exceptions.

1. LA CONSISTANCE HIERARCHIQUE

Rappelons que la définition classique de la consistance hiérarchique signifie que deux valeurs prédéfinies ne doivent pas être amalgamées par la présentation **PRES**. Ceci se traduit par le fait que le morphisme d'adjonction, de $T_{\mathbf{SPEC_1}}$ dans $U(T_{\mathbf{SPEC_2}})$, doit être injectif.

Avec traitement d'exceptions, on veut bien sûr aussi modéliser le fait que deux valeurs prédéfinies distinctes ne doivent pas être amalgamées. Par conséquent, pour qu'une présentation soit hiérarchiquement consistante il faudra que le morphisme d'adjonction $I_{T_{\mathbf{SPEC_1}}}$ soit injectif. Mais cela ne suffit pas.
Il ne faut pas oublier qu'une exception-algèbre est une algèbre classique *munie de sous-ensembles étiquetés*. Il nous faut donc aussi exprimer une notion de consistance vis à vis des étiquetages d'exception. Intuitivement, la notion de consistance relativement aux étiquettes d'exception est claire : il ne faut pas que la présentation **PRES** étiquette une valeur predéfinie avec une étiquette prédéfinie (de $\mathbf{L_1} \cup \{Ok\}$), alors que cette valeur n'était pas étiquetée par elle avant (dans $T_{\mathbf{SPEC_1}}$).
Rappel : notre syntaxe ne contient aucun axiomes d'égalité entre étiquettes, par conséquent on ne risque pas d'amalgamer deux étiquettes prédéfinies distinctes. La seule inconsistance possible est donc bien l'ajout d'étiquetages de valeurs prédéfinies.

Dire que l'on n'a ajouté aucun étiquetage prédéfini à une valeur prédéfinie s'exprime clairement comme suit :

- □ Pour toute valeur $x \in T_{\mathbf{SPEC_1}}$, $I_{T_{\mathbf{SPEC_1}}}(x)$ est étiquetée par l dans $U(T_{\mathbf{SPEC_2}})$ seulement si x est déjà étiquetée par l dans $T_{\mathbf{SPEC_1}}$.

Notre définition de la consistance hiérarchique est donc exprimée en deux points :

- □ L'unité d'adjonction $I_{T_{\mathbf{SPEC_1}}}$ est injective
- □ Pour toute étiquette $l \in \mathbf{L_1} \cup \{Ok\}$, $I_{T_{\mathbf{SPEC_1}}}(T_{\mathbf{SPEC_1},l}) = U(T_{\mathbf{SPEC_2}})_l$
(l'inclusion $I_{T_{\mathbf{SPEC_1}}}(T_{\mathbf{SPEC_1},l}) \subset U(T_{\mathbf{SPEC_2}})_l$ est toujours vérifiée, car $I_{T_{\mathbf{SPEC_1}}}$ est un exception-morphisme).

Il existe un moyen très court d'exprimer ceci dans le cadre de la théorie des catégories, c'est la notion de *morphisme partiellement rétractable* :

Définition 21 :
Soit **C** une catégorie telle que l'image de tout objet par tout morphisme soit un objet (par exemple, dans Alg(**SPEC**), l'image d'un morphisme $\mu: A \rightarrow A'$, est une sous-algèbre, $\mu(A)$, de A').
Un morphisme

$$\mu : \quad A \rightarrow A' \text{ de } \mathbf{C}$$

est dit *partiellement rétractable* si et seulement si le morphisme corestreint

$$\mu : \quad A \rightarrow \mu(A')$$

possède un inverse à gauche :

$$\mu^{-1} : \quad \mu(A') \rightarrow A \quad , \quad \mu^{-1} \, o \, \mu = Id_A \, .$$

La proposition suivante nous montre en quoi les morphismes partiellement rétractables expriment parfaitement notre notion de consistance hiérarchique.

Proposition 2 :
Un exception-morphisme $\mu: A -> A'$ est partiellement rétractable si et seulement si il est *injectif* et pour toute étiquette $l \in \mathbf{L} \cup \{Ok\}$ on a :

$$\mu(A_l) \quad = \quad A'_l \cap \mu(A) \, .$$

(remarquons que l'inclusion $\mu(A_l) \subset A'_l \cap \mu(A)$ est toujours vérifiée, par définition des exception-morphismes).

Preuve :
Il est bien connu que si μ est un **Σ**-morphisme injectif classique, alors il existe un unique **Σ**-morphisme inverse ν de $\mu(A)$ dans A .
Par conséquent, nous devons uniquement prouver que ce morphisme ν est en fait une exception-morphisme. En d'autres termes, il nous faut prouver qu'il vérifie :

$$\nu(\mu(A)_l) \quad \subset \quad A_l \quad \text{pour tout } l \in \mathbf{L} \cup \{Ok\}$$

c'est-à-dire :

$$\nu(\, A'_l \cap \mu(A)\,) \quad \subset \quad A_l \quad \text{pour tout } l \in \mathbf{L} \cup \{Ok\}$$

Mais ceci est exactement équivallent à :

$$A'_l \cap \mu(A) \quad \subset \quad \mu(A_l) \, .$$

Ce qui prouve notre proposition. □

Ainsi, on peut donner la définition suivante de la consistance hiérarchique pour les exception-présentations :

Définition 22 :
Une présentation **PRES** au-dessus de l'exception-spécification $\mathbf{SPEC_1}$ est dite *hiérarchiquement consistante* si et seulement si le morphisme d'adjonction $I_{T_{\mathbf{SPEC_1}}}$ est partiellement rétractable dans

Alg($\mathbf{SPEC_1}$).

Exemple 16 :
La présentation des tableaux *ARRAY* au-dessus des entiers naturels bornés et des booléens (*NAT+BOOL*) (exemple 14) est hiérarchiquement consistante. En effet :

- □ Le morphisme d'adjonction $I_{\{crash\}\cup\mathbf{N}\cup\{True,False\}}$ est injectif car la présentation *ARRAY* n'amalgame jamais deux entiers naturels ou booléens distincts.
- □ La présentation *ARRAY* n'ajoute aucun étiquetage avec des étiquettes prédéfinies car **Lbl-Ax** ne concerne que les étiquettes *NEGATIVE-RANGE, RANGE-TOO-LARGE* et *NOT-INITIAL-IZED*, et non pas celles de *NAT* ; et de plus, en ce qui concerne l'étiquette *Ok*, la présentation *ARRAY* n'ajoute aucune forme *Ok* de sorte *NAT* ou *BOOL*, donc elle n'ajoute aucune valeur *Ok* prédéfinie.

Par contre, si l'on avait ajouté un axiome d'étiquetage tel que :

$$n > Maxrange = True \quad \Rightarrow \quad n \in TOO\text{-}LARGE\text{-}NUMBER$$

alors la présentation *ARRAY* n'aurait plus été hiérarchiquement consistante, car tous les entiers de **N** compris entre *Maxrange* et *Maxint* deviendraient étiquetés par *TOO-LARGE-NUMBER* alors qu'ils ne l'étaient pas avant (rappelons que l'on suppose que *Maxrange* est plus petit que *Maxint* ; en fait, dans le cas où *Maxrange*≥*Maxint*, la présentation reste consistante).

Il nous reste maintenant à traiter la suffisante complétude.

2. LA SUFFISANTE COMPLETUDE

Dans le cas classique (sans traitement d'exceptions), la suffisante complétude signifie que la présentation **PRES** n'ajoute aucune nouvelle valeur dans les sortes prédéfinies. Elle s'exprime donc par la surjectivité du morphisme d'adjonction $I_{T_{\mathbf{SPEC_1}}}$ de $T_{\mathbf{SPEC_1}}$ dans $U(T_{\mathbf{SPEC_2}})$.
Dans le cas avec traitement d'exceptions, cette condition est un peu trop forte. En effet, si l'on considère l'exemple de la présentation *ARRAY* au-dessus de *NAT+BOOL*, on s'aperçoit que de nouvelles valeurs ont été ajoutées dans les sortes prédéfinies : par exemple les valeurs de sorte *NAT* obtenues par un accès à une place non-initialisée d'un tableau ne sont pas des valeurs de {crash}∪**N** ; ce sont de nouvelles valeurs (erronées).
Il est pourtant clairement nécessaire de pouvoir faire de tels ajouts de valeurs erronées dans les sortes prédéfinies. En effet, on ne peut pas imposer, lors de la spécification de $\mathbf{SPEC_1}$, de prévoir *toutes* les valeurs erronées qui peuvent être ajoutées lors d'enrichissements ultérieurs quelconques.
Par conséquent, il faut définir la suffisante complétude avec traitement d'exceptions de telle manière que les ajouts de valeurs erronées soient autorisés. Là où la suffisante complétude doit être remise en cause, c'est seulement dans le cas où l'on ajoute des valeurs qui ne sont pas erronées.
Rappelons que les valeurs erronées sont définies à partir des étiquettes d'exception : une valeur étiquetée crée une erreur, sauf si elle est *Ok* (i.e. *récupérée*) ; et les erreurs se propagent implicitement, sauf en cas de récupération. Ainsi, on constate que la non-suffisante complétude résultera d'un manque de déclaration d'étiquettes d'exception. Ceci semble raisonnable car les étiquettes d'exception modélisent des "diagnostics d'erreur", par conséquent, une présentation ne sera pas suffisamment complète si l'on ajoute des valeurs dans les sortes prédéfinies pour lesquelles on n'a prévu aucun diagnostic d'erreur.

La suffisante complétude doit donc être définie comme suit :

- □ Une présentation sera suffisamment complète si et seulement si l'image du morphisme d'adjonction contient toutes les valeurs non-erronées de $U(T_{\mathbf{SPEC_2}})$. Ce qui signifie bien que la présentation **PRES** ne peut ajouter que des valeurs erronées dans les sortes prédéfinies.

C'est exactement ce que traduit la définition suivante :

Définition 23 :
Une présentation **PRES** au-dessus de l'exception-spécification $\mathbf{SPEC_1}$ est dite *suffisamment complète* si et seulement si le morphisme d'adjonction $I_{T_{\mathbf{SPEC_1}}}$ satisfait la condition suivante :

$$U[\, T_{\mathbf{SPEC_2}} - T_{\mathbf{SPEC_2},err}\,] \;\subset\; I_{T_{\mathbf{SPEC_1}}}(T_{\mathbf{SPEC_1}})\,.$$

(où $\mathbf{SPEC_2} = \mathbf{SPEC_1} + \mathbf{PRES}$)

Nota :
Ceci signifie exactement que la partie de $T_{\mathbf{SPEC_2}}$ relative aux sortes prédéfinies ne peut contenir que des valeurs *erronées* en plus des valeurs que contenait déjà $T_{\mathbf{SPEC_1}}$.
Lorsque l'on écrit "$U[\, T_{\mathbf{SPEC_2}} - T_{\mathbf{SPEC_2},err}\,]$" , on fait un abus de langage car $T_{\mathbf{SPEC_2}} - T_{\mathbf{SPEC_2},err}$ n'est pas une exception-algèbre (c'est seulement un sous-ensemble de $T_{\mathbf{SPEC_2}}$), or nous avons défini U comme un foncteur (il doit donc s'appliquer à des exception-algèbres en toute rigueur). Comme on s'en doute bien, "$U[\, T_{\mathbf{SPEC_2}} - T_{\mathbf{SPEC_2},err}\,]$" est en fait un abus de notation pour : "*la partie de* $[T_{\mathbf{SPEC_2}} - T_{\mathbf{SPEC_2},err}]$ *relative aux sortes de* $\mathbf{SPEC_1}$".

Exemple 17 :
La présentation des tableaux *ARRAY* au-dessus des entiers naturels bornés et des booléens (*NAT+BOOL*) (exemple 14) est suffisamment complète. En effet, les seules valeurs de sorte *NAT* ajoutées par *ARRAY* sont issues d'un accès à un tableau dans l'un des cas suivants (cf. exemple 15) :

- □ un accès à un rang négatif, auquel cas cette valeur répond à l'étiquette d'exception *NEGATIVE-RANGE* (et de toute façon, cette valeur est déjà erronée par le seul fait qu'elle résulte d'un surterme d'une valeur négative erronée)

- □ un accès à un rang supérieur à *Maxrange*, auquel cas cette valeur répond à l'étiquette d'exception *RANGE-TOO-LARGE*, et est donc erronée

- □ un accès à un rang qui n'a pas été initialise, auquel cas cette valeur répond à l'étiquette d'exception *NOT-INITIALIZED*, et est donc erronée

- □ enfin toutes les propagations issues des valeurs précédentes, qui sont donc des valeurs erronées par propagation implicite des erreurs.

On peut ajouter à cette énumération toutes les valeurs booléennes issues des opérations "*eq?*" et "*<*" appliquées aux valeurs erronées entières précédentes. Ces valeurs sont elles aussi erronées par propagation implicite des erreurs.
Donc la présentation *ARRAY* donnée en exemple 14 est suffisamment complète.
Il n'en aurait pas été de même si l'on avait omis les axiomes d'étiquetage relatifs aux étiquettes *RANGE-TOO-LARGE* ou *NOT-INITIALIZED*. En effet, les valeurs issues de tels accès ne seraient alors ni *Ok* (car les axiomes ne leurs donnent aucune forme *Ok*) ni erronées (car elles ne rencontrent aucun étiquetage d'exception dans leur histoire). Dans ce cas, la présentation n'aurait pas été suffisamment complète.
On remarque que rien n'est imposé à propos des étiquettes d'exception associées aux valeurs erronées de type tableaux. On y reviendra.

Nous sommes maintenant aptes à définir ce qu'est une présentation "correcte". On les nomme usuellement les présentations *persistantes* :

Définition 24 :
Une présentation **PRES** au-dessus de l'exception-spécification $\mathbf{SPEC_1}$ est dite *persistante* si et seulement si elle est à la fois hiérarchiquement consistante et suffisamment complète.

3. SUFFISANTE COMPLETUDE 'GLOBALE'

On remarque que lors de la preuve de la suffisante complétude de *ARRAY* au-dessus de *NAT+BOOL*, rien n'impose de fournir une étiquette d'exception aux tableaux qui ne sont pas *Ok*. Par exemple, on pourrait n'associer aucune étiquette d'exception aux tableaux ayant subit une affectation hors des rangs [*0..Maxrange*] ; ceci ne nuirait pas nécessairement à la suffisante complétude de la présentation *ARRAY* (dans le cas ou l'on récupère les entiers issus d'accès "valides" dans ces tableaux).
On constate donc que le fait d'associer "suffisamment d'étiquettes d'exception" à des valeurs exceptionnelles de sorte non-prédéfinie n'est pas du ressort de la suffisante complétude telle que nous l'avons définie (c'est bien naturel, car ce qui nous intéressait était de ne pas "abimer" les sortes prédéfinies ; les sortes de **PRES** n'ont aucun rôle à jouer à ce niveau).

La même remarque s'applique à des cas aussi simples que la spécification des entiers bornés (sans aucune notion de présentation cette fois). La question qui se pose est la suivante :
Lorsque l'on écrit une exception-spécification, a-t'on spécifié suffisamment d'étiquettes d'exception ?
Par exemple, ayant déclaré (au moyen de **Ok-Frm**) que les formes *Ok* correspondent à [*0..Maxint*], rien ne nous obligeait à spécifier que les valeurs plus grandes que *Maxint* répondent à l'étiquette *TOO-LARGE-NUMBER*. Dès lors, des valeurs telles que *succ(Maxint)* ne seraient ni *Ok* (puisqu'associées à aucune forme *Ok*) ni erronée (car elles ne rencontrent aucune étiquette d'exception dans leur histoire). Ces valeurs sont *incomplètement spécifiées*.

On remarque donc que l'on peut définir une notion de suffisante complétude hors de toute préoccupation hiérarchique. On peut définir une notion de suffisante complétude de manière interne à une exception-spécification.
Ce que l'on veut modéliser, c'est le fait que l'algèbre initiale $T_{\mathbf{SPEC}}$ associée à une exception-spécification **SPEC** ne contienne aucune valeur incomplètement spécifiée ; c'est-à-dire aucune valeur qui ne soit ni *Ok* ni erronée. Nous appellerons ceci la suffisante complétute "globale" :

Définition 25 :
Une exception-spécification **SPEC** sera dite *globalement suffisamment complète* si et seulement si son exception-algèbre initiale vérifie :

$$T_{\mathbf{SPEC},Ok} \cup T_{\mathbf{SPEC},err} = T_{\mathbf{SPEC}} .$$

Ce qui traduit exactement le fait que toute valeur de $T_{\mathbf{SPEC}}$ est soit *Ok* soit erronée.
Remarquons que $T_{\mathbf{SPEC},Ok}$ et $T_{\mathbf{SPEC},err}$ sont toujours disjointes, par définition de $T_{\mathbf{SPEC},err}$.
Lorsqu'une exception-spécification est globalement suffisamment complète, on retrouve la partition en valeurs *Ok* et erronées des précédents formalismes (décrits dans le chapitre I, section 3).

Bien que ce point ne soit pas encore étudié (par manque de temps), il doit être possible de formaliser une notion de "spécification gracieuse" pour laquelle la suffisante complétude globale soit assurée. Par exemple, en ce qui concerne la spécification de l'opération *pred*, on est assuré qu'elle est complètement spécifiée lorsque l'on écrit simplement :

$$pred(succ(n)) = n \quad \text{(dans } \mathbf{Ok\text{-}Ax}\text{)}$$

puis :

$$pred(0) \in NEGATIVE\text{-}VALUE \quad \text{(dans } \mathbf{Lbl\text{-}Ax}\text{)}$$

parce que tous les cas d'application de l'opération *pred* qui peuvent créer une nouvelle erreur (ici appliquée à 0) répondent à un message d'erreur (ici *NEGATIVE-VALUE*). Or on constate que l'on a en fait suivi une méthode de dérivation par rapport aux constructeurs de *NAT* (*0* et *succ*) ; ce qui relève des techniques développées pour les présentations gracieuses [Bid 82].

Le résultat suivant prouve que pour construire une exception-spécification globalement suffisamment complète, il suffit d'écrire d'abord une spécification globalement suffisamment complète ne contenant que les constructeurs du type, puis de l'enrichir avec les autres opérations au moyen de présentations persistantes.

Proposition 3 :
Soit **PRES** une présentation au-dessus de l'exception-spécification $\mathbf{SPEC_1}$, et soit $\mathbf{SPEC_2} = \mathbf{SPEC_1} + \mathbf{PRES}$.
Si :

- □ $\mathbf{SPEC_1}$ est globalement suffisamment complète
- □ l'ensemble des sortes de **PRES** est vide (i.e. **PRES** se contente d'enrichir l'ensemble des opérations agissant sur les sortes prédéfinies)
- □ **PRES** est persistante.

alors $\mathbf{SPEC_2}$ est globalement suffisamment complète.

Preuve :
Du fait que **PRES** ne contient aucune nouvelle sorte, l'oubli *U* est simplement l'identité sur les supports de $T_{\mathbf{SPEC_2}}$. Par conséquent, la consistance hiérarchique de **PRES** s'écrit :

$$I_{T_{\mathbf{SPEC_1}}} : T_{\mathbf{SPEC_1}} \rightarrow T_{\mathbf{SPEC_2}} \quad \text{est partiellement rétractable}$$

Puisque $I_{T_{\mathbf{SPEC_1}}}$ n'est qu'une inclusion, cela implique que pour toute étiquette *l* de $\mathbf{L_1} \cup \{Ok\}$, on a :

$$(1) \qquad T_{\mathbf{SPEC_2},l} \cap T_{\mathbf{SPEC_1}} = T_{\mathbf{SPEC_1},l}$$

La suffisante complétude de **PRES** s'ecrit alors :

$$(2) \qquad T_{\mathbf{SPEC_2}} - T_{\mathbf{SPEC_2},err} \subset T_{\mathbf{SPEC_1}}$$

Et le fait que $\mathbf{SPEC_1}$ soit globalement suffisamment complète s'écrit :

$$(3) \qquad T_{\mathbf{SPEC_1},Ok} \cup T_{\mathbf{SPEC_1},err} = T_{\mathbf{SPEC_1}}$$

Par construction des valeurs erronées d'une exception-algèbre, aucune valeur n'est à la fois *Ok* et erronée. Par conséquent, on déduit de (2) :

$$(4) \qquad T_{\mathbf{SPEC_2},Ok} \subset T_{\mathbf{SPEC_1}}$$

On déduit alors de (4) et de (1) appliquée à $l=Ok$:

$$(5) \qquad T_{\mathbf{SPEC_2},Ok} = T_{\mathbf{SPEC_1},Ok}$$

On remarque alors que, par construction de l'ensemble des valeurs erronées d'une exception-algèbre (A_{err}), il est croissant avec les A_{l_i} où l_i parcourt **L** et décroissant avec A_{Ok}. En particulier, on déduit de (5) et de (1) (avec $A=T_{\mathbf{SPEC_1}}$) :

$$(6) \qquad T_{\mathbf{SPEC_1},err} \subset T_{\mathbf{SPEC_2},err}$$

De (6) et de (2) il vient :

$$(7) \qquad T_{\mathbf{SPEC_2}} - T_{\mathbf{SPEC_2},err} \subset T_{\mathbf{SPEC_1}} - T_{\mathbf{SPEC_1},err}$$

(3) implique alors

$$(8) \qquad T_{\mathbf{SPEC_2}} - T_{\mathbf{SPEC_2},err} \subset T_{\mathbf{SPEC_1},Ok}$$

et par (5) il résulte que :

$$(9) \qquad T_{\mathbf{SPEC_2}} - T_{\mathbf{SPEC_2},err} \subset T_{\mathbf{SPEC_2},Ok}$$

Ce qui s'écrit encore

$$T_{\mathbf{SPEC_2}} \subset T_{\mathbf{SPEC_2},Ok} \cup T_{\mathbf{SPEC_2},err}$$

Ce qui signifie que $\mathbf{SPEC_2}$ est globalement suffisamment complète. □

Chapitre VIII :

Les algèbres

post-initiales

Nous allons voir dans ce chapitre que même en suivant une approche initiale (comme nous le faisons dans cette partie B), l'algèbre initiale associée à une exception-spécification **SPEC** ($T_{\mathbf{SPEC}}$) ne convient pas toujours pour caractériser la sémantique de **SPEC**. Une sous-classe légèrement élargie de Alg(**SPEC**) constitue souvent un choix plus judicieux : nous la nommerons la sous-classe des exception-algèbres *post-initiales*.

1. MOTIVATION

Considérons une exception-spécification **SPEC**. Nous avons remarqué durant le chapitre précédent qu'étant donnée une présentation *persistante* quelconque **PRES** au-dessus de **SPEC**, l'action de **PRES** sur l'algèbre initiale $T_{\mathbf{SPEC}}$ n'est pas nulle. En effet, nous savons que **PRES** peut ajouter de nouvelles valeurs erronées dans les sortes de **SPEC** : l'oubli de la (**SPEC**+**PRES**)-algèbre initiale, $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$, contient alors $T_{\mathbf{SPEC}}$ mais ne lui est pas isomorphe.

Il en résulte le fait suivant : si l'on définit successivement deux présentations persistantes $\mathbf{PRES_1}$ et $\mathbf{PRES_2}$ au-dessus de **SPEC**, rien ne prouve que $\mathbf{PRES_2}$ (par exemple) soit encore persistante au-dessus de $U_{\mathbf{SPEC}}(T_{\mathbf{PRES_1}})$. Ceci résulte clairement du fait que $U_{\mathbf{SPEC}}(T_{\mathbf{PRES_1}})$ n'est pas isomorphe à $T_{\mathbf{SPEC}}$, et que la persistance de $\mathbf{PRES_2}$ n'est définie que par rapport à $T_{\mathbf{SPEC}}$.

Pourtant, il arrive souvent que l'on doive considérer successivement deux présentations persistantes au-dessus de la même spécification prédéfinie. C'est par exemple le cas pour l'implémentation (parties A et C de cette thèse) : on considère d'une part la présentation "résidante" $\mathbf{PRES_0}$ et d'autre part la présentation à implémenter $\mathbf{PRES_1}$.
Rappelons que la correction d'une implémentation repose essentiellement sur la *consistance hiérarchique* et la *suffisante complétude*. De telles notions uniquement définies par rapport à l'algèbre initiale ne suffisent donc pas dans le cas d'une implémentation avec traitement d'exceptions, puisque contrairement au cas sans traitement d'exceptions, $U_{\mathbf{SPEC}}(T_{\mathbf{PRES_0}})$ n'est pas isomorphe à l'algèbre initiale $T_{\mathbf{SPEC}}$.

On conçoit donc aisément que l'ensemble des algèbres de la forme $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$ (où **PRES** est une présentation persistante quelconque) est d'un intérêt majeur. Nous allons maintenant nous attacher à caractériser ces exception-algèbres particulières.

2. CARACTERISATION

Soit **SPEC** une exception-spécification. Nous allons considérer toutes les présentations persistantes **PRES** au-dessus de **SPEC**. Notre but est d'énumérer les propriétés des exception-algèbres $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$ que l'on peut déduire du fait que les présentations **PRES** sont persistantes, c'est-à-dire *suffisamment complètes* et *hiérarchiquement consistantes*, au-dessus de **SPEC**.

La consistance hiérarchique de **PRES** au-dessus de **SPEC** signifie que le morphisme d'adjonction

$$T_{\mathbf{SPEC}} \quad \rightarrow \quad U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$$

est *partiellement rétractable*.
La consistance hiérarchique signifie donc que $T_{\mathbf{SPEC}}$ est une sous-algèbre "pleine" de $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$ (rappelons en effet que dans une catégorie, les objets ne sont définis qu'à isomorphisme près). Ceci signifie que d'une part $T_{\mathbf{SPEC}}$ est incluse dans $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$, et d'autre part pour chaque étiquette $l \in \mathbf{L} \cup \{Ok\}$ $T_{\mathbf{SPEC},l}$ est égal à $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})_l \cap T_{\mathbf{SPEC}}$.

La suffisante complétude de **PRES** signifie alors que

$$U_{\mathbf{SPEC}}[\, T_{\mathbf{PRES}} - T_{\mathbf{PRES},err} \,] \quad \subset \quad T_{\mathbf{SPEC}} \, .$$

c'est-à-dire que toutes les valeurs non erronées de $T_{\mathbf{PRES}}$, de sorte prédéfinie, sont élément de $T_{\mathbf{SPEC}}$ (**PRES** n'ajoute aucune valeur non erronée).
Rappelons que nous étudions les algèbres de la forme $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$ pour des présentations persistantes *quelconques* **PRES** ; c'est-à-dire que le contenu des présentations **PRES** n'est pas connu a priori. En particulier, nous ne pouvons pas déterminer exactement quelles sont les valeurs erronées de $T_{\mathbf{PRES}}$ car l'application du foncteur d'oubli ne préserve pas $T_{\mathbf{PRES},err}$.
La seule chose que nous sachions (cf. chapitre IV, section 2) c'est que

$$T_{\mathbf{PRES},err} \quad \cap \quad T_{\mathbf{PRES},Ok} \quad = \quad \varnothing$$

d'où l'on peut alors déduire :

$$U_{\mathbf{SPEC}}(\, T_{\mathbf{PRES},Ok} \,) \quad \subset \quad T_{\mathbf{SPEC}}$$

c'est-à-dire (car le foncteur d'oubli préserve l'étiquette *Ok*) :

$$U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})_{Ok} \quad \subset \quad T_{\mathbf{SPEC}} \, .$$

Nous venons donc de démontrer que les algèbres "intéressantes" sont les algèbres telles que le morphisme d'adjonction

$$T_{\mathbf{SPEC}} \quad \rightarrow \quad U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$$

est partiellement rétractable, et telles que la partie *Ok* vérifie

$$U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})_{Ok} \quad \subset \quad T_{\mathbf{SPEC}} \, .$$

Nous nommerons cette classe d'algèbres les exception-algèbres *post-initiales*.

Définition 26 :
Etant donnée une exception-spécification **SPEC**, une exception-algèbre *post-initiale* est une **SPEC**-algèbre, A ∈ Alg(**SPEC**), telle que :

- □ le morphisme initial

  $$init : \quad T_{\mathbf{SPEC}} \quad \rightarrow \quad A$$

  est *partiellement rétractable*

□ la partie *Ok* vérifie :

$$A_{Ok} \subset init(T_{\mathbf{SPEC}})$$

Ceci signifie exactement que la partie finiment générée de A est isomorphe à l'algèbre initiale, et que A_{Ok} est incluse dans cette partie finiment générée.

Proposition 4 :
Si A est une **SPEC**-algèbre post-initiale alors :

$$A_{Ok} = init(T_{\mathbf{SPEC},Ok})$$

Preuve :
Par définition : $A_{Ok} \subset init(T_{\mathbf{SPEC}})$; et *init* étant un morphisme partiellement rétractable :

$$init(T_{\mathbf{SPEC},Ok}) = A_{Ok} \cap init(T_{\mathbf{SPEC}})$$

d'où la conclusion. □

3. LES PRESENTATIONS STABLES

3.1. LES INSUFFISANCES DE L'ADJONCTION

Etant donnée une exception-spécification **SPEC**, nous venons d'établir que, même en suivant une approche résolument initiale, l'algèbre initiale seule ne suffit pas à caractériser la sémantique associée à la spécification **SPEC**. Nous devons considérer la sous-catégorie de Alg(**SPEC**) formée des **SPEC**-algèbres post-initiales.

Lorsque l'on considère l'approche hiérarchique dans le cas sans traitement d'exceptions, la propriété d'*adjonction* entre les foncteurs d'oubli $U_{\mathbf{SPEC}}$ et de synthèse $F_{\mathbf{PRES}}$ (associés à une présentation **PRES** au-dessus de **SPEC**) implique en particulier que $F_{\mathbf{PRES}}(T_{\mathbf{SPEC}}) = T_{\mathbf{PRES}}$. Ceci signifie que le foncteur $F_{\mathbf{PRES}}$ *conserve les algèbres initiales*. Par contre, le foncteur d'oubli $U_{\mathbf{SPEC}}$ ne conserve pas les algèbres initiales en général : $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$ n'est pas isomorphe à $T_{\mathbf{SPEC}}$ en général. La propriété de *persistance* est exactement conçue dans ce but : $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$ est isomorphe à $T_{\mathbf{SPEC}}$ si et seulement si la présentation **PRES** est persistante.

Dans le cas avec traitement d'exceptions, si nous voulons maintenant une approche hiérarchique compatible avec toutes les algèbres post-initiales, les propriétés intéressantes requises pour une présentation persistante deviennent clairement les suivantes :

□ Pour toute **SPEC**-algèbre post-initiale A, $F_{\mathbf{PRES}}$(A) doit être une (**SPEC**+**PRES**)-algèbre post-initiale.

□ Pour toute (**SPEC**+**PRES**)-algèbre post-initiale B, $U_{\mathbf{SPEC}}$(B) doit être une **SPEC**-algèbre post-initiale.

De la même façon que dans la cas sans traitement d'exceptions, la seconde propriété sera assurée par une notion de persistance (la persistance *forte*). Par contre, la première propriété, qui était automatiquement satisfaite pour l'algèbre initiale par adjonction, n'est pas satisfaite pour toutes les algèbres post-initiales (des contre-exemples sont donnés dans les sections suivantes).

3.2. LA STABILITE

Une exception-présentation **PRES** sera dite *stable* si le foncteur de synthèse préserve les algèbres post-initiales. D'après la définition des exception-algèbres post-initiales, nous pouvons décomposer la stabilité en deux propriétés d'une manière similaire à la décomposition *complétude/consistance* pour la persistance (qui concerne quant à elle le foncteur d'oubli).

Définition 27 :
Une présentation **PRES** au-dessus d'une exception-spécification **SPEC** est dite *Ok-complète* si et seulement si pour toute **SPEC**-algèbre post-initiale A, la partie *Ok* de $F_{\mathbf{PRES}}$(A) est ($\mathbf{\Sigma_{SPEC}} + \mathbf{\Sigma_{PRES}}$)-finiment générée.

Voici un exemple de présentation persistante qui, pourtant, n'est pas *Ok*-complète.

Exemple 18 :
Considérons la spécification **SPEC**=*NAT* (non bornée, avec les opérations *0* et *succ*). Considérons la présentation "*TAS*" avec les opérations

$$\begin{array}{rrcl} \mathit{vide} : & & \rightarrow & \mathit{TAS} \\ \mathit{entasse} : & \mathit{NAT}\ \mathit{TAS} & \rightarrow & \mathit{TAS} \end{array}$$

avec la déclaration de formes *Ok* suivante :

$$\begin{array}{rcl} & & \mathit{vide} \in \text{Ok-Frm} \\ t \in \text{Ok-Frm} & \Rightarrow & \mathit{entasse}(x,t) \in \text{Ok-Frm} \end{array}$$

et aucun axiome.
Cette présentation est clairement persistante au-dessus de *NAT* (aucune opération de cible dans *NAT* et aucun axiome). Pourtant, elle n'est pas *Ok*-complète. Pour le constater, il suffit de considérer la *NAT*-algèbre **Z**, munie de $\mathbf{Z}_{Ok}$=**N** . Elle est non-finiment générée (les valeurs négatives ne sont pas atteintes par les opérations de **SPEC**), mais elle est post-initiale car sa partie finiment générée est égale à **N** et contient $\mathbf{Z}_{Ok}$. D'après les déclarations de formes *Ok*, F_{TAS}(**Z**) contient en particulier la valeur *Ok entasse(-1,vide)* qui n'est pas finiment générée.

Il est clair que l'on peut remédier à ce désagrément en remplaçant la seconde déclaration de formes *Ok* par :

$$x \in \text{Ok-Frm} \ \wedge\ t \in \text{Ok-Frm} \quad \Rightarrow \quad \mathit{entasse}(x,t) \in \text{Ok-Frm}$$

C'est ce qu'exprime le théorème suivant, qui prouve que l'*Ok*-complétude est syntaxiquement très facile à obtenir.

Théorème 6 :
Pour que **PRES** soit *Ok*-complète, il suffit que les deux conditions suivantes soient satisfaites :

- □ pour toute déclaration élémentaire de $\mathbf{Ok\text{-}Frm_{PRES}}$:

$$\left.\begin{array}{r} t_1 \in \text{Ok-Frm} \ \wedge \cdots \wedge\ t_m \in \text{Ok-Frm} \\ \wedge\ x_1 \in l_1 \ \wedge \cdots \wedge\ x_n \in l_n \\ \wedge\ v_1 = w_1 \ \wedge \cdots \wedge\ v_p = w_p \end{array}\right\} \quad \Rightarrow \quad t \in \text{Ok-Frm}$$

  toute variable apparaissant dans *t* doit apparaitre dans l'un au moins des t_i

- □ aucun axiome d'étiquetage de $\mathbf{Lbl\text{-}Ax_{PRES}}$ ne conclut sur l'étiquette *Ok*.

Preuve :
Soit A une **SPEC**-algèbre post-initiale quelconque. Récursivement, par construction minimale du foncteur de synthèse $F_{\mathbf{PRES}}$, tous les termes *Ok* de $T_{\mathbf{\Sigma}(A)}$ sont éléments de $T_{\mathbf{\Sigma}_{\mathbf{SPEC+PRES}}(A_{Ok})}$. De plus, puisque $\mathbf{Lbl\text{-}Ax}_{\mathbf{PRES}}$ ne contient aucun axiome relatif à l'étiquette *Ok*, une valeur de $F_{\mathbf{PRES}}(\mathrm{A})$ est *Ok* si et seulement si elle contient un *Ok*-terme dans sa classe. La conclusion résulte donc du fait que A_{Ok} est finiment généré lorsque A est post-initiale. □

Bien que ce théorème ne fournisse qu'une condition *suffisante* pour assurer l'*Ok*-complétude, cette condition est toujours satisfaite en pratique. En particulier tous les exemples d'exception-présentation fournis dans cette thèse sont *Ok*-complets.
Remarquons que l'*Ok*-complétude n'impose pas que $F_{\mathbf{PRES}}(\mathrm{A})_{Ok}$ soit rétractable sur $T_{\mathbf{PRES},Ok}$; cette propriété sera impliquée par la "post-rétractabilité" :

Définition 28 :
Une présentation **PRES** au-dessus d'une exception-spécification **SPEC** est dite *post-rétractable* si et seulement si pour toute **SPEC**-algèbre post-initiale A, le morphisme initial

$$Init_{F_{\mathbf{PRES}}(A)} : \quad T_{\mathbf{PRES}} \quad \rightarrow \quad F_{\mathbf{PRES}}(\mathrm{A})$$

est partiellement rétractable.

Voici un exemple de présentation persistante qui n'est pas post-rétractable.

Exemple 19 :
Reprenons la présentation *TAS* de l'exemple précédent, et ajoutons lui l'axiome généralisé suivant :

$$succ(n) = 0 \quad \Rightarrow \quad entasse(x,vide) = vide$$

Du fait que $T_{NAT+TAS}$ ne contient aucune valeur négative dans la sorte *NAT*, $T_{NAT+TAS}$ n'est autre qu'une structure de piles d'entiers (mais sans opérations d'accès aux piles). Considérons de nouveau la *NAT*-algèbre post-initiale **Z** avec $\mathbf{Z}_{Ok}$=**N** . La (*NAT+TAS*)-algèbre $F_{TAS}(\mathbf{Z})$ vérifie *succ(-1)=0* ; si bien que notre axiome s'applique, et $F_{TAS}(\mathbf{Z})$ est réduit à {*vide*} dans la sorte *TAS*. Il en résulte que cette présentation n'est pas post-rétractable.
Elle est pourtant persistante car, ne contenant aucune opération de cible dans *NAT*, elle ne peut induire aucune inconsistance ou incomplétude dans *NAT*.

Contrairement à l'*Ok*-complétude, il semble difficile de fournir des conditions suffisantes simples pour la post-rétractabilité. Par exemple, imposer que toute variable des prémisses apparaisse aussi dans la conclusion des axiomes ne résout rien : on peut obtenir la même inconsistance en remplaçant l'axiome :

$$succ(n) = 0 \quad \Rightarrow \quad entasse(x,vide) = vide$$

par :

$$succ(n) = 0 \quad \Rightarrow \quad entasse(n,entasse(x,vide)) = vide$$

et en ajoutant une opération "*detasse*" avec :

$$detasse(entasse(x,t)) = t$$
$$detasse(vide) = vide$$

L'algèbre initiale sera encore une structure de piles car elle ne contient aucune valeur négative, tandis que $F_{TAS}(\mathbf{Z})$ est réduite à {*vide*} dans la sorte *TAS*.
Imposer que toute variable de la conclusion apparaisse dans les prémisses ne résout pas plus la question ; il suffit par exemple d'ajouter une tautologie dans les prémisses :

$$succ(n)=0 \wedge succ(x)=x+1 \quad \Rightarrow \quad entasse(x,vide) = vide$$

De plus, les raisonnements par induction structurelle ne peuvent pas être utilisés puisque les algèbres post-initiales ne sont pas finiment générées. Par contre, les raisonnements de type "remplacement d'égal par égal" restent valides.

Posons maintenant la définition de stabilité :

Définition 29 :
Une présentation **PRES** au-dessus de l'exception-spécification **SPEC** est dite *stable* si et seulement si elle est *Ok*-complète et post-rétractable.

Proposition 5 :
PRES est stable au-dessus de **SPEC** si et seulement si pour toute **SPEC**-algèbre post-initiale A, $F_{\mathbf{PRES}}(\mathrm{A})$ est une (**SPEC**+**PRES**)-algèbre post-initiale.
(résulte directement des définitions)

La stabilité des algèbres post-initiales via le foncteur de synthèse $F_{\mathbf{PRES}}$ étant assurée par la propriété de *stabilité*, nous allons maintenant étudier la stabilité des algèbres post-initiales via le foncteur d'oubli : la persistance forte (suffisante complétude forte et consistance hiérarchique forte).

4. LA COMPLETUDE SUFFISANTE FORTE

Définition 30 :
Une présentation **PRES** au-dessus de l'exception-spécification **SPEC** est dite *fortement complète* si et seulement si elle est *suffisamment complète* et *stable*.

Remarquons que par définition même, la complétude suffisante forte implique la complétude suffisante.

La stabilité assure que le foncteur de synthèse $F_{\mathbf{PRES}}$ préserve les algèbres post-initiales ; et rappelons que la suffisante complétude signifie que toute nouvelle valeur ajoutée par **PRES** à l'algèbre initiale $T_{\mathbf{SPEC}}$ est erronée. Soulignons que la suffisante complétude ne concerne a priori que l'algèbre initiale ; néanmoins nous avons le résultat suivant :

Théorème 7 :
Si **PRES** est fortement complète au-dessus de **SPEC** alors pour toute **SPEC**-algèbre post-initiale A, la partie *Ok* de $U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A))$ est $\mathbf{\Sigma}_{\mathbf{SPEC}}$-finiment généré. Mieux : $U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A))_{Ok}$ est égal à l'image de $T_{\mathbf{SPEC},Ok}$ par le morphisme initial.

Preuve :
La stabilité assure que $F_{\mathbf{PRES}}(A)_{Ok}$ est égal à $Init_{F_{\mathbf{PRES}}(A)}(T_{\mathbf{PRES},Ok})$. Il en résulte que $U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A)_{Ok})$ est égal à l'image de $U_{\mathbf{SPEC}}(T_{\mathbf{PRES},Ok})$ par le morphisme

$$U(Init) : \ U_{\mathbf{SPEC}}(T_{\mathbf{PRES}}) \ \rightarrow \ U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A)) .$$

Or la suffisante complétude s'écrit :

$$U_{\mathbf{SPEC}}(T_{\mathbf{PRES}} - T_{\mathbf{PRES},err}) \ \subset \ I_{T_{\mathbf{SPEC}}}(T_{\mathbf{SPEC}})$$

et du fait que les valeurs erronées ne sont pas *Ok* elle implique :

$$U_{\mathbf{SPEC}}(T_{\mathbf{PRES},Ok}) \ \subset \ I_{T_{\mathbf{SPEC}}}(T_{\mathbf{SPEC}})$$

$I_{T_{\mathbf{SPEC}}}$ étant un exception-morphisme, il vient :

$$U_{\mathbf{SPEC}}(T_{\mathbf{PRES},Ok}) = I_{T_{\mathbf{SPEC}}}(T_{\mathbf{SPEC},Ok})$$

Par composition des exception-morphismes, la conclusion est acquise :

$$U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A)_{Ok}) \ = \ Init_{U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A))}(T_{\mathbf{SPEC},Ok}) \ \ \square$$

5. LA CONSISTANCE HIERARCHIQUE FORTE

Définition 31 :
Etant donnée une exception-spécification **SPEC**, une exception-présentation **PRES** au-dessus de **SPEC** est dite *fortement consistante* si et seulement si pour toute **SPEC**-algèbre post-initiale A , le morphisme d'adjonction

$$I_A : \ A \ \rightarrow \ U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A))$$

est partiellement rétractable.

Remarquons que cette définition est la même que celle de la consistance hiérarchique à ceci près qu'elle concerne toutes les algèbres post-initiales, et pas seulement l'algèbre initiale.
Il est clair que l'algèbre initiale $T_{\mathbf{SPEC}}$ est elle même une algèbre post-initiale, par conséquent la proposition suivante est immédiate :

Proposition 6 :
Si une présentation est fortement consistante alors elle est consistante.

Par ailleurs, les présentations fortement consistantes préservent la rétractabilité des algèbres post-initiales.

Théorème 8 :
Si **PRES** est une présentation fortement hiérarchiquement consistante au-dessus de **SPEC**, alors pour toute **SPEC**-algèbre post-initiale A, le morphisme initial

$$Init_{U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A))} : \ T_{\mathbf{SPEC}} \ \rightarrow \ U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A))$$

est partiellement rétractable.

Preuve :
La consistance forte assure que le morphisme d'adjonction

$$I_A : \ A \ \rightarrow \ U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A))$$

est partiellement rétractable ; et la post-initialité de A implique que

$$init_A : \ T_{\mathbf{SPEC}} \ \rightarrow \ A$$

est partiellement rétractable. La conclusion résulte donc du fait que le composée de deux morphismes partiellement rétractables est encore partiellement rétractable. □

Voici maintenant un exemple de présentation consistante qui n'est pas fortement consistante.

Exemple 20 :
Considérons la specification **SPEC**=*NAT+BOOL* munie des opérations *true*, *false*, 0 et *succ* habituelles, la déclaration de formes *Ok* suivante :

$$true \in \text{Ok-Frm}$$
$$false \in \text{Ok-Frm}$$
$$0 \in \text{Ok-Frm}$$
$$n \in \text{Ok-Frm} \ \Rightarrow \ succ(n) \in \text{Ok-Frm}$$

les ensembles d'axiomes **Ok-Ax**, **Lbl-Ax** et **Gen-Ax** vides.
L'algèbre initiale $T_{\mathbf{SPEC}}$ est égale à $\mathbf{N} \cup \{true,false\}$, et $\mathbf{N}_{Ok} = \mathbf{N} \cup \{true,false\}$.
Considérons maintenant la présentation **PRES** au-dessus de *NAT* n'ajoutant aucune sorte, ajoutant l'opération *eq?(_ ,_)* , définie par les axiomes *généralisés* (**Gen-Ax**) suivants :

$$eq?(0,0) = true$$
$$eq?(0,succ(n)) = false$$
$$eq?(succ(m),0) = true$$
$$eq?(succ(m),succ(n)) = eq?(m,n)$$

Cette présentation est clairement hiérarchiquement consistante ; par contre elle n'est pas *fortement* hiérarchiquement consistante. En effet, spécifier l'opération *eq?(_ ,_)* au moyen d'axiomes *généralisés* n'est pas judicieux ; il est clair que ces axiomes spécifient les cas "normaux" (c'est-à-dire les cas *Ok*) et qu'il ne devraient pas s'appliquer aux cas exceptionnels ou erronés.
Pour constater que cette présentation n'est pas fortement consistante, il suffit de considérer la *NAT+BOOL*-algèbre $\mathbf{Z} \cup \{true,false\}$ munie de $\mathbf{Z}_{Ok}=\mathbf{N} \cup \{true,false\}$.
Du fait que *eq?(_ ,_)* est spécifiée par des axiomes généralisés, ceux-ci s'appliquent aussi aux valeurs non-*Ok* et l'on obtient l'inconsistance suivante dans **Z** :

$$true = eq?(0,0) = eq?(0,succ(-1)) = false$$

Il est clair que si l'opération *eq?(_ ,_)* avait été spécifiée par des *Ok*-axiomes alors **PRES** aurait été fortement consistante car d'une part les *Ok*-axiomes ne s'appliquent qu'aux termes *Ok*, et d'autre part la partie *Ok* de toute algèbre post-initiale est isomorphe à $\mathbf{N} \cup \{true,false\}$.

On constate donc que la consistance hiérarchique forte est un critère très utile, plus sélectif que la seule consistance hiérarchique, et qui permet des spécifications mieux structurées. L'usage des exception-algèbres post-initiales est donc crucial.

6. LA PERSISTANCE FORTE

Définition 32 :
Etant donnée une exception-spécification **SPEC**, une exception-présentation **PRES** au-dessus de **SPEC** est dite *fortement persistante* si et seulement si elle est fortement complète et fortement consistante.

Rappelant que d'une part la complétude forte implique la complétude, et d'autre part la consistance forte implique la consistance, il en résulte que la persistance forte implique la persistance.

Remarquons que, par définition de la complétude forte, une exception-présentation est fortement persistante si et seulement si elle est suffisamment complète, stable, et fortement consistante.

La persistance forte permet de bâtir des spécifications structurées ayant un comportement "sain", comme l'attestent les résultats divers énoncés ci-dessous :

Lemme 7 :
Les foncteurs d'oubli conservent *toujours* les morphismes partiellement rétractables. Ce qui s'énonce encore : pour toute présentation **PRES** au-dessus de **SPEC**, si μ

$$\mu : \quad B \quad \rightarrow \quad B'$$

est un (**SPEC**+**PRES**)-morphisme partiellement rétractable, alors $U_{\mathbf{SPEC}}(\mu)$

$$U_{\mathbf{SPEC}}(\mu) : \quad U_{\mathbf{SPEC}}(B) \quad \rightarrow \quad U_{\mathbf{SPEC}}(B')$$

est encore un **SPEC**-morphisme partiellement rétractable.
(Résulte immédiatement des définitions)

Théorème 9 :
Si **PRES** est une exception-présentation fortement persistante au-dessus de **SPEC** alors l'oubli de toute (**SPEC**+**PRES**)-algèbre post-initiale est encore une **SPEC**-algèbre post-initiale.

Preuve :
Soit B une (**SPEC**+**PRES**)-algèbre post-initiale. Ceci signifie que B_{Ok} est $\mathbf{\Sigma}$(**SPEC**+**PRES**)-finiment généré et que le morphisme initial :

$$T_{\mathbf{PRES}} \quad \rightarrow \quad B$$

est partiellement rétractable.
Prouvons d'abord que $U_{\mathbf{SPEC}}(B)_{Ok}$ est $\mathbf{\Sigma(SPEC)}$-finiment généré. Du fait que B_{Ok} est finiment généré et que la partie finiment générée de B est isomorphe à $T_{\mathbf{PRES}}$, il résulte que $U_{\mathbf{SPEC}}(B)_{Ok}$ est isomorphe à $U_{\mathbf{SPEC}}(T_{\mathbf{PRES},Ok})$. La complétude suffisante de **PRES** nous permet alors de conclure que $U_{\mathbf{SPEC}}(T_{\mathbf{PRES},Ok})$, et donc $U_{\mathbf{SPEC}}(B)_{Ok}$, est $\mathbf{\Sigma(SPEC)}$-finiment généré.
D'autre part, le lemme précédent implique que le morphisme

$$U_{\mathbf{SPEC}}(init_B) : \; U_{\mathbf{SPEC}}(T_{\mathbf{PRES}}) \;\rightarrow\; U_{\mathbf{SPEC}}(B)$$

est partiellement rétractable. Mais la persistance de **PRES** assure que $U_{\mathbf{SPEC}}(T_{\mathbf{PRES}})$ est partiellement retractable sur $T_{\mathbf{SPEC}}$; par composition, il en est de même pour $U_{\mathbf{SPEC}}(B)$. Il en résulte que $U_{\mathbf{SPEC}}(B)$ est post-initiale. □

Le théorème précédent signifie que si une présentation **PRES** est fortement persistante, alors le foncteur d'oubli associé conserve les exception-algèbres post-initiales. Sachant que la persistance forte implique la stabilité, on constate que la persistance forte implique que les algèbres post-initiales sont stables par les foncteurs d'oubli et de synthèse.
En particulier, le foncteur de synthèse *suivi* du foncteur d'oubli préserve les algèbres post-initiales :

Proposition 7 :
Si **PRES** est une présentation fortement persistante au-dessus de l'exception-spécification **SPEC**, alors pour toute **SPEC**-algèbre post-initiale A, $U_{\mathbf{SPEC}}(F_{\mathbf{PRES}}(A))$ est encore une **SPEC**-algèbre post-initiale.
(Immédiat)

Enonçons enfin le théorème suivant, qui sera utile pour traiter la réutilisation d'implémentations abstraites avec traitement d'exceptions.

Théorème 10 :
Si $\mathbf{PRES_1}$ et $\mathbf{PRES_2}$ sont deux présentations fortement persistantes, de signatures disjointes, au-dessus d'une exception-spécification **SPEC**, alors $\mathbf{PRES_2}$ est encore une présentation fortement persistante au-dessus de ($\mathbf{SPEC}$+$\mathbf{PRES_1}$) .

Preuve :
Démontrons d'abord que $\mathbf{PRES_2}$ est suffisamment complète au-dessus de $\mathbf{SPEC}$+$\mathbf{PRES_1}$. Cette propriété ne concerne que les algèbres initiales $T_{\mathbf{SPEC}+\mathbf{PRES_1}}$ et sa synthèse $T_{\mathbf{SPEC}+\mathbf{PRES_1}+\mathbf{PRES_2}}$; il en résulte que toute valeur ajoutée par $\mathbf{PRES_2}$ dans les sortes de $\mathbf{SPEC}$+$\mathbf{PRES_1}$ est la classe d'un terme fermé contenant des opérations de $\mathbf{\Sigma}_{\mathbf{PRES_2}}$. Si ce terme ne contient que des opérations de $\mathbf{SPEC}$+$\mathbf{PRES_1}$ alors la suffisante complétude de $\mathbf{PRES_1}$ assure qu'il est soit erroné, soit congru à une valeur prédéfinie de $T_{\mathbf{SPEC}}$. Si par contre il contient des opérations de $\mathbf{PRES_2}$ alors la suffisante complétude de $\mathbf{PRES_2}$ assure (récursivement) que s'il n'est pas finiment généré alors il contient une sous-terme erroné ; la propagation des erreurs prouve alors qu'il est erroné, ou récupéré, auquel cas c'est qu'il est congru à une valeur *Ok* donc $\mathbf{\Sigma}_{\mathbf{SPEC}+\mathbf{PRES_1}}$-finiment généré (par stabilité de $\mathbf{PRES_1}$). Ceci prouve donc la complétude suffisante.
Il ne nous reste plus qu'à démontrer que pour toute ($\mathbf{SPEC}$+$\mathbf{PRES_1}$)-algèbre post-initiale B, les deux propriétés suivantes sont vérifiées :

- □ $F_{\mathbf{PRES_2}}$(B) est une ($\mathbf{SPEC}$+$\mathbf{PRES_1}$ + $\mathbf{PRES_2}$)-algèbre post-initiale [stabilité]
- □ le morphisme d'adjonction
  $$I : \; \mathrm{B} \;\rightarrow\; U_{\mathbf{SPEC}+\mathbf{PRES_1}}(F_{\mathbf{PRES_2}}(\mathrm{B}))$$
  est partiellement rétractable [consistance forte].

Pour simplifier les notations de cette démonstration, notons $\mathbf{\Sigma}big$ la signature $\mathbf{\Sigma}_{\mathbf{SPEC}+\mathbf{PRES_1}+\mathbf{PRES_2}}$.

Rappelons que

$$F_{\mathbf{PRES_2}}(\mathrm{B}) \;=\; (\, T_{\mathbf{\Sigma}big(B)} \,/\, [\equiv_{eval_B} \cup \equiv_{\mathbf{SPEC}} \cup \equiv_{\mathbf{PRES_1}} \cup \equiv_{\mathbf{PRES_2}}] \,)$$

(munie des ensembles étiquetés minimaux contenant les B_l) où $[\equiv_{eval_B} \cup \equiv_{\mathbf{SPEC}} \cup \equiv_{\mathbf{PRES_1}} \cup \equiv_{\mathbf{PRES_2}}]$ est la plus petite congruence engendrée par :

- □ la congruence $\equiv_{eval_B}$ associée au morphisme d'évaluation $T_{\mathbf{\Sigma}_{\mathbf{SPEC+PRES_1}}(B)} \rightarrow B$
- □ la congruence $\equiv_{\mathbf{SPEC}}$ engendrée par les axiomes et déclarations de la spécification **SPEC**
- □ la congruence $\equiv_{\mathbf{PRES_1}}$ engendrée par les axiomes et déclarations de $\mathbf{PRES_1}$
- □ la congruence $\equiv_{\mathbf{PRES_2}}$ engendrée par les axiomes et déclarations de $\mathbf{PRES_2}$.

Considérons la $\mathbf{\Sigma}big$-algèbre B_1 suivante :

$$B_1 \;=\; (\, T_{\mathbf{\Sigma}big(B)} \,/\, [\equiv_{eval_B} \cup \equiv_{\mathbf{SPEC}} \cup \equiv_{\mathbf{PRES_1}}] \,)$$

Du fait que la signature de $\mathbf{PRES_2}$ est disjointe de celle de $\mathbf{SPEC}$+$\mathbf{PRES_1}$, il résulte que tout terme contenant une opération de $\mathbf{PRES_2}$ est le seul élément de sa classe dans B_1 et n'est étiqueté par aucune étiquette de $\mathbf{L}_{\mathbf{SPEC+PRES_1}} \cup \{Ok\}$. Puisque B est une $\mathbf{SPEC}$+$\mathbf{PRES_1}$-algèbre post-initiale et que $\mathbf{PRES_1}$ est fortement persistant, ceci implique que le morphisme canonique

$$\mathrm{B} \;\rightarrow\; U_{\mathbf{SPEC+PRES_1}}(B_1)$$

est partiellement rétractable ; et puisqu'aucun terme contenant une opération de $\mathbf{PRES_2}$ n'est étiqueté, ceci implique aussi que $U_{\mathbf{SPEC+PRES_1}}(B_1)$ est une ($\mathbf{SPEC}$+$\mathbf{PRES_1}$)-algèbre post-initiale.
Enfin, par construction, et du fait que $U_{\mathbf{SPEC+PRES_1}}(B_1)$ est un quotient de B, on a :

$$F_{\mathbf{PRES_2}}(\mathrm{B}) \;=\; (\, B_1 \,/\, [\equiv_{\mathbf{SPEC}} \cup \equiv_{\mathbf{PRES_1}} \cup \equiv_{\mathbf{PRES_2}}] \,)$$

Considérons maintenant l'algèbre B_2 :

$$B_2 \;=\; (\, B_1 \,/\, [\equiv_{\mathbf{SPEC}} \cup \equiv_{\mathbf{PRES_2}}] \,)$$

Le même raisonnement prouve que : le morphisme canonique

$$\mathrm{B} \;\rightarrow\; U_{\mathbf{SPEC+PRES_2}}(B_2)$$

est partiellement rétractable ; et $U_{\mathbf{SPEC+PRES_2}}(B_2)$ est une ($\mathbf{SPEC}$+$\mathbf{PRES_2}$)-algèbre post-initiale. De plus :

$$F_{\mathbf{PRES_2}}(\mathrm{B}) \;=\; (\, B_2 \,/\, [\equiv_{\mathbf{SPEC}} \cup \equiv_{\mathbf{PRES_1}} \cup \equiv_{\mathbf{PRES_2}}] \,)$$

Mais puisque les signatures de $\mathbf{PRES_1}$ et $\mathbf{PRES_2}$ sont disjointes, que B_2 valide $\mathbf{PRES_2}$, et que les amalgames relatifs à $\mathbf{PRES_1}$ ont déjà été effectués au niveau de B_1, il en résulte qu'aucun nouvel amalgame issu de $\mathbf{PRES_1}$ ne peut déclancher de nouvelle occurrence d'axiome ou déclaration de $\mathbf{PRES_2}$; il en résulte que :

$$(\, B_2 \,/\, [\equiv_{\mathbf{SPEC}} \cup \equiv_{\mathbf{PRES_1}} \cup \equiv_{\mathbf{PRES_2}}] \,) \;=\; (\, B_2 \,/\, [\equiv_{\mathbf{SPEC}} \cup \equiv_{\mathbf{PRES_1}}] \,)$$

c'est-à-dire :

$$F_{\mathbf{PRES_2}}(\mathrm{B}) \;=\; (\, B_2 \,/\, [\equiv_{\mathbf{SPEC}} \cup \equiv_{\mathbf{PRES_1}}] \,)$$

Or, puisque $\mathbf{PRES_2}$ est fortement consistante au-dessus de $\mathbf{SPEC}$ et $U_{\mathbf{SPEC+PRES_2}}(B_2)$ est post-initiale, $U_{\mathbf{SPEC}}(B_2)$ est une $\mathbf{SPEC}$-algèbre post-initiale. Utilisant de nouveau le fait que les signatures de $\mathbf{PRES_1}$ et $\mathbf{PRES_2}$ sont disjointes, on déduit que la partie de $F_{\mathbf{PRES_2}}(\mathrm{B})$ relative aux sortes de $\mathbf{PRES_2}$ est isomorphe à la partie de B_2 relative à ces mêmes sortes ; et du fait que $\mathbf{PRES_1}$ est fortement persistance au-dessus de $\mathbf{SPEC}$, on a :

$$U_{\mathbf{SPEC+PRES_1}}(F_{\mathbf{PRES_2}}(\mathrm{B})) \text{ est post-initiale}$$

et le morphisme canonique

$$U_{\mathbf{SPEC}}(\mathrm{B}) \rightarrow U_{\mathbf{SPEC}}(F_{\mathbf{PRES_2}}(\mathrm{B})) \text{ est partiellement rétractable}$$

Il en résulte, par recollement que :

- □ $F_{\mathbf{PRES_2}}(\mathrm{B})$ est une ($\mathbf{SPEC}$+$\mathbf{PRES_1}$ + $\mathbf{PRES_2}$)-algèbre post-initiale
- □ le morphisme canonique (qui est donc le morphisme d'adjonction)

$$\mathrm{B} \;\rightarrow\; U_{\mathbf{SPEC+PRES_1}}(F_{\mathbf{PRES_2}}(\mathrm{B}))$$

est partiellement rétractable.

Ceci prouve finalement que $\mathbf{PRES_2}$ et fortement persistante au-dessus de ($\mathbf{SPEC}$+$\mathbf{PRES_1}$). □

Corollaire 1 :
Sous les mêmes hypothèses que le théorème précédent, ($\mathbf{PRES_1}$ + $\mathbf{PRES_2}$) est fortement persistant au-dessus de **SPEC**.

Preuve :
Pour toute **SPEC**-algèbre post-initiale A, $F_{\mathbf{PRES_1}+\mathbf{PRES_2}}(A) = F_{\mathbf{PRES_2}}(F_{\mathbf{PRES_1}}(A))$. La persistance forte de $\mathbf{PRES_1}$ implique que $F_{\mathbf{PRES_1}}(A)$ est une ($\mathbf{SPEC}$+$\mathbf{PRES_1}$)-algèbre post-initiale, et le morphisme d'adjonction est partiellement rétractable. Le théorème précédent, appliqué à B=$F_{\mathbf{PRES_1}}(A)$, conclut. □

Chapitre IX :

Conclusion

Nous avons montré dans cette partie B comment le traitement d'exceptions peut être intégré dans le formalisme des types abstraits algébriques sans perdre les notions classiques de *congruences*, *congruences minimales*, *objets initiaux*, *enrichissements*, *consistance hiérarchique* et *suffisante complétude*.

Nous avons développé :

- La modélisation des diagnostics d'exception au moyen des *étiquettes d'exception*, et des *axiomes d'étiquetage*
- La déclaration des valeurs *Ok* au moyen des déclarations de *formes Ok*
- Le traitement des cas *Ok* au moyen des *Ok-axiomes*
- Le traitement des cas exceptionnels (incluant la récupération) au moyen des *axiomes généralisés*
- La *propagation implicite des erreurs et des exceptions* incluse dans la définition des modèles : les *exception-algèbres*.

Soulignons que l'introduction des étiquettes d'exception nous permet en particulier de spécifier des récupérations "cas par cas" d'une manière simple ; et que malgré ces facilités de spécifications nous ne restreignons pas la puissance de modélisation. Par exemple, nous n'avons introduit aucune restriction sur les classes de modèles pris en considération pour assurer l'existence d'un objet initial ; et bien que la plupart des exemples fournis ici puissent s'orienter en des systèmes de réécriture convergents, cette condition n'est nullement requise.

D'autre part, notre sémantique permet de prendre en compte les exception-algèbres non finiment générées (ce qui permet en particulier de traiter les questions d'enrichissement en toute généralité), grâce à une idée simple :
il faut bien distinguer les *termes exceptionnels* des *valeurs erronées*, et cette distinction est possible même sur les algèbres non finiment générées grâce à la technique consistant à travailler dans l'algèbre des termes avec variables dans l'algèbre ($T_{\Sigma(A)}$).
En fait, la plupart des résultats développés durant cette partie reposent sur cette simple idée. En ce sens, le résultat le plus typique du formalisme des exception-algèbres est sans doute la proposition 1 du chapitre IV, car c'est sur cette technique de construction de la sémantique des *Ok*-axiomes que

repose le traitement d'exceptions décrit ici.

Enfin, du fait que l'on peut étendre au cas avec traitement d'exceptions les notions classiques de présentations, foncteurs d'oubli et de synthèse, consistance hiérarchique et suffisante complétude, la plupart des primitives classiques de structuration des spécifications peuvent facilement être étendues aux exception-algèbres. La partie suivante (partie C) en est encore un exemple : elle traite de l'implémentation abstraite avec traitement d'exceptions.

PARTIE C :

IMPLEMENTATION ABSTRAITE

EN PRESENCE D'EXCEPTIONS

Résumé de cette partie C

Cette partie montre comment on peut définir l'implémentation abstraite en présence d'exceptions, en utilisant les formalismes d'implémentation abstraite et de traitement d'exceptions respectivement définis en parties A et B de cette thèse.

Le chapitre I rappelle très brièvement les idées clefs de ces deux formalismes ; tandis que les autres chapitres décrivent pas à pas comment traduire avec traitement d'exceptions la démarche que nous avons déjà décrite sans traitement d'exceptions dans la partie A.

On constate que l'extension s'effectue sans difficulté, que les preuves de correction d'une implémentation abstraite avec traitement d'exceptions peuvent, là encore, être exprimées en termes de suffisante complétude et consistance hiérarchique ; mais puisqu'il faut travailler au niveau des algèbres post-initiales (pas seulement au niveau des algèbres initiales), il s'agit de consistance et complétude *fortes*.

TABLE DES MATIERES

TABLE DES MATIERES (SUITE)

Chapitre I :

Rappels

Ce chapitre rappelle très brièvement les formalismes de traitement d'exceptions et d'implémentation abstraite décrits dans les parties B et A de cette thèse.

1. LES EXCEPTION-ALGEBRES

Rappelons que nous avons défini les *exception-spécifications* comme suit :

SPEC = < **S** , **Σ** , **L** , **Ok-Frm** , **Ok-Ax** , **Lbl-Ax** , **Gen-Ax** >

où :

- <**S**,**Σ**,**L**> est une *exception-signature* ; c'est-à-dire que <**S**,**Σ**> est une signature au sens classique, et **L** est un ensemble d'*étiquettes d'exception* modélisant des diagnostics d'exception
- **Ok-Frm** est une *déclaration de formes Ok* ; elle permet de définir (récursivement) l'ensemble des formes *Ok*
- **Ok-Ax** est un ensemble d'*Ok-axiomes* ; ils ne s'appliquent qu'à des termes que l'on peut amalgamer avec une forme *Ok*, et ne traitent pas les termes exceptionnels
- **Lbl-Ax** est un ensemble d'*axiomes d'étiquetage* ; ils permettent d'attacher, récursivement, des étiquettes d'exception à certaines valeurs des exception-algèbres
- enfin **Gen-Ax** est un ensemble d'*axiomes généralisés* ; ils permettent de spécifier les divers traitements exceptionnels (amalgames de plusieurs valeurs erronées ou récupérations), ceci en utilisant éventuellement des étiquettes d'exception dans leurs prémisses.

La sémantique des exception-algèbres repose sur une distinction fondamentale entre les *termes exceptionnels* et les *valeurs erronées*. Rappelons en effet que, du fait qu'un *terme* exceptionnel peut être récupéré, ce n'est pas parce qu'il est exceptionnel que sa *valeur* est automatiquement erronée. Un tel terme nécessite alors un traitement exceptionnel ; il ne doit donc pas être traité par les *Ok*-axiomes (bien que sa valeur soit *Ok*) mais par les axiomes généralisés.
Par exemple, si l'on spécifie le type intervalle [*0...10*] . Le terme *succ(10)* est exceptionnel, mais on peut poser la récupération suivante dans **Gen-Ax** :

succ(10) = 10

dès lors, la valeur d'un terme tel que *pred(succ(10))* est égale à la valeur du terme *pred(10)*

(traitement exceptionnel de *succ(10)*), c'est-à-dire à *9*. Du fait que *pred(succ(10))* est exceptionnel (il contient *succ(10)*), l'*Ok*-axiome

$$pred(succ(n)) = n$$

ne s'applique nullement, *pred(succ(10)) n'est pas égal* à *10*.

Rappelons enfin que nous avons démontré l'existence d'un *foncteur de synthèse* (adjoint à gauche au foncteur d'oubli) associé à toute présentation, ce qui nous a permis de redéfinir de manière adéquate les notions de *suffisante complétude* et de *consistance hiérarchique*. Enfin nous avons dégagé une classe privilégiée d'exception-algèbres validant une exception-spécification (contenant entre autres l'algèbre initiale) : les exception-algèbres *post-initiales*. L'usage des exception-algèbres post-initiales permet de définir la *consistance hiérarchique forte* et la *complétude suffisante forte*, dont l'intérêt sera particulièrement démontré durant cette partie C.

2. IMPLEMENTATION ABSTRAITE

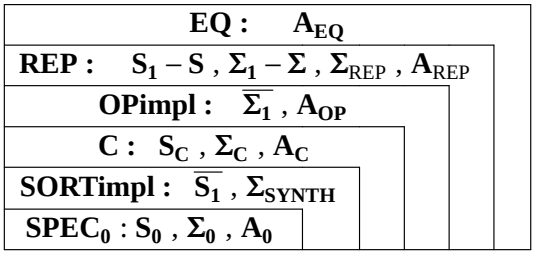
Le formalisme d'implémentation abstraite que nous avons développé dans la partie A est fondé sur deux idées principales :

- Utiliser à la fois des opérations de *synthèse* et des opérations de *représentation*. Les opérations de synthèse permettent de construire des "produits libres" dans des sortes intermédiaires, dites sortes *constructives*. Les opérations de *représentation* permettent la restriction aux produits réellement utilisés par l'implémentation, en faisant correspondre à chaque terme à implémenter l'ensemble des valeurs "constructives" qui le représentent.

- Inclure explicitement la *représentation de l'egalité* dans la description d'une implémentation, afin d'identifier deux objets "constructifs" (i.e. deux n-uplets obtenus par les opérations de synthèse) représentant le même objet à implémenter.

Une implémentation abstraite est alors spécifiée au moyen de cinq présentations :

Figure 1 : *Le "niveau des présentations" sans traitement d'exceptions*



où :

- $SPEC_0 = < S_0 , \Sigma_0 , A_0 >$ est la spécification de la structure de données déjà implémentée (dite *résidante*), tandis que $SPEC_1 = < S_1 , \Sigma_1 , A_1 >$ est la spécification à implémenter, et $P = <S,\Sigma,A>$ est la spécification commune à $SPEC_0$ et $SPEC_1$ ($P=SPEC_0 \cap SPEC_1$). Ces spécifications sont dites *descriptives* car elle décrivent les propriétés connues des structures qu'elles spécifient, mais nullement la manière "constructive" dont elles sont implémentés.

- $\overline{S_1}$ est l'ensemble des sortes "produit" synthétisées par les opérations de synthèses de

$\mathbf{\Sigma_{SYNTH}}$. $\overline{\mathbf{S_1}}$ est l'ensemble des sortes *constructives* car c'est au sein de ces sortes que nous spécifions "constructivement" l'implémentation abstraite.

- □ **C** est la composante cachée (elle facilite la spécification de l'implémentation)
- □ $\overline{\mathbf{\Sigma_1}}$ est une copie de l'ensemble des opérations à implémenter ; les opérations de $\overline{\mathbf{\Sigma_1}}$ prennent leur arité dans les sortes de $\overline{\mathbf{S_1}}$ et non dans $\mathbf{S_1}$; c'est pourquoi $\overline{\mathbf{\Sigma_1}}$ est appelé l'ensemble des opérations *constructives*. Leur implémentation est décrite au moyen des axiomes d'implémentation des opérations ($\mathbf{A_{OP}}$).
- □ $\mathbf{\Sigma}_{\text{REP}}$ est l'ensemble des opérations de représentation, et $\mathbf{A}_{\text{REP}}$ spécifie le fait que chaque opération de $\mathbf{\Sigma_1}$ est représentée par une opération de $\overline{\mathbf{\Sigma_1}}$.
- □ enfin $\mathbf{A_{EQ}}$ est un ensemble d'axiomes spécifiant la représentation de l'égalité.

Rappelons que $\mathbf{\Sigma}_{\text{REP}}$ et $\mathbf{A}_{\text{REP}}$ sont automatiquement déduits de l'isomorphisme de signatures entre celle à implémenter ($\mathbf{S_1}, \mathbf{\Sigma_1}$), et celle qui l'implémente ($\overline{\mathbf{S_1}}, \overline{\mathbf{\Sigma_1}}$). Par conséquent une implémentation abstraite peut être textuellement caractérisée par le quintuplet

$$\mathbf{IMPL} \;=\; < \rho \,,\, \mathbf{\Sigma_{SYNTH}} \,,\, \mathbf{C} \,,\, \mathbf{A_{OP}} \,,\, \mathbf{A_{EQ}} >$$

où ρ est cet isomorphisme de signatures.

Le niveau sémantique associé à une telle implémentation abstraite est alors particulièrement simple, puisque composé d'un foncteur de synthèse suivi d'un foncteur d'oubli :

$$\begin{array}{ccccc} \text{Alg}(\mathbf{SPEC_0}) & \rightarrow & \text{Alg}(\mathbf{SPEC_0}+\mathbf{SORTimpl}+\mathbf{C}+\mathbf{OPimpl}+\mathbf{REP}+\mathbf{EQ}) & \rightarrow & \text{Alg}(\mathbf{\Sigma_1}) \\ T_{\mathbf{SPEC_0}} & \rightarrow & T_{\mathbf{EQ}} = T_{\mathbf{SPEC_0}+\mathbf{SORTimpl}+\mathbf{C}+\mathbf{OPimpl}+\mathbf{REP}+\mathbf{EQ}} & \rightarrow & \text{SEM}_{\mathbf{IMPL}} \end{array}$$

Ce formalisme nous a permis de fournir des critères "simples" pour prouver qu'une implémentation est correcte (utilisant essentiellement la consistance hiérarchique). Plus précisément, ces critères ne reposent que sur la connaissance de la *spécification* de l'implémentation ; contrairement aux formalismes d'implémentation abstraite existants, nos critères n'imposent pas une description exhaustive de la sémantique.

3. IMPLEMENTATION ABSTRAITE AVEC EXCEPTIONS

Nous devons maintenant étendre cette définition d'implémentation abstraite au formalisme des exception-algèbres.
Grossièrement, les décisions qui nous incombent peuvent être décrites comme suit :

- □ "placer judicieusement" des ensembles d'etiquettes d'exception (**L**) et des déclarations de formes *Ok* (**Ok-Frm**) dans cette description d'une implémentation abstraite
- □ "remplacer judicieusement" les ensembles d'axiomes des présentations précédentes par des *Ok*-axiomes (**Ok-Ax**), des axiomes d'étiquetage (**Lbl-Ax**) ou des axiomes généralisés (**Gen-Ax**) [le "ou" est inclusif, bien sûr].

Les chapitres suivants décrivent notre formalisme d'implémentation abstraite étendu aux exception-algèbres, et motivent intuitivement les choix qui y sont faits. Nous suivrons la même démarche que dans la partie A de cette thèse :

- □ Définition du *niveau textuel*
- □ Description du *niveau des présentations* qui lui est associé
- □ Description du *niveau sémantique*
- □ Définition de la *correction* d'une implémentation abstraite
- □ *Réutilisation* d'implémentations abstraites.

Dans toute la suite, nous développerons l'exemple de l'implémentation des *files bornées* au moyen des *tableaux bornés*. D'autres exemples d'implémentations abstraites avec traitement d'exceptions sont décrits dans l'annexe 4.

Chapitre II :

Le niveau

textuel

---

1. REVISION DU CONTEXTE

Lorsque l'on veut définir une implémentation abstraite, on dispose au départ de deux spécifications algébriques (non necessairement disjointes) :

- la spécification algébrique de la structure de données *résidante* (déjà implémentée), par exemple celle des tableaux bornés (*ARRAY*), des entiers naturels bornés (*NAT*) et des éléments placés dans les tableaux (*ELEM*) ; on notera $\mathbf{SPEC_0}$ cette spécification, qui est donc maintenant une exception-spécification :
  $$\mathbf{SPEC_0} = < \mathbf{S_0} , \mathbf{\Sigma_0} , \mathbf{L_0} , \mathbf{Ok\text{-}Frm_0} , \mathbf{Ok\text{-}Ax_0} , \mathbf{Lbl\text{-}Ax_0} , \mathbf{Gen\text{-}Ax_0} >$$
  [rappelons que cette spécification est "descriptive" car elle décrit les propriétés abstraites connues de la structure résidante, mais ne fournit aucune indication sur la manière dont celle ci a été préalablement implémentée]

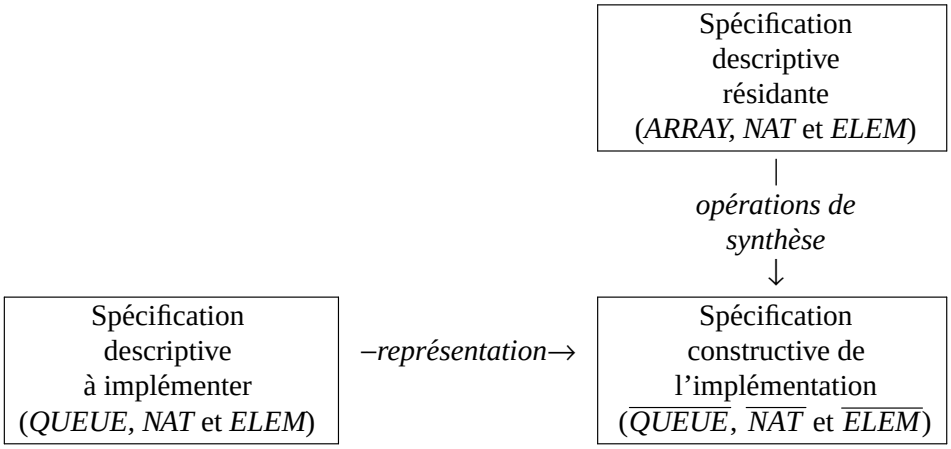
- la spécification algébrique de la structure de données que l'on veut implémenter, par exemple celle des files bornées (*QUEUE*), des entiers naturels bornés (*NAT*) et des éléments placés dans les files et les tableaux (*ELEM*) ; on notera $\mathbf{SPEC_1}$ cette spécification
  $$\mathbf{SPEC_1} = < \mathbf{S_1} , \mathbf{\Sigma_1} , \mathbf{L_1} , \mathbf{Ok\text{-}Frm_1} , \mathbf{Ok\text{-}Ax_1} , \mathbf{Lbl\text{-}Ax_1} , \mathbf{Gen\text{-}Ax_1} >$$
  [rappelons que $\mathbf{SPEC_1}$ est une spécification "descriptive" car elle décrit les propriétés que l'on veut obtenir après que l'implémentation soit faite, et non l'implémentation elle même]

- enfin la "partie commune" entre ces deux spécifications (par exemple *NAT* et *ELEM*) est elle même une exception-spécification, dite *prédéfinie* :
  $$\mathbf{SPEC_0 \cap SPEC_1 = P = <S,\Sigma,L,Ok\text{-}Frm,Ok\text{-}Ax,Lbl\text{-}Ax,Gen\text{-}Ax>}$$
  Nous supposerons que $\mathbf{SPEC_0}$ et $\mathbf{SPEC_1}$ sont toutes deux *fortement persistantes* au-dessus de **P**.

Comme dans le cas sans traitement d'exceptions, pour spécifier l'implémentation de $\mathbf{SPEC_1}$ au moyen de $\mathbf{SPEC_0}$ nous utiliserons une spécification intermédiaire (celle de l'implémentation) qui

travaille sur une signature “constructive” intermédiaire (une copie de $<\mathbf{S_1}, \mathbf{\Sigma_1}>$, notée $<\overline{\mathbf{S_1}}, \overline{\mathbf{\Sigma_1}}>$). La correspondance entre $\mathbf{SPEC_0}$ et la spécification intermédiaire est assurée par les *opérations de synthèse*, tandis que la correspondance entre $\mathbf{SPEC_1}$ et cette spécification intermédiaire est assurée par les *opérations de représentation*.

Figure 2 : *implémentation abstraite = représentation + synthèse*



Pour l’exemple de l’implémentation des files bornées par les tableaux bornés, les opérations de synthèse sont des opérations de copies (une pour chacune des sortes *NAT* et *ELEM*) et une opération produit (pour la sorte *QUEUE*) ; tandis que la représentation est un isomorphisme de signatures ($< \mathbf{S_1}, \mathbf{\Sigma_1} > - \rho \rightarrow < \overline{\mathbf{S_1}}, \overline{\mathbf{\Sigma_1}} >$). [Se reporter au chapitre II de la partie A pour plus de détails intuitifs].

2. DEFINITION

Le niveau textuel d’une implémentation abstraite est définit comme suit :

Définition 1 :
Une *implémentation abstraite* (dans le cadre des exception-algèbres), notée **IMPL**, est définie par :

$< \rho , \mathbf{\Sigma_{SYNTH}} , \mathbf{Ok\text{-}Frm_{SYNTH}} , \mathbf{C} , \mathbf{Ok\text{-}Ax_{OP}} , \mathbf{Lbl\text{-}Ax_{OP}} , \mathbf{Gen\text{-}Ax_{OP}} , \mathbf{Ok\text{-}Ax_{EQ}} , \mathbf{Gen\text{-}Ax_{EQ}} >$

où :

- ρ est un isomorphisme de signatures entre $<\mathbf{S_1}, \mathbf{\Sigma_1}>$ et $<\overline{\mathbf{S_1}}, \overline{\mathbf{\Sigma_1}}>$
- $\mathbf{\Sigma_{SYNTH}}$ est l’ensemble des opérations de synthèse
- $\mathbf{Ok\text{-}Frm_{SYNTH}}$ est une déclaration de formes *Ok* dans les sortes de $\overline{\mathbf{S_1}}$
- **C** est la composante cachée de l’implémentation abstraite
- $\mathbf{Ok\text{-}Ax_{OP}}$ est l’ensemble des *Ok*-axiomes d’implémentation des opérations
- $\mathbf{Lbl\text{-}Ax_{OP}}$ est l’ensemble des axiomes d’implémentation des étiquettes d’exception
- $\mathbf{Gen\text{-}Ax_{OP}}$ est l’ensemble des axiomes généralisés d’implémentation des opérations
- $\mathbf{Ok\text{-}Ax_{EQ}}$ est l’ensemble des *Ok*-axiomes de représentation de l’égalité

- et **Gen-Ax$_{EQ}$** est l'ensemble des axiomes généralisés de représentation de l'égalité.

Toutes ces composantes de l'implémentation abstraite sont définies précisément dans les sections suivantes.
Intuitivement, par rapport à l'implémentation abstraite sans traitement d'exceptions définie dans la partie A de cette thèse, nous avons effectué les généralisations suivantes :

- la représentation est traduite par un isomorphisme de signatures exactement comme dans la partie A
- nous avons complété les opérations de synthèse ($\mathbf{\Sigma_{SYNTH}}$) par des déclarations de formes *Ok* (**Ok-Frm$_{SYNTH}$**) qui synthétisent les termes *Ok* des sortes constructives
- la composante cachée sera bien sûr une exception-présentation
- nous avons remplacé les axiomes d'implémentation des opérations ($\mathbf{A_{OP}}$) par les trois ensembles **Ok-Ax$_{OP}$**, **Lbl-Ax$_{OP}$** et **Gen-Ax$_{OP}$** afin d'autoriser toutes les possibilités de traitement d'exceptions lors de l'implémentation
- enfin nous avons remplacé les axiomes de représentation de l'égalité ($\mathbf{A_{EQ}}$) par **Ok-Ax$_{EQ}$** et **Gen-Ax$_{EQ}$**, permettant ainsi de spécifier la représentation de l'égalité dans les cas "normaux" aussi bien que dans les cas exceptionnels.

3. LA REPRESENTATION

Cette section définit la composante "ρ" de **IMPL**.

La représentation est modélisée au moyen d'un isomorphisme de signatures entre $<\mathbf{S_1}, \mathbf{\Sigma_1}>$ et $<\overline{\mathbf{S_1}}, \overline{\mathbf{\Sigma_1}}>$ exactement comme dans le cas sans traitement d'exceptions, à savoir :

- on note $\overline{\mathbf{S_1}}$ une copie de $\mathbf{S_1}$; $\overline{\mathbf{S_1}}$ est l'ensemble des "*sortes constructives*" : pour chaque sorte à implémenter s de **SPEC$_1$**, on se donne une sorte constructive $\bar{s}$ qui la représente.
- pour chaque opération à implémenter de **SPEC$_1$**, $op \in \mathbf{\Sigma_1}$, d'arité op: $s_1\, s_2 \cdots s_n \rightarrow s$, on se donne l'*opération constructive* qui la représente, $\overline{op}$, dont l'arité est "translatée" vers les sortes constructives : $\overline{op}$: $\overline{s_1} \cdots \overline{s_n} \rightarrow \bar{s}$, où les $\overline{s_i}$ et $\bar{s}$ sont les sortes de $\overline{\mathbf{S_1}}$ représentant les s_i et s, comme définies plus haut. On note $\overline{\mathbf{\Sigma_1}}$ l'ensemble de ces opérations constructives.

ρ est simplement l'isomorphisme de signatures entre $< \mathbf{S_1}, \mathbf{\Sigma_1} >$ et $< \overline{\mathbf{S_1}}, \overline{\mathbf{\Sigma_1}} >$ qui fait correspondre la sorte constructive $\bar{s} \in \overline{\mathbf{S_1}}$ à chaque sorte à implémenter $s \in \mathbf{S_1}$, et l'opération constructive $\overline{op} \in \overline{\mathbf{\Sigma_1}}$ à chaque opération à implémenter $op \in \mathbf{\Sigma_1}$.

Une remarque s'impose ici :

Remarque 1 :
On peut se demander pourquoi ρ est un isomorphisme de signatures "classiques" (**S**,**Σ**), et non un isomorphisme d'exception-signatures (**S**,**Σ**,**L**).
En fait, contrairement aux opérations, les étiquettes d'exception ne sont pas attachées à une sorte précise : nous avons donné un exemple (les tableaux) pour lequel une étiquette d'exception (*RANGE-TOO-LARGE*) peut étiqueter des valeurs de sortes différentes (*ARRAY* et *ELEM*). Par conséquent, contrairement aux opérations ($op \vdash \rho \rightarrow \overline{op}$) il n'est pas nécessaire de renommer les étiquettes

d'exception ($l \vdash \rho \rightarrow \bar{l}$) pour les appliquer aux sortes constructives ($\bar{s}$).
Naturellement, si l'on tient, pour des raisons d'homogénéité, à ce que ρ soit un isomorphisme d'exception-signatures, rien n'empêche d'effectuer ce renommage. On pourra constater que le formalisme que nous développons ici n'en est nullement affecté.

Exemple 1 :
Pour l'implémentation abstraite des files bornées par les tableaux bornés, la représentation est l'isomorphisme de signatures suivant :

QUEUE	$\vdash\rho\rightarrow$	$\overline{QUEUE}$	*new*	$\vdash\rho\rightarrow$	$\overline{new}$
NAT	$\vdash\rho\rightarrow$	$\overline{NAT}$	*add*	$\vdash\rho\rightarrow$	$\overline{add}$
ELEM	$\vdash\rho\rightarrow$	$\overline{ELEM}$	*remove*	$\vdash\rho\rightarrow$	$\overline{remove}$
			first	$\vdash\rho\rightarrow$	$\overline{first}$
			length	$\vdash\rho\rightarrow$	$\overline{length}$
			0	$\vdash\rho\rightarrow$	$\bar{0}$
			succ	$\vdash\rho\rightarrow$	$\overline{succ}$
			... idem pour les opérations de *ELEM* ...		

Il est inutile de renommer les étiquettes d'exception (*OVERFLOW*, *UNDERFLOW* ...).
L'exception-spécification *descriptive* des files bornées (*QUEUE*) est donnée en annexe 3.

4. LA SYNTHESE DES SORTES D'IMPLEMENTATION

Cette section définit les composantes $\mathbf{\Sigma_{SYNTH}}$ et **Ok-Frm$_{\mathbf{SYNTH}}$** de **IMPL**.

Comme dans le cas sans traitement d'exceptions, les opérations de synthèse sont définies comme suit :
$\mathbf{\Sigma_{SYNTH}}$ est l'ensemble des *opérations de synthèse* : pour chaque sorte constructive $\bar{s} \in \overline{\mathbf{S_1}}$, $\mathbf{\Sigma_{SYNTH}}$ contient une opération qui la "synthétise" $< \cdots >_s : r_1 \cdots r_m \rightarrow \bar{s}$, où les sortes r_i sont des sortes résidantes ($\in \mathbf{S_0}$).

Naturellement, lorsque s est une sorte déjà implémentée, $s \in \mathbf{S}$ (rappelons que $\mathbf{S_0} \cap \mathbf{S_1} = \mathbf{S}$ n'est pas nécessairement vide), $< \_ >_s$ est en fait une opération de copie ($< \_ >_s : s \rightarrow \bar{s}$).

Maintenant que les opérations de synthèse sont définies, on sait que les sortes $\bar{s}$ contiennent des n-uplets, ou des copies, des sortes déjà implémentées (les opérations $< \cdots >_s : r_1 \cdots r_m \rightarrow \bar{s}$ étant des opérations "produit", éventuellement monocomposantes).
Cependant, on ne dispose d'aucune information quant aux n-uplets *Ok*. Il est donc nécessaire de spécifier quels sont les formes *Ok* des sortes $\bar{s}$. C'est là le rôle de la déclaration de formes *Ok* **Ok-Frm$_{\mathbf{SYNTH}}$**. Elle est définie comme suit.

Ok-Frm$_{\mathbf{SYNTH}}$ est une déclaration de formes *Ok* sur la signature $<\mathbf{S_0} \cup \overline{\mathbf{S_1}}, \mathbf{\Sigma_0} \cup \mathbf{\Sigma_{SYNTH}}, \mathbf{L_0}>$ dont les déclarations élémentaires

$$\left.\begin{array}{r} t_1 \in \text{Ok-Frm} \ \wedge \cdots \wedge \ t_m \in \text{Ok-Frm} \\ \wedge \ \ x_1 \in l_1 \ \wedge \cdots \wedge \ x_n \in l_n \\ \wedge \ \ v_1 = w_1 \ \wedge \cdots \wedge \ v_p = w_p \end{array}\right\} \quad \Rightarrow \quad t \in \text{Ok-Frm}$$

sont telles que la sorte de t est élément de $\overline{\mathbf{S_1}}$ et la sorte de chacun des v_i ou w_i est élément de $\mathbf{S_0}$.

Nous avons imposé que t appartienne à une sorte constructive ($\overline{\mathbf{S_1}}$) : c'est bien naturel puisque le but de **Ok-Frm$_{\mathbf{SYNTH}}$** est de caractériser les formes *Ok* des sortes constructives.
Nous avons de plus imposé que les équations des prémisses des déclarations élémentaires de **Ok-Frm$_{\mathbf{SYNTH}}$** ne concernent que les sortes déjà implémentées ($\mathbf{S_0}$). Ceci traduit le fait qu'à ce niveau, aucun axiome n'est spécifié concernant les sortes de $\overline{\mathbf{S_1}}$ (on n'a pas encore spécifié la représentation de l'égalité ni les axiomes d'implémentation ; pour l'instant, les éléments de sorte dans $\overline{\mathbf{S_1}}$ ne sont que des produits libres) ; par conséquent, les seules égalités pertinentes que l'on puisse tester sont celles concernant les sortes déjà implémentées ($\mathbf{S_0}$).

Exemple 2 :
Pour l'exemple de l'implémentation des files par tableaux et entiers naturels, les opérations de synthèse sont les suivantes :

$$\begin{array}{rrcl} <\_>_{ELEM} : & ELEM & \rightarrow & \overline{ELEM} \\ <\_>_{NAT} : & NAT & \rightarrow & \overline{NAT} \\ <\_,\_,\_>_{QUEUE} : & ARRAY\ NAT\ NAT & \rightarrow & \overline{QUEUE} \end{array}$$

Intuitivement, une file sera implémentée circulairement dans le tableau borné t de $< t, i, j >_{QUEUE}$. L'indice i pointera sur le premier élément placé dans la file, tandis que l'indice j pointera sur la place où doit être ajouté le prochain élément de la file. Par conséquent, si $i \leq j$ alors les éléments de la file sont ceux compris entre i et j (la file est vide lorsque $i=j$) ; et si $i>j$ alors les éléments de la file sont ceux compris entre i et la taille maximale de la file, suivis de ceux compris entre l'indice *0* et j (i inclus, j exclu).
Les déclarations élémentaires de formes *Ok* sont par exemple :

$$\begin{array}{rcl} e \in \text{Ok-Frm} & \Rightarrow & < e >_{ELEM} \in \text{Ok-Frm} \\ n \in \text{Ok-Frm} & \Rightarrow & < n >_{ELEM} \in \text{Ok-Frm} \\ t \in \text{Ok-Frm} \wedge i \in \text{Ok-Frm} \wedge j \in \text{Ok-Frm} & \Rightarrow & < t, i, j >_{QUEUE} \in \text{Ok-Frm} \end{array}$$

Ok-Frm$_{\mathbf{SYNTH}}$ spécifie alors qu'un terme de la forme $< r_1, \ldots, r_m >_s$ est une forme *Ok* si et seulement si chacun des r_i est une forme *Ok*.

Remarque 2 :
Pour chaque sorte s de $\mathbf{S}$, les déclarations de formes *Ok* concernant $\bar{s}$ n'ont pas à être spécifiées explicitement par le concepteur d'une implémentation abstraite. En effet, on peut construire automatiquement les déclarations élémentaires des sortes prédéfinies :

$$x \in \text{Ok-Frm} \quad \Rightarrow \quad < x >_s \in \text{Ok-Frm}$$

On peut penser, au vu de l'exemple précédent, qu'il en est de même pour les sortes de $\mathbf{S_1} - \mathbf{S}$ car, dans la plupart des exemples, on spécifiera les formes *Ok* comme suit :

$$r_1 \in \text{Ok-Frm} \wedge \cdots \wedge r_m \in \text{Ok-Frm} \quad \Rightarrow \quad < r_1, \ldots, r_m >_s \in \text{Ok-Frm}$$

Cependant, il est utile de pouvoir spécifier **Ok-Frm$_{\mathbf{SYNTH}}$** différemment pour certains exemples ; c'est le cas de l'exemple très simple suivant.

Exemple 3 :
Il s'agit simplement d'implémenter le type intervalle [*0...10*] (i.e. les entiers bornés avec *Max-int=10*) au moyen du type intervalle [*0...20*] . On a alors une opération de synthèse

$$<\_>_{[0...10]} : [0 \ldots 20] \rightarrow \overline{[0 \ldots 10]} .$$

Il est cette fois utile de pouvoir spécifier **Ok-Frm$_{\mathbf{SYNTH}}$** comme suit

$$i \in \text{Ok-Frm} \wedge i \leq 10 = True \quad \Rightarrow \quad < i >_{[0...10]} \in \text{Ok-Frm} .$$

5. LA COMPOSANTE CACHEE

La composante cachée **C** de **IMPL** est définie comme suit :

C est une exception-présentation au dessus de

$$\mathbf{SORTimpl} = \mathbf{SPEC_0} + < \overline{\mathbf{S_1}}, \mathbf{\Sigma_{SYNTH}}, \mathbf{Ok\text{-}Frm_{SYNTH}}, \mathbf{L_1} >$$

notée : $\mathbf{C} = <\mathbf{S_C}, \mathbf{\Sigma_C}, \mathbf{L_C}, \mathbf{Ok\text{-}Frm_C}, \mathbf{Ok\text{-}Ax_C}, \mathbf{Lbl\text{-}Ax_C}, \mathbf{Gen\text{-}Ax_C}>$

Exemple 4 :
Du fait que l'on veut implémenter les files de manière circulaire dans un tableau, on aura évidemment besoin d'une opération "modulo" sur les entiers naturels. Si cette opération n'est pas déjà spécifiée dans *NAT*, il nous faut la spécifier comme une opération cachée.
En fait, nous n'utiliserons l'opération *modulo* que pour calculer "*succ(n) modulo succ(Maxlength)*" (où *Maxlength* est la longueur maximale d'une file). Pour simplifier les axiomes d'implémentation qui vont suivre, nous allons donc plutôt définir une opération cachée "*Next*" telle que *Next(n)* est égal à *succ(n)* modulo *succ(Maxlength)*.
$\mathbf{S_C}$ est vide.
$\mathbf{\Sigma_C}$ contient l'opération *Next*, d'arité

$$Next : NAT \rightarrow NAT$$

$\mathbf{L_C}$ est vide.
$\mathbf{Ok\text{-}Frm_C}$ est vide.
On supposera de *NAT* contient déjà l'opération *div* (sinon, il suffit d'inclure l'étiquette *DIV-BY-0-ERROR* dans la composante cachée et d'y spécifier l'opération *div*). $\mathbf{Ok\text{-}Ax_C}$ contient alors l'*Ok*-axiome suivant :

$$Next(n) = succ(n) - (succ(Maxlength) \times (succ(n)\ div\ succ(Maxlength)))$$

Cet *Ok*-axiome s'applique toujours (car *succ(Maxlength)* n'est pas nul), $\mathbf{Lbl\text{-}Ax_C}$ sera donc vide ; et $\mathbf{Gen\text{-}Ax_C}$ est vide.

6. LES AXIOMES D'IMPLEMENTATION

Cette sous-section définit les *Ok*-axiomes d'implémentation des opérations ($\mathbf{Ok\text{-}Ax_{OP}}$), les axiomes d'implémentation des étiquettes d'exception ($\mathbf{Lbl\text{-}Ax_{OP}}$) et les axiomes généralisés d'implémentation des opérations ($\mathbf{Gen\text{-}Ax_{OP}}$), de l'implémentation abstraite **IMPL**.

Ces ensembles d'axiomes sont définis comme suit.
$\mathbf{Ok\text{-}Ax_{OP}}$ (resp. $\mathbf{Lbl\text{-}Ax_{OP}}$ / $\mathbf{Gen\text{-}Ax_{OP}}$) est un ensemble d'*Ok*-axiomes (resp. d'axiomes d'étiquetage / d'axiomes généralisés) sur l'exception-signature :

$$< \mathbf{S_0} \cup \overline{\mathbf{S_1}} \cup \mathbf{S_C}\ ,\ \mathbf{\Sigma_0} \cup \mathbf{\Sigma_{SYNTH}} \cup \mathbf{\Sigma_C} \cup \overline{\mathbf{\Sigma_1}}\ ,\ \mathbf{L_1} >$$

ils spécifient l'implémentation des opérations constructives ainsi que le traitement exceptionnel s'y afférant.

Exemple 5 :
Voici par exemple les axiomes d'implémentation des files bornées par les tableaux bornés.
Soit *Maxlength* la longueur maximale des files que l'on veut implémenter. Les *Ok*-axiomes d'implémentation des opérations peuvent être spécifiés comme suit :

$$
\begin{array}{rcl}
i \leq Maxlength = True & \Rightarrow & \overline{new} = <t, i, i>_{QUEUE} \\
 & & \overline{add}(<e>_{ELEM}, <t, i, j>_{QUEUE}) = <t[j]:=e, i, Next(j)>_{QUEUE} \\
eq(i,j) = False & \Rightarrow & \overline{remove}(<t, i, j>_{QUEUE}) = <t, Next(i), j>_{QUEUE} \\
eq(i,j) = False & \Rightarrow & \overline{first}(<t, i, j>_{QUEUE}) = <t[i]>_{ELEM} \\
 & & \overline{length}(<t, i, j>_{QUEUE}) = <Next((j + Maxlength) - i)>_{NAT}
\end{array}
$$

Intuitivement, une file est implémentée circulairement sur la partie du tableau comprise entre *0* et *Maxlength*. Bien sûr, si la taille maximale du tableau, *Maxrange* ([1]), est strictement inférieure à *Maxlength*, alors nous ne pourrons pas prouver que l'implémentation est correcte (elle ne sera pas valide). Remarquons que l'on implémente la longueur de la file via [*(j–i) modulo succ(Maxlength)*], néanmoins nous écrivons [*Next((j+Maxlength)–i)*] (qui lui est égal), afin d'éviter le cas où *(j–i)* est négatif (auquel cas il serait erroné et l'*Ok*-axiome définissant *length* ne s'appliquerait pas).

Les axiomes d'implémentation des étiquettes d'exception peuvent être spécifiés comme suit :

$$
\begin{array}{rcl}
Next(j) = i & \Rightarrow & \overline{add}(\bar{x}, <t, i, j>_{QUEUE}) \in OVERFLOW \\
\bar{X} \in OVERFLOW & \Rightarrow & \overline{add}(\bar{x}, \bar{X}) \in OVERFLOW \\
i = j & \Rightarrow & \overline{remove}(<t, i, j>_{QUEUE}) \in UNDERFLOW \\
\bar{X} \in UNDERFLOW & \Rightarrow & \overline{remove}(\bar{X}) \in UNDERFLOW \\
i = j & \Rightarrow & \overline{first}(<t, i, j>_{QUEUE}) \in QUEUE\text{-}IS\text{-}EMPTY
\end{array}
$$

Les axiomes généralisés d'implémentation sont évidemment directement dépendants des récupérations spécifiées dans $\mathbf{SPEC_1}$=*QUEUE*. Si aucune récupération n'est spécifiée dans *QUEUE*, alors $\mathbf{Gen\text{-}Ax_{OP}}$ est vide. Pour donner une exemple avec récupération, on peut imaginer le cas suivant.
Lorsque l'on ajoute un élément *x* (avec *add*) à une file pleine (i.e. de longueur *Maxlength*), on perd le premier élément de la file avant d'ajouter *x* (la file reste donc de longueur *Maxlenth*). Ceci peut par exemple être traduit dans la spécification *descriptive* de *QUEUE* par l'axiome généralisé de $\mathbf{Gen\text{-}Ax}_{QUEUE}$ suivant :

$$length(X) = Maxlength \Rightarrow add(x,X) = add(x,remove(X))$$

L'implémentation d'une telle récupération peut alors être *constructivement* spécifiée comme suit, grâce aux axiomes généralisés d'implémentation des opérations de $\mathbf{Gen\text{-}Ax_{OP}}$:

$$Next(j) = i \Rightarrow \overline{add}(<e>_{ELEM}, <t, i, j>_{QUEUE}) = <t[j]:=e, Next(i), Next(j)>_{QUEUE}$$

(on détermine en effet que la file est pleine en testant si le successeur de *j*, modulo *succ(Maxlength)*, est égal à *i*).

7. LA REPRESENTATION DE L'EGALITE

Cette section définit les *Ok*-axiomes ($\mathbf{Ok\text{-}Ax_{EQ}}$) et axiomes généralisés ($\mathbf{Gen\text{-}Ax_{EQ}}$) spécifiant la représentation de l'égalité.

Ces ensembles d'axiomes sont définis comme suit.
$\mathbf{Ok\text{-}Ax_{EQ}}$ (resp. $\mathbf{Gen\text{-}Ax_{EQ}}$) est un ensemble d'*Ok*-axiomes (resp. d'axiomes généralisés) pouvant utiliser toutes les opérations (et toutes les étiquettes d'exception) de l'implémentation abstraite.

(1) Ce tableau contient alors *Maxrange+1* places (on utilise le rang *0*).

Pour notre exemple (avec ou sans la récupération décrite précédemment), la représentation de l'égalité peut être spécifiée comme suit.

Exemple 6 :
Pour spécifier la représentation de l'égalité associée à l'implémentation des files bornées décrite précédemment, on peut spécifier **Gen-Ax$_{EQ}$** vide, et **Ok-Ax$_{EQ}$** égal à :

$$< t, i, i >_{QUEUE} = < t', k, k >_{QUEUE}$$

$$< t, i, j >_{QUEUE} = < t', k, l >_{QUEUE} \wedge t[j] = t'[l] \implies < t, i, Next(j) >_{QUEUE} = < t', k, Next(l) >_{QUEUE}$$

Remarquons que les axiomes généralisés peuvent être utiles pour spécifier la représentation de l'égalité. Par exemple, si les axiomes généralisés de *QUEUE* (**Gen-Ax**$_{QUEUE}$) amalgament toutes les files "*UNDERFLOW*" sur une constante *CRASH* :

$$X \in UNDERFLOW \implies X = CRASH ,$$

les axiomes généralisés d'implémentation des opérations (**Gen-Ax$_{OP}$**) traduisent ceci en :

$$\bar{X} \in UNDERFLOW \implies \bar{X} = \overline{CRASH} .$$

Dès lors, pour spécifier la représentation de l'égalité, on peut utiliser l'axiome généralisé de **Gen-Ax$_{EQ}$** suivant :

$$\bar{X} \in UNDERFLOW \wedge \bar{Y} \in UNDERFLOW \implies \bar{X} = \bar{Y}$$

En ce point, nous avons défini textuellement une implémentation abstraite pour le formalisme des exception-algèbres. Dans le chapitre suivant, nous allons décrire les présentations qui lui sont associées.

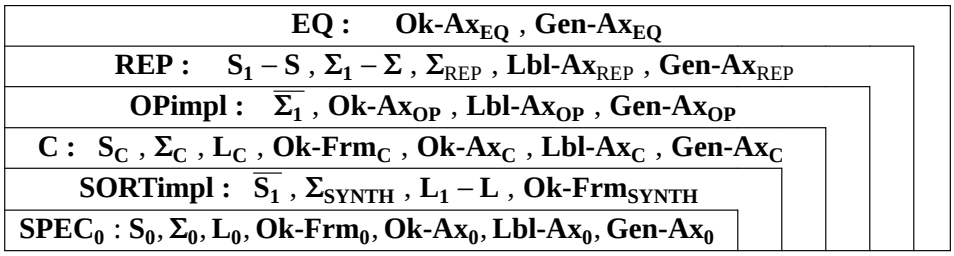
Chapitre III :

Le niveau

des présentations

Nous généralisons comme suit le “niveau des présentations” d’une implémentation abstraite que nous avions décrit dans la partie A, chap. II, section 3 (rappelée dans le chapitre I de cette partie) :

Figure 3 : *Syntaxe d’une implémentation avec traitement d’exceptions*



Comme dans le cas sans traitement d’exceptions, **SORTimpl** est une présentation au-dessus de $\mathbf{SPEC_0}$; **C** est une présentation au-dessus de $\mathbf{SPEC_0}$ + **SORTimpl** ; ... etc ... ; et la description intuitive en est encore la même :

- **SORTimpl** est la composante de *synthèse* (produits libres ou copies) ; elle synthétise les sortes constructives ($\overline{\mathbf{S_1}}$) au moyen des opérations de synthèse ($\mathbf{\Sigma_{SYNTH}}$). De plus, les formes *Ok* de ces sortes constructives sont caractérisées grâce aux déclarations de formes *Ok* (**Ok-Frm$_{SYNTH}$**). On lui adjoint les étiquettes d’exception à implémenter ($\mathbf{L_1 - L}$) afin de compléter la signature ($\overline{\mathbf{S_1}}, \mathbf{\Sigma_{SYNTH}}$) en l’exception-signature ($\overline{\mathbf{S_1}}, \mathbf{\Sigma_{SYNTH}}, \mathbf{L_1}$), qui pourra alors être utilisée dans les composantes ultérieures.

- **C** est la *composante cachée* ; elle enrichit **SORTimpl** pour faciliter la spécification de l’implémentation abstraite.

- **OPimpl** est la composante d’*implémentation des opérations* (*et des étiquettes d’exception*) ; elle permet de spécifier l’implémentation des opérations constructives ($\overline{op} \in \overline{\mathbf{\Sigma_1}}$) et des étiquettes d’exception à implémenter ($\mathbf{L_1}$).

- **REP** est la composante de *représentation* ; elle spécifie la correspondance entre la signature à implémenter ($<\mathbf{S_1}, \mathbf{\Sigma_1}, \mathbf{L_1}>$), et la signature constructive qui la représente ($<\overline{\mathbf{S_1}}, \mathbf{\Sigma_{SYNTH}}, \mathbf{L_1}>$). Les ensembles $\mathbf{\Sigma}_{\text{REP}}$, $\mathbf{Lbl\text{-}Ax}_{\text{REP}}$ et $\mathbf{Gen\text{-}Ax}_{\text{REP}}$ sont définis plus loin.

- Enfin **EQ** spécifie la *représentation de l'égalité* au moyen des *Ok*-axiomes et des axiomes généralisés de représentation de l'égalité ($\mathbf{Ok\text{-}Ax_{EQ}}$ et $\mathbf{Gen\text{-}Ax_{EQ}}$).

Par abus de langage, pour ne pas alourdir les notations, il nous arrivera souvent par la suite de noter "**EQ**" au lieu de **SPEC$_0$+SORTimpl+C+OPimpl+REP+EQ** ; idem pour "**REP"**, "**OPimpl**" ... etc. Le contexte permettra de différencier s'il s'agit de la *présentation* **EQ** seule, ou de la *spécification* complète qu'elle domine.

Les diverses composantes des présentations **SORTimpl**, **C**, **OPimpl** et **EQ** ont déjà été définies dans le chapitre précédent. Il ne nous reste donc qu'à décrire $\mathbf{\Sigma}_{\text{REP}}$, $\mathbf{Lbl\text{-}Ax}_{\text{REP}}$ et $\mathbf{Gen\text{-}Ax}_{\text{REP}}$.

Comme dans le cas sans traitement d'exceptions, les opérations et les axiomes de représentation sont tous automatiquement déduits de l'isomorphisme de signatures ρ.

- $\mathbf{\Sigma}_{\text{REP}}$ est l'ensemble des *opérations de représentation*. Pour chaque sorte à implémenter, $s \in \mathbf{S_1}$, $\mathbf{\Sigma}_{\text{REP}}$ contient une (et une seule) opération de représentation dont l'arité est :
$$\overline{\rho_s} : \; s \;\rightarrow\; \bar{s} \qquad \text{où } \bar{s} = \rho(s) \; .$$

- $\mathbf{Lbl\text{-}Ax}_{\text{REP}}$ est l'ensemble des *axiomes d'étiquetage associé à la représentation*. Il traduit syntaxiquement le fait que si un terme à implémenter est représenté par un terme étiqueté, alors, après que l'implémentation soit faite, il doit lui aussi être étiqueté (par la même étiquette).
Par conséquent, pour chaque sorte à implémenter, $s \in \mathbf{S_1}$, et pour chaque étiquette $l \in \mathbf{L} \cup \{Ok\}$, $\mathbf{Lbl\text{-}Ax}_{\text{REP}}$ contient l'axiome suivant :
$$\overline{\rho_s}(x) \in l \quad \Longrightarrow \quad x \in l \; .$$
Remarquons que ces axiomes concernent non seulement les étiquettes d'exception, mais aussi l'étiquette "*Ok*". En effet, il ne suffit pas de "remonter" les étiquetages d'exception, il faut aussi spécifier que : si un terme à implémenter est représenté par une valeur constructive *Ok*, alors il doit être lui-même *Ok* après que l'implémentation soit faite.

- $\mathbf{Gen\text{-}Ax}_{\text{REP}}$ est l'ensemble des *axiomes généralisés de représentation*. Il traduit syntaxiquement le fait que la représentation associe son implémentation constructive ($\overline{op}$) à chaque opération (op) à implémenter. Ceci signifie que pour chaque opération ($op \in \mathbf{\Sigma_1}$), $\mathbf{Gen\text{-}Ax}_{\text{REP}}$ contient l'axiome :
$$\overline{\rho_s}(op(x_1, \ldots, x_n)) \;=\; \rho(op)(\,\overline{\rho_{s_1}}(x_1), \ldots, \overline{\rho_{s_n}}(x_n)\,) \;\; (^2)$$
où s est la sorte cible de op, et s_i est la sorte de x_i .
Il faut encore spécifier que $\overline{\rho_s}$ et $< \cdots >_s$ ne sont que des opérations de *copies* lorsque s est une sorte prédéfinie. Ce qui s'écrit :
$$< x >_s \;=\; \overline{\rho_s}(x)$$
pour chaque sorte prédéfinie $s \in \mathbf{S}$.
De plus, il faut traduire le fait que si deux termes à implémenter possèdent la même représentation, alors ils apparaissent comme égaux après que l'implémentation soit faite. Par conséquent, pour chaque sorte à implémenter, $s \in \mathbf{S_1}$, $\mathbf{Gen\text{-}Ax}_{\text{REP}}$ contient l'axiome supplémentaire suivant :

---

(2) rappelons que $\rho(op)$ est noté $\overline{op}$

$$\overline{\rho_s}(x) = \overline{\rho_s}(y) \quad \Rightarrow \quad x = y \ .$$

Ces axiomes permettent de “remonter” vers les sortes à implémenter ($\mathbf{S_1}$) les identifications faites lors de l’implémentation dans les sortes constructives ($\overline{\mathbf{S_1}}$).

Remarquons que nous n’avons utilisé que des axiomes généralisés pour spécifier la représentation. En voici la raison : le but des axiomes associés à la représentation est de traduire l’isomorphisme de signatures ρ décrit dans le chapitre précédent. Or cet isomorphisme de signatures s’applique dans tous les cas d’application des opérations constructives, aussi bien lorsqu’elles retournent un résultat *Ok* que lorsqu’elles retournent un résultat exceptionnel. Il fallait donc utiliser des axiomes généralisés plutôt que des *Ok*-axiomes.

Voici comment sont spécifiés $\mathbf{\Sigma}_{\text{REP}}$, $\mathbf{Lbl\text{-}Ax}_{\text{REP}}$ et $\mathbf{Gen\text{-}Ax}_{\text{REP}}$ pour notre exemple d’implémentation des files bornées par les tableaux bornés :

Exemple 7 :
Pour l’implémentation des files bornées par les tableaux bornés et les entiers naturels, on a :

□ $\mathbf{\Sigma}_{\text{REP}}$ contient les opérations

$$\begin{array}{rrcl} \overline{\rho_{QUEUE}}: & QUEUE & \rightarrow & \overline{QUEUE} \\ \overline{\rho_{NAT}}: & NAT & \rightarrow & \overline{NAT} \\ \overline{\rho_{ELEM}}: & ELEM & \rightarrow & \overline{ELEM} \end{array}$$

□ $\mathbf{Lbl\text{-}Ax}_{\text{REP}}$ contient les axiomes

$$\begin{array}{rcl} \overline{\rho_{QUEUE}}(X) \in UNDERFLOW & \Rightarrow & X \in UNDERFLOW \\ \overline{\rho_{QUEUE}}(X) \in OVERFLOW & \Rightarrow & X \in OVERFLOW \\ \overline{\rho_{QUEUE}}(X) \in QUEUE\text{-}IS\text{-}EMPTY & \Rightarrow & X \in QUEUE\text{-}IS\text{-}EMPTY \end{array}$$

...etc pour chaque étiquette de $\mathbf{L_1}$
et chaque sorte *QUEUE, NAT, ELEM*

On constate que des axiomes tels que

$$\overline{\rho_{NAT}}(n) \in OVERFLOW \quad \Rightarrow \quad n \in OVERFLOW$$

doivent aussi être spécifiés, bien que l’étiquette *OVERFLOW* ne concerne *a priori* que les objets de sorte *QUEUE*. De fait, au moins dans l’algèbre initiale aucun entier ne sera étiqueté par *OVERFLOW*. Cependant, pour ne pas perdre le fait que $\mathbf{Lbl\text{-}Ax}_{\text{REP}}$ est automatiquement déduit de ρ, on doit, en toute rigueur, les spécifier. De plus, il existe des algèbres non initiales pour lesquelles des occurrences de ces axiomes s’appliquent ; spécifier ces axiomes nous permet donc de définir une sémantique fonctorielle de l’implémentation (et pas seulement une sémantique fondée sur les objets initiaux).

□ $\mathbf{Gen\text{-}Ax}_{\text{REP}}$ est spécifié par

$$
\begin{array}{rcl}
\overline{\rho_{QUEUE}}(empty) & = & \overline{empty} \\
\overline{\rho_{QUEUE}}(add(x, X)) & = & \overline{add}(\overline{\rho_{NAT}}(x), \overline{\rho_{QUEUE}}(X)) \\
\overline{\rho_{QUEUE}}(remove(X)) & = & \overline{remove}(\overline{\rho_{QUEUE}}(X)) \\
\overline{\rho_{ELEM}}(first(X)) & = & \overline{first}(\overline{\rho_{QUEUE}}(X)) \\
\overline{\rho_{NAT}}(length(X)) & = & \overline{length}(\overline{\rho_{QUEUE}}(X))
\end{array}
$$

... etc ...

$$
\begin{array}{c}
\overline{\rho_{QUEUE}}(X) = \overline{\rho_{QUEUE}}(Y) \quad \Longrightarrow \quad X = Y \\
\overline{\rho_{NAT}}(m) = \overline{\rho_{NAT}}(n) \quad \Longrightarrow \quad m = n \\
\overline{\rho_{ELEM}}(e) = \overline{\rho_{ELEM}}(e') \quad \Longrightarrow \quad e = e' \\
\overline{\rho_{NAT}}(n) \quad = \quad < n >_{NAT} \\
\overline{\rho_{ELEM}}(e) \quad = \quad < e >_{ELEM}
\end{array}
$$

Soulignons une fois encore que $\mathbf{\Sigma}_{\text{REP}}$, $\mathbf{Lbl\text{-}Ax}_{\text{REP}}$ et $\mathbf{Gen\text{-}Ax}_{\text{REP}}$ sont entièrement automatiquement déduits de l'isomorphisme de signatures ρ ; et qu'ils ne nécessitent nullement l'intervention du concepteur pour être spécifiés. La composante **REP** n'est qu'un "détour" pour modéliser algébriquement le fait qu'un utilisateur de l'implémentation n'a le droit d'utiliser que les opérations de $\mathbf{SPEC_1}$ ($\mathbf{\Sigma_1}$), et ne peut pas utiliser directement les opérations propres à l'implémentation (synthèse, opérations cachées ...).

Chapitre IV :

Le niveau

Sémantique

Nous avons fourni une spécification de l'implémentation abstraite avec traitement d'exceptions qui reprend exactement les mêmes composantes que dans le cas sans traitement d'exceptions, la sémantique sera donc définie de la même façon que dans la partie A de cette thèse, avec les mêmes explications intuitives que nous ne reprendrons donc pas ici.

$$\text{Alg}(\mathbf{SPEC_0}) \quad -F_{\mathbf{EQ}}\rightarrow \quad \text{Alg}(\mathbf{EQ}) \quad -U_{<\mathbf{S_1},\mathbf{\Sigma_1},\mathbf{L_1}>}\rightarrow \quad \text{Alg}(<\mathbf{S_1},\mathbf{\Sigma_1},\mathbf{L_1}>)$$

où $F_{\mathbf{EQ}}$ est le foncteur de synthèse associé à la présentation (**SORTimpl**+**C**+**OPimpl**+**REP**+**EQ**) au-dessus de $\mathbf{SPEC_0}$, et $U_{<\mathbf{S_1},\mathbf{\Sigma_1},\mathbf{L_1}>}$ est le foncteur d'oubli sur l'exception-signature de la spécification à implémenter.

Nous noterons "*sémantique*" le foncteur composé :

$$\textit{sémantique} \quad = \quad U_{\mathbf{\Sigma}\text{-}exc_1} \;\; o \;\; F_{\mathbf{EQ}}$$

On constate ici combien est cruciale l'existence des foncteurs de synthèse pour les exception-algèbres. C'est elle qui nous permet de définir une sémantique simple de l'implémentation abstraite en présence d'exceptions, et par là même de ramener les critères de correction à des notions connues des types abstraits algébriques telles que la suffisante complétude ou la consistance hiérarchique.

Nous n'avons pas cité l'exception-algèbre

$$\text{SEM}_{\mathbf{IMPL}} = U_{<\mathbf{S_1},\mathbf{\Sigma_1},\mathbf{L_1}>}(F_{\mathbf{EQ}}(T_{\mathbf{SPEC_0}})) = \textit{sémantique}(T_{\mathbf{SPEC_0}})$$

car elle ne concerne ici que le traitement de l'algèbre initiale $T_{\mathbf{SPEC_0}}$; or nous avons remarqué qu'avec traitement d'exceptions, c'est une classe d'algèbres légèrement plus étendue qui nous intéresse : la classe des exception-algèbres *post-initiales*.
Rappelons que l'usage des exception-algèbres post-initiales répond à la préoccupation suivante : bien que l'implémentation abstraite décrite n'utilise que $\mathbf{SPEC_0}$ (par exemple *ARRAY+NAT+ELEM*) ceci ne préjuge pas du "contexte" dans lequel est effectuée l'implémentation. En effet, il est fort possible qu'il existe par ailleurs d'autres présentations au-dessus de $\mathbf{SPEC_0}$ qui n'interviennent pas directement dans l'implémentation considérée. Par exemple si l'on dispose d'une structure résidante de piles d'entiers, *STACK(NAT)*, en plus de *ARRAY*, cette structure n'intervient pas dans notre implémentation de *QUEUE* ; néanmoins on sait que *STACK(NAT)* ajoute de nouvelles valeurs erronées dans *NAT*. Il

en résulte que l'algèbre initiale $T_{\mathbf{SPEC_0}}$ ne prend pas en compte ce "contexte" ; il faut faire appel aux algèbres post-initiales (cf. Partie B, chapitre VIII). Nous devrons donc définir la correction d'une implémentation non seulement sur $T_{\mathbf{SPEC_0}}$, mais aussi relativement à toutes $\mathbf{SPEC_0}$-algèbres post-initiales.

Remarque 3 :
Les axiomes de $\mathbf{Lbl\text{-}Ax}_{\mathrm{REP}}$ et $\mathbf{Gen\text{-}Ax}_{\mathrm{REP}}$ de la forme

$$\overline{\rho_s}(x) \in l \quad \Longrightarrow \quad x \in l$$

ou

$$\overline{\rho_s}(x) = \overline{\rho_s}(y) \quad \Longrightarrow \quad x = y$$

jouent un rôle important dans la sémantique de l'implémentation abstraite. En effet, bien que la représentation aille dans le sens $\mathbf{\Sigma_1}\rightarrow$*implémentation*, ces axiomes permettent de *remonter* vers les sortes à implémenter les étiquetages et les identifications faites lors de l'étape d'implémentation. Il s'agit aussi bien des identifications issues des termes à implémenter qui possèdent la même représentation, que des identifications issues de la représentation de l'égalité ($\mathbf{Ok\text{-}Ax_{EQ}}$ et $\mathbf{Gen\text{-}Ax_{EQ}}$).

Nous allons maintenant nous appuyer sur la sémantique que nous venons de définir pour établir les critères de correction d'une implémentation.

Chapitre V :

Preuves de correction

Dans le cas sans traitement d'exceptions, deux conditions sont requises pour qu'une implémentation abstraite soit correcte :

- □ tout terme fermé à implémenter doit posséder une représentation parmi les n-uplets synthétisés par les opérations de synthèse
- □ la "vision utilisateur" de l'implémentation doit être isomorphe à la structure décrite par la spécification descriptive à implémenter $\mathbf{SPEC_1}$.

Dans le cas avec traitement d'exceptions, comme nous l'avons déjà remarqué, ces deux conditions ne doivent pas concerner uniquement $T_{\mathbf{SPEC_0}}$ et *sémantique*($T_{\mathbf{SPEC_0}}$), mais toutes les $\mathbf{SPEC_0}$-algèbres post-initiales A , et *sémantique(*A*)*. D'autre part, il est clair que les termes *erronés* ne doivent pas nécessairement posséder une représentation sous forme de n-uplets synthétisés. Par exemple il n'est pas nécessaire d'associer un triplet $< t, i, j >_{QUEUE}$ à un terme erroné tel que *add(x,"une-file-pleine")* ; il suffit que l'implémentation lui associe l'étiquette *OVERFLOW* (rendant ainsi ce terme erroné).

Il en résulte qu'avec traitement d'exceptions, ces deux conditions s'ecrivent :

- □ Quelle que soit la $\mathbf{SPEC_0}$-algèbre post-initiale A , tout $\mathbf{\Sigma_1}$-terme fermé *non erroné* doit posséder une représentation parmi les n-uplets synthétisés par les opérations de synthèse.
- □ Pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A , *sémantique(*A*)* doit être une $\mathbf{SPEC_1}$-algèbre post-initiale.

Comme dans le cas sans traitement d'exceptions, pour étudier précisémant la correction d'une implémentation, nous scindons ces deux conditions en quatre critères, en éclatant la seconde condition en trois parties :

1) Le fait que tout terme fermé non erroné possède une représentation est l'*opération complétude*.

2) Pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A , la partie *Ok* de *sémantique(*A*)* doit être $\mathbf{\Sigma_1}$-finiment générée. Ce critère est la *protection des données Ok à implémenter*.

3) Pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A , *sémantique(*A*)* doit être une $\mathbf{SPEC_1}$-algèbre ; c'est-à-dire qu'elle doit valider $\mathbf{SPEC_1}$. Ce critère est la *validité* de l'implémentation.

4) Enfin, parmi les $\mathbf{SPEC_1}$-algèbres dont la partie *Ok* est finiment générée, *sémantique(*A*)* doit être post-initiale, c'est-à-dire que sa partie finiment générée doit être initiale. Ce critère est la *consistance* de l'implémentation.

Cet éclatement nous permet d'étudier des conditions minimales pour qu'une implémentation soit correcte (sections 1 à 4). Néanmoins la *correction forte* (section 5) n'utilisera pas cet éclatement en quatre critères ; elle fournira une condition *suffisante* pour assurer la correction. La correction forte est très utile puisqu'elle permet d'assurer une *réutilisation* correcte des implémentations, fait que n'assure pas la correction.

1. L'OPERATION-COMPLETUDE

L'introduction de traitement d'exceptions modifie légèrement la définition d'opération-complétude que nous avions donnée en partie A, chapitre II, section 5.1, définition 5. En effet, pour que toutes les opérations à implémenter soient complètement représentées, il faut soit qu'elles retournent un résultat *Ok* possédant une représentation de la forme $< \cdots >_s$, soit que l'implémentation soit apte à déterminer que le résultat est erroné. Inductivement, ceci est traduit sur les termes fermés de la manière suivante :

Définition 2 :
L'implémentation abstraite **IMPL** est *op-complète* si et seulement si, pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A , la représentation de tout $\mathbf{\Sigma_1}$-terme fermé non erroné dans $F_{\mathbf{REP}}(\mathrm{A})$ possède une valeur parmi les objets synthétisés par $\mathbf{\Sigma_{SYNTH}}$. Ce qui s'écrit, en notant $F_{\mathbf{REP}}(\mathrm{A}) = \mathrm{A}_{\mathbf{REP}}$ et $F_{\mathbf{SORTimpl}}(\mathrm{A}) = \mathrm{A}_{\mathbf{SORTimpl}}$:

$$\forall\, t \in T_{\mathbf{\Sigma_1}}\,,\ [\, t \in (\mathrm{A}_{\mathbf{REP}} - A_{\mathbf{REP},err}) \ \Rightarrow\ (\exists\, \alpha \in \mathrm{A}_{\mathbf{SORTimpl}} \ \textit{tel que}\ \overline{\rho_s}(t) = \alpha \ \textit{dans}\ \mathrm{A}_{\mathbf{REP}})\,]$$

(où *s* est la sorte de *t*)

Remarquons que si A est la $\mathbf{SPEC_0}$-algèbre initiale $T_{\mathbf{SPEC_0}}$, alors $\mathrm{A}_{\mathbf{REP}}$ est égale à $T_{\mathbf{REP}}$ et $\mathrm{A}_{\mathbf{SORTimpl}}$ est égale à $T_{\mathbf{SORTimpl}}$; si bien que cette définition n'est autre qu'une généralisation de l'opération-complétude déjà définie dans la cas sans traitement d'exceptions.

Pour les mêmes raisons que dans le cas sans traitement d'exceptions, on ne veut en aucune façon que la représentation de l'égalité soit prise en compte lorsque l'on vérifie l'op-complétude. L'introduction de traitement d'exceptions ne remet pas en cause cet argument. Car, même si la représentation de l'égalité permet de rendre erroné un terme qui ne l'était pas dans $\mathrm{A}_{\mathbf{REP}}$ (en l'amalgamant avec une valeur déjà erronée), il faut que l'implémentation puisse déterminer que ce terme est erroné sans l'aide de la représentation de l'égalité ; sinon, ce terme serait incomplètement implémenté.

Remarque 4 :
Pour que l'implémentation soit op-complète, *il suffit de vérifier l'op-complétude pour l'exception-algèbre initiale* $T_{\mathbf{SPEC_0}}$.

Ceci résulte simplement du fait que pour toute algèbre post-initiale A, il existe un morphisme (issu du morphisme initial) de $T_{\mathbf{REP}}$ dans $\mathrm{A}_{\mathbf{REP}}$:

$$F_{\mathbf{REP}}(init_{\mathrm{A}}) :\ T_{\mathbf{REP}} \ \rightarrow\ \mathrm{A}_{\mathbf{REP}}$$

par conséquent, étant donné un $\mathbf{\Sigma_1}$-terme fermé quelconque *t*, si *t* est dans la classe d'un **SORTimpl**-terme α modulo $T_{\mathbf{REP}}$, alors *a fortiori* il est dans la classe de α modulo $\mathrm{A}_{\mathbf{REP}}$.

Intuitivement, rappelant que l'usage des algèbres post-initiales permet de prendre en compte le "contexte" dans lequel l'implémentation est effectuée ([3]), ceci signifie que l'op-complétude est une propriété indépendante de ce "contexte".

Grâce à cette remarque importante, nous pouvons énoncer le théorème suivant, qui montre que, exactement comme dans le cas sans traitement d'exceptions, on peut vérifier l'op-complétude directement dans **OPimpl** plutôt que dans **REP**.

Théorème 1 :
Pour que **IMPL** soit op-complète, il faut et il suffit que pour tout $\overline{\mathbf{\Sigma_1}}$-terme fermé $t' \in T_{\overline{\Sigma_1}}$, si t' n'est pas erroné dans $T_{\mathbf{OPimpl}}$ alors il existe un élément α de $T_{\mathbf{SORTimpl}}$ tel que $t' = \alpha$ dans $T_{\mathbf{OPimpl}}$.

Preuve :
Soit $\bar{\rho}$ l'isomorphisme entre $T_{\mathbf{\Sigma_1}}$ et $T_{\overline{\Sigma_1}}$ déduit de l'isomorphisme de signatures ρ entre $<\mathbf{S_1}, \mathbf{\Sigma_1}>$ et $<\overline{\mathbf{S_1}}, \overline{\mathbf{\Sigma_1}}>$. D'après les axiomes de **REP** (**Lbl-Ax**$_{\text{REP}}$ et **Gen-Ax**$_{\text{REP}}$), il résulte que pour tout $\mathbf{\Sigma_1}$-terme fermé t, $\overline{\rho_s}(t)$ est dans la même classe d'équivalence, modulo **REP**, que le terme $t' = \bar{\rho}(t)$, et t' est erroné dans $T_{\mathbf{OPimpl}}$ si et seulement si t l'est dans $T_{\mathbf{REP}}$.
Notre condition nécessaire et suffisante résulte donc du fait que $\bar{\rho}$ est un isomorphisme entre $T_{\mathbf{\Sigma_1}}$ et $T_{\overline{\Sigma_1}}$. □

Exemple 8 :
On prouve l'op-complétude de notre implémentation des files bornées grâce à une méthode d'induction structurelle.
Du fait de la définition des valeurs erronées d'une exception-algèbre (partie B, chap. IV, section 2, définition 8), nous devons démontrer la propriété suivante :
pour toute opération de $\overline{\mathbf{\Sigma_1}}$ (i.e. toute opération $\overline{op}$), si $t_1 \cdots t_n$ sont des $\overline{\mathbf{\Sigma_1}}$-termes non erronés dans $T_{\mathbf{OPimpl}}$, alors : ou $\overline{op}(t_1, \ldots, t_n)$ est égal à une valeur de la forme $< r_1, \ldots, r_m >_s$, ou il existe une étiquette d'exception $l \in \mathbf{L_1}$ telle que $\overline{op}(t_1, \ldots, t_n) \in T_{\mathbf{OPimpl},l}$.

- □ L'opération $\overline{empty}$ est toujours égale à $< t, i, i >_{QUEUE}$.
- □ Si x est de la forme $< e >_{ELEM}$ et X est de la forme $< t, i, j >_{QUEUE}$, alors deux cas sont possibles :
  - soit *Next(j)* est égal à i ; auquel cas $\overline{add}(x, X) = \overline{add}(< e >_{ELEM}, < t, i, j >_{QUEUE})$ répond à l'étiquette d'exception *OVERFLOW*
  - soit *Next(j)* n'est pas égal à i ; auquel cas $\overline{add}(x, X) = \overline{add}(< e >_{ELEM}, < t, i, j >_{QUEUE})$ est égal à $< t[j] := e, i, Next(j) >_{QUEUE}$.
- □ Le même type de raisonnement vaut pour les opérations *remove*, *first* et *length*.

Nous retrouvons de plus la proposition suivante :

Proposition 1 :
Pour que l'implémentation abstraite **IMPL** soit op-complète, il *suffit* que la spécification **OPimpl** soit suffisamment complète au-dessus de **SORTimpl**.

---

(3) En d'autres termes : même si l'implémentation n'utilise que la structure résidente spécifiée par $\mathbf{SPEC_0}$, d'autres structures peuvent être présentes lorsque l'implémentation est faite.

Preuve :
Supposons que **OPimpl** soit suffisamment complète au-dessus de **SORTimpl**. La suffisante complétude (partie B, chap. VII, section 2, définition 23) nous assure que tout **Σ**(**OPimpl**)-terme fermé de sorte dans $\overline{\mathbf{S_1}}$ est dans la classe d'un **Σ**(**SORTimpl**)-terme, ou est erroné. La conclusion résulte donc du fait que **Σ**(**OPimpl**) contient la signature $\overline{\mathbf{\Sigma_1}}$, et du théorème 2. □

2. PROTECTION DES DONNEES *Ok* A IMPLEMENTER

Définition 3 :
L'implémentation abstraite **IMPL** *protège les données Ok à implémenter* si et seulement si, pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A , la partie *Ok* de *sémantique(*A*)* est $\mathbf{\Sigma_1}$-finiment générée.

Exemple 9 :
Pour notre exemple d'implémentation des files bornées, les opérations $< \_ >_{ELEM}$ et $< \_ >_{NAT}$ ne sont que des opérations de copies, y compris pour les étiquetages. Il en résulte que la protection des données *Ok* prédéfinies est immédiate à ce niveau.
En ce qui concerne la sorte d'intérêt *QUEUE*, rappelons qu'un triplet $< t, i, j >_{QUEUE}$ de $\overline{QUEUE}$ est *Ok* si et seulement si chacune des valeurs *t, i* et *j* le sont. En particulier, *i, j* et toutes les valeurs mémorisées dans le tableau *t* sont *Ok*, donc finiment générées par hypothèse. De plus, par construction de **REP**, un élément *X* de sorte *QUEUE* ne peut être *Ok* que si sa représentation $\overline{\rho_{QUEUE}}(X)$ est une valeur *Ok* de $\overline{QUEUE}$. Il en résulte que les valeurs *Ok* de sorte *QUEUE* sont finiment générées puisque de la forme $add(t[r_m], \ldots, add(t[r_0], new)\ldots)$; où $r_0 = i$, $r_{n+1} = Next(r_n)$, jusqu'à r_m tel que $Next(r_m) = j$.

Le théorème suivant prouve que la protection des données *Ok* à implémenter peut être obtenue via un critère de *complétude suffisante forte*, exactement comme la "protection des données prédéfinies" était obtenue via la suffisante complétude dans le cas sans traitement d'exceptions.

Théorème 2 :
Une condition suffisante pour que **IMPL** protège les données *Ok* à implémenter est que la spécification de l'implémentation (**SORTimpl**+...+**EQ**) soit fortement complète au-dessus de $\mathbf{\Sigma}$-exc_1 (signature de $\mathbf{SPEC_1}$).

Preuve :
La complétude forte de **EQ** implique que $U_{\mathbf{\Sigma}\text{-}exc_1}(F_{\mathbf{EQ}}(B))_{Ok}$ est $\mathbf{\Sigma_1}$-finiment généré pour toute $\mathbf{\Sigma}$-exc_1-algèbre post-initiale B . Nous voulons démontrer cette propriété pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A. La conclusion résulte alors simplement du fait que pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A, B=$F_{\mathbf{\Sigma}\text{-}exc_1-\mathbf{\Sigma}\text{-}exc}(A)$ est une $\mathbf{\Sigma}$-exc_1-algèbre post-initiale (les foncteurs libres préservent les algèbres post-initiales), et $F_{\mathbf{EQ}}(A) = F_{\mathbf{EQ}}(F_{\mathbf{\Sigma}\text{-}exc_1-\mathbf{\Sigma}\text{-}exc}(A))$. □

La complétude suffisante forte de **EQ** n'est qu'une condition suffisante, néanmoins elle est satisfaite pour tous les exemples usuels.

3. LA VALIDITE

Définition 4 :
L'implémentation abstraite **IMPL** est dite *valide* si et seulement si pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A, *sémantique(*A*)* valide $\mathbf{SPEC_1}$.

Ceci signifie que *sémantique(*A*)* doit valider **Ok-Frm$_1$**, **Ok-Ax$_1$**, **Lbl-Ax$_1$** et **Gen-Ax$_1$**. Rappelons que, de même que la spécification d'une implémentation sans traitement d'exception ne contenait pas les axiomes descriptifs (**A$_1$**), la spécification d'une implémentation avec traitement d'exceptions ne contient pas **Ok-Frm$_1$**, **Ok-Ax$_1$**, **Lbl-Ax$_1$** et **Gen-Ax$_1$**.

Notation 1 :
On note **IDimpl** l'exception-spécification suivante :

SPEC$_0$+SORTimpl+C+OPimpl+REP+EQ + $\downarrow$
< Ok-Frm$_1$ – Ok-Frm , Ok-Ax$_1$ – Ok-Ax , Lbl-Ax$_1$ – Lbl-Ax , Gen-Ax$_1$ – Gen – Ax >

Ce qui revient à dire que **IDimpl** est égale à
SPEC$_1$+SORTimpl+C+OPimpl+REP+EQ .

Ceci signifie que **IDimpl** contient absolument tous les éléments de toutes les spécifications invoquées dans notre formalisme. **IDimpl** ajoute à la spécification de l'implémentation abstraite toutes les parties de la spécification descriptive à implémenter qui n'y sont pas déjà incluses.

Le théorème suivant montre que la condition de validité peut toujours se ramener à des méthodes de preuves de théorème. Il généralise le théorème 4 (partie A, chap. II, section 5.3) au cas avec traitement d'exceptions.

Théorème 3 :
Etant donnée une implémentation abstraite avec traitement d'exceptions **IMPL**, les quatre conditions suivantes sont équivalentes :

□ (1) **IMPL** est valide

□ (2) pour toute **SPEC$_0$**-algèbre post-initiale A, *sémantique(*A*)* valide
<**Ok-Frm$_1$**, **Ok-Ax$_1$ – Ok-Ax**, **Lbl-Ax$_1$ – Lbl-Ax**, **Gen-Ax$_1$ – Gen-Ax**>

□ (3) pour toute **SPEC$_0$**-algèbre post-initiale A, $F_{\mathbf{EQ}}$(A) valide
<**Ok-Frm$_1$**, **Ok-Ax$_1$ – Ok-Ax**, **Lbl-Ax$_1$ – Lbl-Ax**, **Gen-Ax$_1$ – Gen-Ax**>

□ (4) pour toute **SPEC$_0$**-algèbre post-initiale A, le morphisme d'adjonction
$$F_{\mathbf{EQ}}(A) \quad \rightarrow \quad F_{\mathbf{IDimpl}}(A)$$
est partiellement rétractable (mieux : c'est un isomorphisme).

Preuve :
[1⇔2] la validité de **IMPL** signifie que *sémantique(*A*)* valide **SPEC$_1$**. Mais, puisque la spécification de l'implémentation abstraite contient **SPEC$_0$**, donc **P**, *sémantique(*A*)* valide toujours **P** . Par conséquent, il suffit de vérifier que *sémantique(*A*)* valide **SPEC$_1$**–**P**.
[2⇔3] résulte directement du fait que les axiomes ou déclarations de **SPEC$_1$** ne concernent que l'exception-signature <**S$_1$**, **Σ$_1$**, **L$_1$**> (voir la remarque 5 ci-dessous) ; or *sémantique(*A*)* est justement l'oubli de $F_{\mathbf{EQ}}$(A) sur cette exception-signature.
[3⇔4] résulte directement de la définition de **IDimpl**.
Ceci clôt notre preuve. □

Remarque 5 :
Nous avons utillisé le fait que
< Ok-Frm$_1$ – Ok-Frm$_0$, Ok-Ax$_1$ – Ok-Ax$_0$, Lbl-Ax$_1$ – Lbl-Ax$_0$, Gen-Ax$_1$ – Gen-Ax$_0$ >
ne concerne que la signature **Σ**-*exc*$_1$=<**S$_1$**, **Σ$_1$**, **L$_1$**>. Remarquons que, bien que les étiquettes

d'exception ne soient pas typées (i.e. ne sont attachées à aucune sorte), les axiomes d'étiquetage de **Lbl-Ax$_1$** le sont, car pour tout axiome d'étiquetage

$$[\, t_1 \in l_1 \wedge \cdots \wedge t_n \in l_n \ \wedge\ v_1 = w_1 \wedge \cdots \wedge v_m = w_m \,] \quad \Rightarrow \quad t \in l$$

les termes avec variables t_i, v_j, w_j et t sont typés.
En particulier, les axiomes d'étiquetage de **Lbl-Ax$_1$** ne s'appliquent pas aux sortes intermédiaires de $\overline{\mathbf{S_1}}$; ce qui confirme notre argument.

Exemple 10 :
Soit à prouver que notre implémentation des files bornées par les tableaux bornés est valide.

D'abord, on peut prouver que si *Maxrange* est strictement inférieur à *Maxlength*, alors l'implémentation abstraite que nous avons décrite n'est pas valide.
En effet, $F_{\mathbf{EQ}}$(A) doit en autres valider les *Ok*-axiomes suivants de *QUEUE*

$$\mathit{first(add(x,empty))} \ = \ x$$
$$\mathit{remove(add(x,empty))} \ = \ \mathit{empty}$$
$$\mathit{remove(add(x,add(y,X)))} \ = \ \mathit{add(x,remove(x,add(y,X)))}$$

En particulier, il doit valider :

$$\mathit{first}(\mathit{remove}^{\mathit{Maxlength}-1}(\mathit{add}(x_1, \ldots \mathit{add}(x_{\mathit{Maxlength}}, \mathit{empty}) \ldots))) \ = \ x_1$$

Or, si *Maxrange* est strictement inférieur à *Maxlength*, le tableau, *t*, qui implémente $\mathit{add}(x_1, \ldots \mathit{add}(x_{\mathit{Maxlength}}, \mathit{empty}) \ldots)$ est erroné ; donc l'accès *t[i]* à ce tableau, qui devrait retourner x_1, retourne en fait une valeur erronée (de sorte *ELEM*, par propagation d'erreur). Il en résulte que si *Maxrange* est strictement inférieur à *Maxlength*, alors $F_{\mathbf{EQ}}$(A) ne valide pas *QUEUE*, donc l'implémentation abstraite n'est pas valide.
Le même raisonnement vaut si la borne supérieure de *NAT*, *Maxint*, est strictement inférieure à *Maxlength*.

Supposons donc maintenant que *Maxrange* et *Maxlength* soient supérieurs à *Maxlength*. On veut prouver que $F_{\mathbf{EQ}}$(A) valide *QUEUE* pour toute **SPEC$_0$**-algèbre post-initiale A .

- □ Validation de **Ok-Frm**$_{QUEUE}$:
  Les formes *Ok* de *QUEUE* sont les termes de la forme
  $$\mathit{add}(x_n, \ldots \mathit{add}(x_1, \mathit{empty}) \ldots) \qquad \text{avec } n \leq \mathit{Maxlength}.$$
  Or les axiomes d'implémentation imposent que ces termes sont implémentés par un triplet $< t, i, j >_{QUEUE}$ tel que *i* et *j* sont inférieurs à *Maxlength*, et *t* ne subit que des affectations dans les rangs compris entre *0* et *Maxlength*. Il en résulte que, puisque $\mathit{Maxint} \geq \mathit{Maxlength}$ et $\mathit{Maxrange} \geq \mathit{Maxlength}$, que *t*, *i* et *j* sont *Ok*. En particulier, $< t, i, j >_{QUEUE}$ est *Ok* ; et la validation de **Ok-Frm**$_{QUEUE}$ résulte donc de l'axiome de **REP** :
  $$\overline{\rho_{QUEUE}}(X) \in \mathit{Ok} \quad \Rightarrow \quad X \in \mathit{Ok}$$

- □ Validation de **Ok-Ax**$_{QUEUE}$:
  Prenons par exemple l'*Ok*-axiome
  $$\mathit{remove(add(x,add(y,X)))} \ = \ \mathit{add(x,remove(add(y,X)))} \ .$$
  Soit $< t, i, j >_{QUEUE}$ l'implémentation de *X*. Aussi bien pour la spécification *QUEUE* que pour **IMPL**, cet axiome (resp. l'*Ok*-implémentation de chacun de ces membres) ne s'applique que si la longueur de *X* est inférieure à *Maxlength-2*. Dans ce cas, on constate aisément que l'*Ok*-implémentation de chacun de ces deux membres donne :
  $$< (t[j] := y)[\mathit{Next}(j)] := x, \mathit{Next}(i), \mathit{Next}(\mathit{Next}(j)) >_{QUEUE}$$
  Ce qui prouve que cet *Ok*-axiome de *QUEUE* est validé par *IMPL*.
  Les autres *Ok*-axiomes de *QUEUE* se prouvent de la même façon.

□ Validation de **Lbl-Ax**$_{QUEUE}$:
Prenons par exemple l'axiome d'étiquetage de *QUEUE*

$$remove(empty) \in UNDERFLOW\ .$$

Les axiomes d'implémentation nous assurent d'abord que *empty* est représenté par $< t, i, i >_{QUEUE}$. Il suffit alors d'appliquer l'axiome d'implémentation des étiquettes de **IMPL** :

$$i = j \quad \Longrightarrow \quad \overline{remove}(< t, i, j >_{QUEUE}) \in UNDERFLOW$$

pour conclure grâce aux axiomes de représentation :

$$\overline{\rho_{QUEUE}}(remove(X)) \;=\; \overline{remove}(\overline{\rho_{QUEUE}}(X))$$
$$\overline{\rho_{QUEUE}}(X) \in UNDERFLOW \quad \Longrightarrow \quad X \in UNDERFLOW\ .$$

□ Validation de **Gen-Ax**$_{QUEUE}$:
Prenons l'axiome généralisé de *QUEUE*

$$length(X) = Maxlength \quad \Longrightarrow \quad add(x,X) = add(x,remove(X))$$

Soit $< e >_{ELEM}$ l'implémentation de *x*, et $< t, i, j >_{QUEUE}$ celle de *X*. On peut d'abord montrer que lorsque *length(X)=Maxlength*, *Next(j)* est égal à *i* (rappelons que *length(X)* est justement représenté par *(j–i)* modulo *succ(Maxlength)*).
Ensuite, en appliquant l'axiome généralisé d'implémentation :

$$Next(j) = i \;\Longrightarrow\; \overline{add}(< e >_{ELEM}\ , < t, i, j >_{QUEUE}) = < t[j] := e, Next(i), Next(j) >_{QUEUE}$$

On constate que le membre de gauche (*add(x,X)*) est implémenté par $< t[j] := e, Next(i), Next(j) >_{QUEUE}$.
Il suffit alors d'appliquer les *Ok*-axiomes d'implémentation pour constater que le membre de droite (*add(x,remove(X))*) est aussi implémenté par $< t[j] := e, Next(i), Next(j) >_{QUEUE}$.

On constate que l'on n'a pas utilisé les axiomes de représentation de l'égalité pour prouver que l'implémentation des files bornées est valide, mais ceci est particulier à notre exemple (en fait, les axiomes de représentation de l'égalité peuvent être ici déduits des axiomes d'implémentation des opérations). Comme nous l'avons déjà remarqué dans le cas sans traitement d'exceptions, il existe des exemples où la représentation de l'égalité est nécessaire (cf. l'implémentation des ensembles par les chaînes).

4. LA CONSISTANCE

Lorsque l'implémentation **IMPL** protège les données *Ok* à implémenter et est valide, la consistance vérifie si pour toute **SPEC$_0$**-algèbre post-initiale A, *sémantique(*A*)* (qui est alors une **SPEC$_1$**-algèbre dont la partie *Ok* est finiment générée) est une **SPEC$_1$**-algèbre *post-initiale*. D'une manière plus générale, sans supposer la protection des données *Ok* ni la validité, la consistance est définie comme suit.

Définition 5 :
L'implémentation abstraite **IMPL** est dite *consistante* si et seulement si pour toute **SPEC$_0$**-algèbre post-initiale A : deux $\mathbf{\Sigma_1}$-termes fermés ne sont égaux dans *sémantique*(A) que s'ils le sont déjà dans $T_{\mathbf{SPEC_1}}$, et un $\mathbf{\Sigma_1}$-terme fermé n'est étiqueté par une étiquette de $\mathbf{L_1} \cup \{Ok\}$ dans *sémantique*(A) que s'il l'est déjà dans $T_{\mathbf{SPEC_1}}$. Ce qui s'écrit :

$$\forall\ t \in T_{\mathbf{\Sigma_1}}\ ,\ \forall\ t' \in T_{\mathbf{\Sigma_1}}\ ,\ (\ t = t' \text{ dans } sémantique(\text{A}) \ \Longrightarrow\ t = t' \text{ dans } T_{\mathbf{SPEC_1}}\ )$$

et

$$\forall\ t \in T_{\mathbf{\Sigma_1}}\ ,\ \forall\ l \in \mathbf{L_1} \cup \{Ok\}\ ,\ (\ t \in sémantique(\text{A})_l \ \Longrightarrow\ t \in T_{\mathbf{SPEC_1},l}\ )$$

Le théorème suivant montre que la consistance de **IMPL** peut toujours être vérifiée dans la partie finiment générée de $F_{\mathbf{EQ}}(A)$. Ce théorème généralise le théorème 5 (partie A, chap. II, section 5.4) au

cas avec traitement d'exceptions.

Théorème 3 :
Si l'implémentation **IMPL** est valide alors les trois conditions suivantes sont équivalentes :

- □ (1) **IMPL** est consistante
- □ (2) Pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A, le morphisme initial de $T_{\mathbf{SPEC_1}}$ dans *sémantique*(A) est partiellement rétractable
- □ (3) Pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A, le morphisme initial de $T_{\mathbf{SPEC_1}}$ dans $U_{\mathbf{\Sigma}\text{-}exc_1}(F_{\mathbf{IDimpl}}(A))$ est partiellement rétractable

Preuve :
[1⇔2] résulte directement des définitions, et du fait que la validité implique que *sémantique*(A) est toujours une $\mathbf{SPEC_1}$-algèbre (ce morphisme initial existe donc).
[2⇔3] la validité de **IMPL** assure que $F_{\mathbf{IDimpl}}(A) = F_{\mathbf{EQ}}(A)$; donc $U_{\mathbf{\Sigma}\text{-}exc_1}(F_{IDimpl}(A)) =$ *sémantique*(A).
(nota : en fait, [1⇔3] ne suppose pas la validité de l'implémentation).
Ceci clôt la preuve du théorème. □

Comme dans le cas sans traitement d'exceptions, ce théorème démontre que pour vérifier la consistance d'une implémentation, il suffit de vérifier qu'aucun axiome (ou déclaration) de l'implémentation ne peut induire d'inconsistance dans les sortes à implémenter. Ces axiomes, ou déclarations, d'implémentation sont $\mathbf{Ok\text{-}Frm_{SYNTH}}$, ceux de la composante cachée **C**, $\mathbf{Ok\text{-}Ax_{OP}}$, $\mathbf{Lbl\text{-}Ax_{OP}}$, $\mathbf{Gen\text{-}Ax_{OP}}$, $\mathbf{Ok\text{-}Ax_{EQ}}$ et $\mathbf{Gen\text{-}Ax_{EQ}}$ (les axiomes de **REP** se contentent de traduire l'isomorphisme de signatures entre les opérations à implémenter et leur représentation).

Exemple 11 :
Pour prouver que notre implémentation des files bornées par les tableaux bornés est consistante, il nous faut d'abord prouver que si $\overline{\rho_{QUEUE}}(X) \in l$ (dans $T_{\mathbf{EQ}}$) alors $X \in l$ (dans T_{QUEUE}). Par exemple, pour l'étiquette *UNDERFLOW*, on a les axiomes

$$i = j \quad \Rightarrow \quad \overline{remove}(< t, i, j >_{QUEUE}) \in UNDERFLOW$$
$$\bar{X} \in UNDERFLOW \quad \Rightarrow \quad \overline{remove}(\bar{X}) \in UNDERFLOW$$

Le second axiome ne pose aucun problème puisqu'il correspond directement à l'axiome de *QUEUE*

$$X \in UNDERFLOW \quad \Rightarrow \quad remove(X) \in UNDERFLOW$$

Le premier axiome ne pose pas plus de problème, grâce au fait que nous connaissons la forme normale de la file représentée par $< t, i, j >_{QUEUE}$ lorsque $i=j$: c'est *empty*. Le premier axiome correspond donc à l'axiome de *QUEUE*

$$remove(empty) \in UNDERFLOW\ .$$

Le traitement de l'étiquette *Ok* est un peu plus délicat. Il nous faut prouver que toute file X dont la représentation est un triplet $< t, i, j >_{QUEUE}$ tel que t, i et j sont *Ok*, alors X est *Ok* dans T_{QUEUE}. Prouver cette propriété rigoureusement est long et fastidieux. En fait, elle est dû aux prémisses des *Ok*-axiomes d'implémentation des opérations : les prémisses ne permettent de donner une valeur de la forme $< t, i, j >_{QUEUE}$ que si les préconditions d'applications sont satisfaites (la longueur est strictement inférieur à *Maxlength* pour appliquer $\overline{add}$, la file n'est pas vide pour appliquer $\overline{remove}$ ou $\overline{first}$). Ainsi, seuls les termes possédant un forme *Ok* (pour *QUEUE*) possèdent une représentation *Ok* (pour **IMPL**).

Il nous faut maintenant traiter les *Ok*-axiomes ou les axiomes généralisés de l'implémentation. Pour ce faire, on peut d'abord prouver (par induction structurelle) que si $\overline{\rho_{QUEUE}}(X) = < t, i, j >_{QUEUE}$ alors

X égale

$$add(t[k_{length}], add(t[k_{length-1}], \ldots, add(t[k_1], empty)\ldots))$$

où les k_n sont définis par : $k_n = Next(i + n - 1)$. En particulier, k_1 est égal à *i*, et k_{length} est égal à *(j-1)* modulo *succ(Maxlength)* (*length* étant la longueur de la file, c'est-à-dire que *length* égale *(j-i)* modulo *succ(Maxlength)*).

Ensuite, il suffit de raisonner comme nous l'avions fait pour l'exemple de l'implémentation des piles (partie A, chap. II, section 5.4, exemple 13). Considérons par exemple l'*Ok*-axiome d'implémentation

$$eq(i, j) = False \implies \overline{remove}(< t, i, j >_{QUEUE}) = < t, Next(i), j >_{QUEUE}$$

Deux cas sont à considérer :

- □ Si *eq(i,j)* n'est pas égal à *False*, alors cet axiome ne s'applique pas, donc il ne risque pas de créer d'inconsistance

- □ Si *eq(i,j)* est égal à *False*, alors nous admettrons que *i* est différent de *j* dans T_{NAT} (on suppose que cette propriété a déjà été démontrée avant de faire l'implémentation). Dans ce cas, la file *X* représentée par $< t, i, j >_{QUEUE}$ (membre de gauche de l'égalité) n'est pas vide : elle a pour forme normale $add(t[k_{length}], add(t[k_{length-1}], \ldots, add(t[k_1], empty)\ldots))$.
  Par conséquent, le membre de gauche de l'égalité représente la file
  $$remove(add(t[k_{length}], add(t[k_{length-1}], \ldots, add(t[k_1], empty)\ldots)))$$
  or le membre droit représente la file $add(t[k_{length}], add(t[k_{length-1}], \ldots, add(t[k_2], empty)\ldots))$
  ce qui ne crée pas d'inconsistance par rapport à T_{QUEUE}, à cause des *Ok*-axiomes
  $$remove(add(x,empty)) = empty$$
  $$remove(add(x,add(y,X))) = add(x,remove(add(y,X)))$$
  et du fait que la file représentée par $< t, i, j >_{QUEUE}$ (membre gauche) n'est pas vide.

Nous avons maintenant examiné chacune des trois conditions pour qu'une implémentation soit correcte. Nous pouvons poser la définition suivante :

Définition 6 :
Une implémentation abstraite (avec traitement d'exceptions) **IMPL** est *correcte* si et seulement si elle est op-complète, valide et consistante.

5. LA CORRECTION FORTE

Cette section fournit des conditions suffisantes simples pour qu'une implémentation (avec traitement d'exceptions) soit correcte. Pour ce faire, nous définissons une notion de *correction forte* qui implique la correction. De manière similaire au cas sans traitement d'exceptions, la plupart des exemples usuels d'implémentation sont fortement corrects, et nous montrerons dans le chapitre suivant que la correction forte est utile pour dégager les implémentations compatibles avec la réutilisation.

Définition 7 :
Une implémentation **IMPL** de $\mathbf{SPEC_1}$ au moyen de $\mathbf{SPEC_0}$ est *fortement correcte* si et seulement si :

- □ **IMPL** est *op-complète*

- □ **IDimpl** est fortement persistant au-dessus de **EQ** (*validité forte* de **IMPL**)

- □ **IDimpl** est fortement persistante au-dessus de $\mathbf{SPEC_0}$ (**IMPL** *protège la spécification résidante*)

□ **IDimpl** est fortement persistante au-dessus de $\mathbf{SPEC_1}$ (*consistance forte* de **IMPL**).

Remarquons que la différence principale entre la *correction* et la *correction forte* est la protection de le spécification résidante. En effet, lorsque nous avons défini la correction, nous n'avons pris garde qu'au fait que le résultat sémantique soit post-initial ; la correction ne se soucie nullement de ne préserver les sortes résidantes. On comprend bien dès lors que la correction ne soit pas compatible en général avec la réutilisation.

Théorème 4 :
Si l'implémentation **IMPL** est fortement correcte alors elle est correcte.

Preuve :
Rappelons que **IMPL** est correcte si et seulement si elle est op-complète et pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A, *sémantique(*A*)* est une $\mathbf{SPEC_1}$-algèbre post-initiale.
L'op-complétude est déjà incluse dans la définition de correction forte.
Du fait que **IDimpl** est fortement persistante au-dessus de $\mathbf{SPEC_1}$, il résulte que $U_{\mathbf{SPEC_1}}(F_{\mathbf{IDimpl}}(A))$ est une $\mathbf{SPEC_1}$-algèbre post-initiale. De plus, du fait que **IDimpl** est fortement persistante au-dessus de **EQ** et **IDimpl** n'ajoute aucune opération à la signature de **EQ** il résulte que $F_{\mathbf{IDimpl}}(A) = F_{\mathbf{EQ}}(A)$.
Par conséquent, *sémantique(*A*)*$=U_{\mathbf{SPEC_1}}(F_{\mathbf{EQ}}(A)) = U_{\mathbf{SPEC_1}}(F_{\mathbf{IDimpl}}(A))$ est une $\mathbf{SPEC_1}$-algèbre post-initiale. □

Soulignons que la correction forte n'est pas une condition très restrictive. En effet, elle est fondée sur la persistance forte, qui autorise en fait toutes les présentations "raisonnables" ; par exemple la persistance forte autorise l'ajout de valeurs exceptionnelles, pourvu qu'il existe un étiquetage indiquant que cette valeur est erronée (éventuellement par *propagation* d'erreur).
Les résultats précédemment énoncés prouvent que la correction forte fournit une condition suffisante très simple à exprimer pour qu'une implémentation soit correcte.

Chapitre VI :

Réutilisation

d'implémentations

1. MOTIVATION

Rappelons que lorsqu'un utilisateur spécifie un *enrichissement* d'une structure déjà implémentée, décrite par $\mathbf{SPEC_1}$, il le fait en regard de la spécification descriptive $\mathbf{SPEC_1}$, et non en regard de la spécification constructive de **IMPL**. En particulier, toutes les preuves relatives à cet enrichissement seront faites au-dessus de la spécification $\mathbf{SPEC_1}$; donc rien ne prouve *a priori* que l'on obtienne les résultats "attendus" lorsque l'enrichissement est en fait appliqué au-dessus de l'implémentation **IMPL**.

Plus précisément, étant donnée une implémentation **IMPL** de $\mathbf{SPEC_1}$ au moyen de $\mathbf{SPEC_0}$, $F_{\mathbf{EQ}}(\mathrm{A})$ modélise la totalité de l'implémentation, alors que *sémantique(*A*)* n'en modélise que la "vision utilisateur" (pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A). Par conséquent, pour les mêmes raisons que dans le cas sans traitement d'exceptions, les algèbres modélisant un enrichissement **PRES** au-dessus de l'implémentation sont des enrichissements de $F_{\mathbf{EQ}}(\mathrm{A})$, et non l'enrichissement d'une algèbre post-initiale *sémantique(*A*)*.
Ainsi, dire que l'implémentation est compatible avec les enrichissements signifie que, étant donné un enrichissement spécifié par une présentation **PRES** au-dessus de $\mathbf{SPEC_1}$, l'oubli de $F_{\mathbf{EQ+PRES}}(A)$ sur l'exception-signature de $\mathbf{SPEC_1}$+**PRES** doit être ($\mathbf{SPEC_1}$+**PRES**)-post-initiale pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A.

De manière similaire au cas sans traitement d'exceptions, nous démontrons dans les sections suivantes que les implémentations fortement correctes sont compatibles avec les enrichissements fortements persistants ; et nous en déduisons que les implémentations fortement correctes sont compatibles avec la composition.

2. IMPLEMENTATIONS ET ENRICHISSEMENTS

Le théorème suivant généralise, avec traitement d'exceptions, le résultat que nous avions obtenu sans traitement d'exceptions (partie A, chap. II, section 6.1, théorème 6).

Théorème 4 :
Si **IMPL** est une implémentation *fortement correcte* de $\mathbf{SPEC_1}$ au moyen de $\mathbf{SPEC_0}$, alors pour toute présentation *fortement persistante* **PRES** au-dessus de $\mathbf{SPEC_1}$, on a :
pour toute $\mathbf{SPEC_0}$-algèbre post-initiale, $U_{\mathbf{\Sigma}\text{-}exc_1+\mathbf{\Sigma}\text{-}exc_{\mathbf{PRES}}}(F_{\mathbf{EQ+PRES}}(A))$ est une ($\mathbf{SPEC_1}$+**PRES**)-algèbre post-initiale.

Preuve :
La correction forte de **IMPL** implique que **IDimpl** est fortement persistante au-dessus de $\mathbf{SPEC_1}$. La proposition 7, section 6, chapitre VIII de la partie A implique que (**IDimpl**+**PRES**) est fortement persistante au-dessus de ($\mathbf{SPEC_1}$+**PRES**) ; il en résulte que $U_{\mathbf{\Sigma}\text{-}exc_1+\mathbf{\Sigma}\text{-}exc_{\mathbf{PRES}}}(F_{\mathbf{IDimpl+PRES}}(A))$ est une ($\mathbf{SPEC_1}$+**PRES**)-algèbre post-initiale. Il suffit donc de prouver que $F_{\mathbf{IDimpl+PRES}}(A)$ est isomorphe à $F_{\mathbf{EQ+PRES}}(A)$.
Remarquons tout d'abord que la correction forte implique que **IDimpl** est fortement persistante au-dessus de $\mathbf{SPEC_0}$, et la même proposition 7 implique que **IDimpl**+**PRES** est fortement persistante au-dessus de $\mathbf{SPEC_0}$. Il en résulte que $F_{\mathbf{IDimpl+PRES}}(A)$ est une (**IDimpl**+**PRES**)-algèbre post-initiale ; et le même raisonnement, appliqué à la persistance forte de **IDimpl** au-dessus de **EQ** implique que $U_{\mathbf{\Sigma}\text{-}exc_{\mathbf{EQ}}+\mathbf{\Sigma}\text{-}exc_{\mathbf{PRES}}}(F_{\mathbf{IDimpl+PRES}}(A))$ est une (**EQ**+**PRES**)-algèbre post-initiale. Or cette algèbre est un quotient de $F_{\mathbf{EQ+PRES}}(A)$ parce que **IDimpl** n'ajoute aucune opération à **EQ**. Il en résulte finalement que $F_{\mathbf{EQ+PRES}}(A)$ est une (**EQ**+**PRES**)-algèbre post-initiale.
Dès lors l'isomorphisme des algèbres $F_{\mathbf{EQ+PRES}}(A)$ et $F_{\mathbf{IDimpl+PRES}}(A)$ résulte directement du fait que **IDimpl** est fortement persistante au-dessus de **EQ**. □

3. COMPOSITION D'IMPLEMENTATIONS

Supposons que l'on veuille spécifier l'implémentation abstraite d'une spécification $\mathbf{SPEC_2}$ au moyen d'une spécification $\mathbf{SPEC_1}$; et que $\mathbf{SPEC_1}$ soit déjà implémentée au moyen de $\mathbf{SPEC_0}$. Là encore, toutes les preuves de corrections relatives à l'implémentation ($\mathbf{IMPL_2}$) de $\mathbf{SPEC_2}$ sont faites en regard de la spécification descriptive $\mathbf{SPEC_1}$, et nullement par rapport à la spécification constructive de l'implémentation ($\mathbf{IMPL_1}$) de $\mathbf{SPEC_1}$. Par conséquent, rien ne prouve *a priori* que la composition de ces deux implémentations correctes fournisse à l'utilisateur un résultat *globalement* correct.

Lorsque les deux implémentations sont composées, la spécification globale de la structure effectivement implémentée est la réunion des spécifications des deux implémentations $\mathbf{IMPL_1}$ et $\mathbf{IMPL_2}$ à savoir :

$$\mathbf{SPEC_0} + (\mathbf{SORTimpl_1} + \mathbf{C_1} + \cdots + \mathbf{EQ_1}) + (\mathbf{SORTimpl_2} + \mathbf{C_2} + \cdots + \mathbf{EQ_2}) .$$

Le théorème suivant prouve que la composition de deux implémentations (avec traitement d'exceptions) fortement correctes fournit toujours un résultat correct "du point de vue de l'utilisateur". Comme auparavant, le "point de vue de l'utilisateur" est modélisé par l'oubli, sur l'exception-signature de $\mathbf{SPEC_2}$, de la synthèse de toute $\mathbf{SPEC_0}$-algèbre post-initiale.

Théorème 5 :
Etant données une implémentation, $\mathbf{IMPL_1}$, de $\mathbf{SPEC_1}$ au moyen de $\mathbf{SPEC_0}$, et une implémentation, $\mathbf{IMPL_2}$, de $\mathbf{SPEC_2}$ au moyen de $\mathbf{SPEC_1}$. Considérons la spécification **IMPL(1,2)** obtenue en sommant les spécifications des deux implémentations :

$$\mathbf{IMPL(1,2)} = \mathbf{SPEC_0} + (\mathbf{SORTimpl_1} + \cdots + \mathbf{EQ_1}) + (\mathbf{SORTimpl_2} + \cdots + \mathbf{EQ_2})$$

Si $\mathbf{IMPL_1}$ et $\mathbf{IMPL_2}$ sont *fortement correctes*, alors on a :
pour toute $\mathbf{SPEC_0}$-algèbre post-initiale A, $U_{\mathbf{\Sigma}\text{-}exc_2}(F_{\mathbf{IMPL(1,2)}}(\mathrm{A}))$ est une $\mathbf{SPEC_2}$-algèbre post-initiale.

Preuve :
La correction forte de $\mathbf{IMPL_2}$ implique que $(\mathbf{SORTimpl_2} + \cdots + \mathbf{EQ_2})$ est fortement persistante au-dessus de $\mathbf{SPEC_1}$. La correction forte de $\mathbf{IMPL_1}$ et le théorème de la section précédente impliquent donc que $U_{(\mathbf{\Sigma}\text{-}exc_1+\mathbf{\Sigma(IMPL_2)})}(F_{\mathbf{IMPL(1,2)}}(\mathrm{A}))$ est une $(\mathbf{SPEC_1} + \cdots + \mathbf{EQ_2})$-algèbre post-initiale.
La correction forte de **IMPL2** implique de $\mathbf{IDimpl_2}$ est fortement persitant au-dessus de $(\mathbf{SPEC_1} + \cdots + \mathbf{EQ_2})$; et, rappelant que $\mathbf{IDimpl_2}$ n'ajoute aucune opération, ceci implique que toute $(\mathbf{SPEC_1} + \cdots + \mathbf{EQ_2})$-algèbre post-initiale est une $\mathbf{IDimpl_2}$-algèbre post-initiale. $U_{(\mathbf{\Sigma}\text{-}exc_1+\mathbf{\Sigma(IMPL_2)})}(F_{\mathbf{IMPL(1,2)}}(\mathrm{A}))$ est donc une $\mathbf{IDimpl_2}$-algèbre post-initiale.
Enfin la correction forte de $\mathbf{IMPL_2}$ impose que $\mathbf{IDimpl_2}$ est fortement persistante au-dessus de $\mathbf{SPEC_2}$; il en résulte que $U_{\mathbf{\Sigma}\text{-}exc_2}(F_{\mathbf{IMPL(1,2)}}(\mathrm{A}))$ est une $\mathbf{SPEC_2}$-algèbre post-initiale. □

On constate ainsi que tous les résultats obtenus dans le cas sans traitement d'exceptions se généralisent sans difficulté aux exception-algèbres.

CONCLUSION

CONCLUSION

1. RECAPITULATION

Au cours de cette thèse, nous avons principalement développé trois formalismes :

- un nouveau formalisme d'*implémentation abstraite* dans le cadre des types abstraits algébriques, qui fournit d'une part un relevé exhaustif des critères pour qu'une implémentation soit *correcte*, et qui montre d'autre part que ces critères peuvent toujours être exprimés au moyen des notions bien connues de *suffisante complétude* et *consistance hiérarchique* ; de plus, nous avons fourni des conditions suffisantes simples pour que la *composition*, et plus généralement la *réutilisation*, d'implémentations correctes fournisse un résultat correct

- un nouveau formalisme de traitement d'exceptions dans la cadre des types abstraits algébriques, offrant toutes les particularités utiles pour ce sujet : modélisation des *messages d'erreurs*, *propagation implicite* des exceptions et des erreurs, possibilités de *récupérations* d'exceptions ; de plus, nous avons pu redéfinir les notions usuelles de *spécifications structurées*, *consistance hiérarchique* et *suffisante complétude* dans le cadre des types abstraits algébriques avec traitement d'exceptions ; plus généralement, le fait qu'il existe toujours une *congruence minimale* associée à toute exception-présentation permet de généraliser au cas avec traitement d'exceptions la plupart des primitives de structuration des spécifications algébriques déjà connues sans traitement d'exceptions

- enfin, nous avons plus particulièrement porté notre attention sur l'extension du formalisme d'implémentations abstraites au cas avec traitement d'exceptions ; nous avons obtenu sans aucune difficulté les mêmes résultats que dans le cas sans traitement d'exceptions.

Le fait que les deux formalismes d'implémentation abstraite et de traitement d'exceptions puissent être "fondus" sans difficulté semble révélateur de la fiabilité de ces deux formalismes. En effet, ce sont là deux sujets particulièrement délicats des types abstraits algébriques, et le fait que chacun de ces formalismes puisse supporter les particularités de l'autre nous semble être garant des possibilités d'extensions de chacun d'eux. Plus précisément :

- du fait que notre formalisme d'implémentation abstraite utilise une sémantique fondée uniquement sur les foncteurs d'*oubli* et de *synthèse* (synthèse = adjoint à gauche d'un foncteur d'oubli), il pourra être étendu sans difficulté à tout formalisme offrant ces deux types de foncteurs

- du fait que notre formalisme de traitement d'exceptions possède une congruence minimale associée à toute exception-présentation, il offre les mêmes potentialités d'extensions que le

formalisme des types abstraits algébriques “classique” (ADJ), dont les résultats reposent justement sur l’existence de congruences minimales.

Insistons sur le point suivant : bien que le traitement rigoureux des formalismes que nous avons développés ici soit parfois complexe, ces deux formalismes reposent sur des idées simples :

- Les difficultés soulevées par l’implémentation abstraite sont d’une part que certaines valeurs manipulées par l’implémentation ne représentent aucun objet à implémenter, et d’autre part que deux valeurs différentes peuvent représenter le même objet à implémenter. Par rapport aux approches antérieures, celle de [EKMP 80] a permis de définir précisément les critères pour qu’une implémentation soit correcte grâce à l’introduction d’une sémantique traitant explicitement ces deux problèmes. Ces critères étaient cependant difficiles à prouver sur les exemples, car les foncteurs utilisés dans la sémantique de [EKMP 80] étaient relativement complexes.
  Dans notre formalisme, nous traitons ces deux difficultés dès le niveau des présentations, en utilisant à la fois l’abstraction (opérations de *synthèse*) et la *représentation*. La sémantique d’une implémentation abstraite s’exprime alors simplement, et les preuves de corrections peuvent être faites par des “manipulations formelles” sur la syntaxe. Il n’est pas étonnant dès lors que les critères de correction puissent toujours s’exprimer grâce aux notions de consistance hiérarchique et suffisante complétude.

- En ce qui concerne le traitement d’exceptions, nous sommes partis de deux idées :

  - quite à bâtir un formalisme de traitement d’exceptions, autant modéliser aussi la notion de “message d’erreur” ; ce qui conduit assez naturellement à étendre la définition d’une signature en lui adjoignant des *étiquettes d’exception*

  - puisque nous voulions modéliser des *récupérations*, il devenait évident que certains termes auraient une valeur “*Ok*” (par récupération) mais que leur traitement serait exceptionnel ; par conséquent, il fallait distinguer les *termes exceptionnels* de leur *valeur* qui n’est pas nécessairement erronée.

  Du fait que nous ne pouvions donner une sémantique algébrique des exception-algèbres restreinte aux algèbres finiment générées (pour ne pas nuire aux possibilités d’extensions de ce formalisme), il nous a fallu définir une notion de validation “à deux niveaux” (via $T_{\Sigma(A)}$), mais l’idée de base n’en reste pas moins simple.

2. PERSPECTIVES

Du fait de sa similitude avec le formalisme classique (sans traitement d’exceptions), le formalisme des exception-algèbres offre les mêmes possibilités d’extensions. Nous avons déjà précisément défini les notions de *présentation*, *foncteur d’oubli*, *foncteur de synthèse*, *consistance hiérarchique*, *suffisante complétude* et tout ce qui se rapporte à l’*implémentation abstraite* dans le cadre des exception-algèbres ; mais il reste bien sûr de nombreuses voies à explorer.

Par exemple, bien que nous ayons défini les notions de consistance hiérarchique et de suffisante complétude, nous n’avons fourni aucun résultat nouveau sur la manière de les vérifier. Nous avons montré sur quelques exemples de la partie C comment vérifier la suffisante complétude ou la consistance hiérarchique, mais il doit être possible de développer systématiquement des outils déjà connus

dans le cas sans traitement d'exceptions tels que le raisonnement équationnel, l'induction structurelle ou les présentations gracieuses ...

Dans le même ordre d'idée, de même que l'on peut obtenir certains résultats relatifs à une spécification classique lorsque ses axiomes peuvent être orientés en un système de réécriture à terminaison finie, il serait intéressant de définir une notion de réécriture prenant en compte les diverses sortes d'axiomes ou déclarations des exception-spécifications. Plus concrètement, il sera utile de concevoir un évaluateur symbolique fondé sur les exception-algèbres.

Nous avons développé ici une vision résolument initiale des exception-algèbres ; il serait utile de définir une vision "terminale" des spécifications structurées. Notons que ceci doit présenter quelques difficultés non triviales. En effet, les approches terminales sans traitement d'exceptions sont le plus souvent fondées sur un ensemble distingué de sortes "d'observation" fixé à l'avance. Cette approche doit être modifiée ici car une exception-présentation peut ajouter des "messages d'erreur" (modélisés par des étiquettes d'exception), or les messages d'erreur de la présentation en question doivent clairement faire partie des "observateurs". Il faut donc redéfinir la notion d'observateur de manière à prendre en compte les étiquettes d'exception.

Bien d'autres sujets peuvent être abordés en utilisant le formalisme des exception-algèbres ; le fait qu'il existe toujours une congruence minimale associée à une exception-spécification permettra de traiter ces sujets d'une manière très similaire à ce qui est déjà fait avec le formalisme "classique" des types abstraits algébriques.

ANNEXES

TABLE DES MATIERES

TABLE DES MATIERES (SUITE)

ANNEXE 1 :

EXEMPLES DE SPECIFICATIONS ET PRESENTATIONS SANS TRAITEMENT D'EXCEPTIONS

Cet annexe contient les exemples élémentaires les plus courants de spécifications et présentations sans traitement d'exceptions, c'est-à-dire dans le cadre classique des types abstraits algébriques (ADJ).

Dans toute la suite, une spécification **SPEC**, ou une présentation **PRES**, sera définie par :

$$< \mathbf{S} , \mathbf{\Sigma} , \mathbf{E} >$$

où :

- **S** est une ensemble fini de *sortes*
- **Σ** est une ensemble fini d'opérations à arité dans **S** (c'est-à-dire qu'un mot non vide sur **S** est associé à chaque nom d'opération)
- **E** est une ensemble fini d'équations sur la signature <**S**,**Σ**>, c'est-à-dire une paire (non orientée) de **Σ**-termes avec variables.

Les booléens :

BOOL

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 :

$\neg$ *True*	=	*False*
$\neg$ *False*	=	*True*
True $\wedge$ *b*	=	*b*
False $\wedge$ *b*	=	*False*
a $\vee$ *b*	=	$\neg$ *(*$\neg$*a* $\wedge$ $\neg$*b)*
if True then a else b	=	*a*
if False then a else b	=	*b*

Les entiers naturels :

NAT

Il s'agit de la spécification des entiers naturels (non bornés, puisque nous ne disposons d'aucun traitement d'exceptions ici).
Les opérations "prédecesseur" (*pred*), "soustraction" (−), "division" (*div*) et "modulo" (*mod*) posent quelques problèmes, car elles présentent de manière intrinsèque des cas d'application exceptionnels. 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

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 :
$$
\begin{aligned}
pred(succ(n)) &= n \\
n + 0 &= n \\
n + succ(m) &= succ(n) + m \\
n - 0 &= n \\
n - succ(m) &= pred(n) - m \\
n \times 0 &= 0 \\
n \times succ(m) &= (n \times m) + n \\
n < 0 &= False \\
0 < succ(m) &= True \\
succ(n) < succ(m) &= n < m \\
eq(0,0) &= True \\
eq(0,succ(m)) &= False \\
eq(succ(n),0) &= False \\
eq(succ(n),succ(m)) &= eq(n,m) \\
if\ True\ then\ n\ else\ m &= n \\
if\ False\ then\ n\ else\ m &= m \\
n\ div\ succ(m) &= if\ n<succ(m)\ then\ \ n\ \ else\ \ succ(\ (n-succ(m))\ div\ succ(m)\ ) \\
n\ mod\ succ(m) &= n - (n\ div\ succ(m))
\end{aligned}
$$

et par convention :

$$
\begin{aligned}
pred(0) &= 0 \\
n\ div\ 0 &= 0 \\
n\ mod\ 0 &= 0
\end{aligned}
$$

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)*.

Les entiers relatifs :

INT

Il s'agit de la spécification des entiers relatifs (non bornés, puisque nous ne disposons d'aucun traitement d'exceptions ici).
Les opérations "division" (*div*) et "modulo" (*mod*) posent quelques problèmes, car elles présentent de manière intrinsèque des cas d'application exceptionnels. 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 :
$$
\begin{array}{rcl}
pred(succ(x)) & = & x \\
op(0) & = & 0 \\
op(succ(x)) & = & pred(op(x)) \\
op(pred(x)) & = & succ(op(x)) \\
x + 0 & = & x \\
x + succ(y) & = & succ(x) + y \\
x + pred(y) & = & pred(x) + y \\
x - 0 & = & x \\
x - succ(y) & = & pred(x) - y \\
x - pred(y) & = & succ(x) - y \\
x \times 0 & = & 0 \\
x \times succ(y) & = & (x \times y) + x \\
x \times pred(y) & = & (x \times y) - x \\
0 < 0 & = & False \\
0 < succ(0) & = & True \\
0 < succ(succ(y)) & = & if\ \ 0<succ(y)\ \ then\ True\ else\ \ 0<succ(succ(y)) \\
0 < pred(y) & = & if\ \ 0<y\ \ then\ \ 0<pred(y)\ \ else\ False \\
succ(x) < y & = & x < pred(y) \\
pred(x) < y & = & x < succ(y) \\
eq(x,y) & = & \neg\ (\ x<y \vee y<x\ ) \\
if\ True\ then\ x\ else\ y & = & x \\
if\ False\ then\ x\ else\ y & = & y \\
x\ div\ y & = & if\ \ y \times d \leq x < y \times succ(d)\ \ then\ \ d\ \ else\ \ x\ div\ y \\
x\ mod\ y & = & if\ eq(y,0)\ then\ 0\ else\ \ x - (x\ div\ y) \\
x\ div\ 0 & = & 0
\end{array}
$$

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.

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.

- 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≤v*" est une abbréviation pour "*(u<v ⋁ eq(u,v))*" ; et "*0≤x<y*" est une abbréviation pour "*(0≤x ⋀ x<y)*").
Prouver que ces deux manières de spécifier la division sont équivalentes relève du formalisme d'implémentation abstraite.

Les tableaux :

ARRAY

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).

Annexe 1.4 : Les tableaux

Les piles :

STACK

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*

Les chaînes :

STRING

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*)

Les ensembles :

SET

Nous spécifions les ensembles (non bornés) 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*	$\rightarrow$	*SET*	
	$\emptyset$ *:*		$\rightarrow$	*SET*	(ensemble vide)
	ins :	*ELEM SET*	$\rightarrow$	*SET*	(insertion)
	del :	*ELEM SET*	$\rightarrow$	*SET*	(enlever un élément)
	$\in$ *:*	*ELEM SET*	$\rightarrow$	*BOOL*	(élément de)
	$\cup$ *:*	*SET SET*	$\rightarrow$	*SET*	(union)
	$\cap$ *:*	*SET SET*	$\rightarrow$	*SET*	(intersection)
	$-$ *:*	*SET SET*	$\rightarrow$	*SET*	(différence)
	Δ *:*	*SET SET*	$\rightarrow$	*SET*	(différence symétrique)
	card :	*SET*	$\rightarrow$	*NAT*	(cardinal)
	is-empty :	*SET*	$\rightarrow$	*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 :

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

Les files :

QUEUE

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-empty(X)	=	*eq(length(X),0)*

et par convention :

remove(new)	=	*new*
first(new)	=	*e*

ANNEXE 2 :

EXEMPLES D'IMPLEMENTATIONS ABSTRAITES SANS TRAITEMENT D'EXCEPTIONS

Cet annexe contient divers exemples simples d'implémentations abstraites sans traitement d'exceptions.

Pour les spécifier, nous suivons les notations décrites dans la partie A, chapitre II, à la fin de la section 2 (pages 49 et 50).

Annexe 2 : exemples d´implémentations abstraites classiques

Annexe 2.1

Implémentation de *NAT*

par *INT*

Cette implémentation est particulièrement simple, puisque *NAT* est simplement un sous ensemble de *INT*. La seule difficulté résulte de l'application de l'opération *pred* à *0*, qui retourne *0* pour les entiers naturels, mais *-1* pour les relatifs.
Voici la spécification de cette implémentation :

L'arité de l'opération d'abstraction est :

$$< \_ > : \quad INT \quad -implémente\rightarrow \quad NAT$$

$$\begin{array}{rcrcl}
 & & 0_{NAT} & \approx & < 0_{INT} > \\
 & & succ_{NAT}(< z >) & \approx & < succ_{INT}(z) > \\
eq_{INT}(z, 0) = False & \Longrightarrow & pred_{NAT}(< z >) & \approx & < pred_{INT}(z) > \\
 & & pred_{NAT}(< 0_{INT} >) & \approx & < 0_{INT} > \\
 & & < z > +_{NAT} < z' > & \approx & < z +_{INT} z' > \\
z <_{INT} z' = False & \Longrightarrow & < z > -_{NAT} < z' > & \approx & < z -_{INT} z' > \\
z <_{INT} z' = True & \Longrightarrow & < z > -_{NAT} < z' > & \approx & < 0_{INT} >
\end{array}$$

Les autres opérations ne posent aucun problème ...
il ne s'agit que de copies.

On constate ici, une fois encore, le manque évident de traitement d'exceptions pour traiter cette implémentation de manière efficace.

Remarquons que la représentation de l'égalité est vide, puisque deux entiers relatifs distincts représentent deux entiers naturels distincts.

Annexe 2.2

Implémentation de *STACK*

par *ARRAY* et *NAT*

Récapitulons ici cette implémentation déjà spécifiée au cours de la partie A. Rappelons que, pour pouvoir spécifier complètement le cas exceptionnel *top(empty)*, il s'agit de piles et de tableaux d'éléments tels que la sorte *ELEM* contienne un élément particulier *e* .

L'opération d'abstraction est :

$$<\_\ ,\_\ > :\ \ ARRAY\ \ NAT\ \ -implémente\rightarrow\ \ STACK$$

Avec les axiomes :

$$\begin{aligned} empty &\approx <t,0> \\ push(n,<t,i>) &\approx <t[i]:=n,succ(i)> \\ pop(<t,0>) &\approx <t,0> \\ pop(<t,succ(i)>) &\approx <t,i> \\ top(<t,0>) &\approx e \\ top(<t,succ(i)>) &\approx t[i] \\ height(<t,i>) &\approx i \\ is\text{-}empty(<t,i>) &\approx eq(i,0) \end{aligned}$$

La représentation de l'égalité peut être spécifiée comme suit :

$$<t,0> = <t',0>$$

$$<t,i>=<t',i> \wedge t[i]=t'[i] \Rightarrow <t,succ(i)> = <t',succ(i)>$$

Annexe 2.3

Implémentation de *STACK*

par *STRING*

Ici encore, on suppose qu'il s'agit de chaînes et piles d'éléments tels que la sorte *ELEM* contienne un élément particulier *e*, afin de spécifier complètement le cas exceptionnel *top(empty)*.

L'opération d'abstraction est :

$$<\_> : \quad STRING \quad -implémente\rightarrow \quad STACK$$

Et les axiomes d'implémentation sont :

$$\begin{aligned}
empty &\approx <\lambda> \\
push(x,<s>) &\approx <add(x,s)> \\
pop(<\lambda>) &\approx <\lambda> \\
pop(<add(x,s)>) &\approx <s> \\
top(<\lambda>) &\approx e \\
top(<add(x,s)>) &\approx x \\
height(<s>) &\approx length(s) \\
is\text{-}empty(<s>) &\approx is\text{-}empty(s)
\end{aligned}$$

La représentation de l'égalité est bien sûr vide puisque deux chaînes différentes représentent nécessairement deux piles différentes.

Annexe 2.4

Implémentation de *STRING*

par *ARRAY* et *NAT*

L'opération d'abstraction est :

$$<\_\ ,\_\ > :\quad ARRAY\ NAT\quad -implémente\rightarrow\quad STRING$$

Nous utiliserons une composante cachée, afin de définir une opération qui transfert une portion de tableau dans un autre tableau :
l'opération cachée : *transfert(t,i,t',j,k)* a pour rôle de transférer les éléments du tableau *t'* compris entre les rangs *j* et *j+k* (*j+k* exclu), dans le tableau *t* à partir du rang *i* (*i* inclus).
Ceci est spécifié au moyen des axiomes cachés suivants :

$$\begin{aligned} transfert(t,i,t',j,0) &= t \\ transfert(t,i,t',j,succ(k)) &= transfert(t[i]:=t'[j],succ(i),t',succ(j),k) \end{aligned}$$

Les axiomes d'implémentation sont alors :

$$\begin{aligned} \lambda &\approx <t,0> \\ add(x,<t,i>) &\approx <t[i]:=x,succ(i)> \\ concat(<t,i>,<t',j>) &\approx <transfert(t,i,t',0,j),i+j> \\ length(<t,i>) &\approx i \\ is\text{-}empty(<t,i>) &\approx eq(i,0) \end{aligned}$$

La représentation de l'égalité peut être spécifiée comme suit :

$$<t,0> = <t',0>$$
$$<t,i>=<t',i> \wedge t[i]=t'[i] \Rightarrow <t,succ(i)> = <t',succ(i)>$$

Annexe 2.5

Implémentation de *SET*

par *STRING*

Récapitulons ici cette implémentation déjà spécifiée au cours de la partie A.

Nous spécifions d'abord une composante cachée qui enrichit la spécification *STRING* des opérations *occurs*, *remove* et *delete*. L'opération *occurs(x,s)* retourne *True* si l'élément *x* est dans la chaîne *s*, et *False* sinon. L'opération *remove(x,s)* retourne la chaîne *s* privée de sa premiere occurrence (éventuelle) de l'élément *x* ; en particulier, si *x* n'est pas dans *s*, alors la chaîne *s* elle même est retournée. Enfin l'opération *delete(d,s)* retourne la chaîne *s* privée de la première première occurrence (éventuelle) de chacun des éléments contenus dans la chaîne *d* ; en particulier, si *d* ne contient aucun élément en commun avec *s* alors la chaîne *s* est elle même retournée, et si *d* contient tous les éléments de *s* (dans n'importe quel ordre) alors la chaîne vide est retournée.
Ceci est obtenu par les axiomes cachés suivants :

$$
\begin{array}{lrcl}
 & occurs(x,\lambda) & = & False \\
 & occurs(x,add(y,s)) & = & eq(x,y) \vee occurs(x,s) \\
 & remove(x,\lambda) & = & \lambda \\
 & remove(x,add(x,s)) & = & s \\
eq(x,y)=False \quad \Rightarrow & remove(x,add(y,s)) & = & add(y,remove(x,s)) \\
 & delete(\lambda,s) & = & s \\
 & delete(add(x,d),s) & = & delete(d,remove(x,s))
\end{array}
$$

L'opération d'abstraction est :

$$< \_ > : \quad STRING \quad -implémente\rightarrow \quad SET$$

Les axiomes d'implémentation sont alors les suivants :

$$
\begin{array}{lcrcl}
 & & \emptyset & \approx & <\lambda> \\
occurs(x,s)=False & \Rightarrow & ins(x,<s>) & \approx & <add(x,s)> \\
occurs(x,s)=True & \Rightarrow & ins(x,<s>) & \approx & s \\
 & & del(x,<s>) & \approx & <remove(x,s)> \\
 & & x \in <s> & \approx & occurs(x,s) \\
 & & <s> \cup <s'> & \approx & concat(delete(s,s'),s) \\
 & & <s> \cap <s'> & \approx & delete(s,delete(s',s)) \\
 & & <s> - <s'> & \approx & delete(s',s) \\
 & & <s> \Delta <s'> & \approx & concat(delete(s',s),delete(s,s')) \\
 & & card(<s>) & \approx & length(s) \\
 & & is\text{-}empty_{SET}(< s >) & \approx & is\text{-}empty_{STRING}(s)
\end{array}
$$

Enfin la représentation de l'égalité traduit simplement la commutativité de l'insertion ensembliste :

$$<add(x,add(y,s))> \quad = \quad <add(y,add(x,s))>$$

Annexe 2.5 : implémentation de *SET* par *STRING*

Annexe 2.6

Implémentation de *SET*

par *ARRAY*

Rappelons que nous avons défini la présentation *ARRAY* au-dessus d'une spécification *RANG* quelconque (munie d'un prédicat d'égalité), et d'une spécification *ELEM* quelconque (munie d'une valeur distinguée *e*, que contient uniformément le tableau initial *create*).
Nous allons considérer des tableaux de booléens, avec *e=False*, et la spécification *RANG* sera celle des éléments des ensembles que nous voulons implémenter. Donc, en fait, nous instancions les rangs des tableaux (*RANG*) par la spécification des éléments des ensembles ; et nous instancions les éléments des tableaux (*ELEM*) par *BOOL*.

Nous allons implémenter des ensembles (*SET*) dont les éléments seront de sorte *RANG*. Un ensemble *s* sera donc implémenté par un tableau *t* de telle sorte que *r* (de sorte *RANG*) sera élément de *s* si et seulement si *t[r]* égale *True*.

L'opération d'abstraction est bien sûr :

$$<\_> : \quad ARRAY \quad -implémente\rightarrow \quad SET$$

Nous utilisons une composante cachée afin de définir les opérations *AND* (*et* logique), *OR* (*ou* logique) et Δ (difference symétrique) globalement sur les tableaux de booléens (rappelons que *create* contient uniformément *False*) :

$$\begin{aligned}
create\ AND\ t &= create \\
t'[r]:=True\ \ AND\ \ t &= (t'\ AND\ t)[r] := t[r] \\
create\ OR\ t &= t \\
t'[r]:=True\ \ OR\ \ t &= (t'\ OR\ t)[r] := True \\
create\ \Delta\ t &= t \\
t'[r]:=True\ \ \Delta\ \ t &= (t'\ \Delta\ t)[r] := \neg\ t[r]
\end{aligned}$$

Et les axiomes d'implémentation sont les suivants :

$$
\begin{array}{rcrcl}
 & & \varnothing & \approx & create \\
 & & ins(r,<t>) & \approx & t[r]:=True \\
 & & del(r,<t>) & \approx & t[r]:=False \\
 & & r \in <t> & \approx & t[r] \\
 & & <t> \cup <t'> & \approx & t\ OU\ t' \\
 & & <t> \cap <t'> & \approx & t\ AND\ t' \\
 & & <t> - <t'> & \approx & (t \Delta t')\ AND\ t \\
 & & <t> \Delta <t'> & \approx & t \Delta t' \\
 & & card(<create>) & \approx & 0 \\
t[r]:=False & \Rightarrow & card(<t[r]:=True>) & \approx & succ(card(<t>)) \\
 & & is\text{-}empty(<t>) & \approx & eq(card(<t>),0)
\end{array}
$$

La représentation de l'égalité est vide puisque deux tableaux ne représentent le même ensemble que s'ils sont égaux.

Annexe 2.6 : implémentation de *SET* par *ARRAY*

Annexe 2.7

Implémentation de *QUEUE*

par *ARRAY* et *NAT*

Récapitulons ici cette implémentation déjà spécifiée au cours de la partie A. Rappelons que, pour pouvoir spécifier complètement le cas exceptionnel *first(empty)*, il s'agit de piles et de tableaux d'éléments tels que la sorte *ELEM* contienne un élément particulier *e* . Mais nous savons que sur cet exemple, le manque de traitement d'exceptions est surtout ressenti par une "fuite vers l'infini" de la file dans le tableau.

L'opération d'abstraction est :

$$<\_,\_,\_> : \quad ARRAY \quad NAT \quad NAT \quad -implémente\rightarrow \quad QUEUE$$

Avec les axiomes :

$$\begin{array}{lrcl}
 & new & \approx & <t,i,i> \\
 & add(x,<t,i,j>) & \approx & <t[j]:=x,i,succ(j)> \\
eq(i,j)=False \quad \Rightarrow & remove(<t,i,j>) & \approx & <t,succ(i),j> \\
 & remove(<t,i,i>) & \approx & <t,i,i> \\
eq(i,j)=False \quad \Rightarrow & first(<t,i,j>) & \approx & t[i] \\
 & first(<t,i,i>) & \approx & e \\
 & length(<t,i,j>) & \approx & j-i \\
 & is\text{-}empty(<t,i,j>) & \approx & eq(i,j)
\end{array}$$

Enfin les axiomes de représentation de l'égalité sont :

$$<t,i,i> \; = \; <t',k,k>$$

$$<t,i,j> = <t',k,l> \wedge t[j]=t'[l] \quad \Rightarrow \quad <t,i,succ(j)> = <t',k,succ(l)>$$

ANNEXE 3 :

EXEMPLES D'EXCEPTION-SPECIFICATIONS ET EXCEPTION-PRESENTATIONS

Cet annexe contient des exemples parmi les plus courants d'exception-spécifications et d'exception-présentations dans le cadre du formalisme des exception-algèbres.

Dans toute la suite, une exception-spécification **SPEC**, ou une exception-présentation **PRES**, sera définie par (cf. partie B) :

$$< \mathbf{\Sigma}\text{-}exc\ ,\ \mathbf{Ok\text{-}Frm}\ ,\ \mathbf{Ok\text{-}Ax}\ ,\ \mathbf{Lbl\text{-}Ax}\ ,\ \mathbf{Gen\text{-}Ax} >$$

où :

- □ $\mathbf{\Sigma}$-*exc* = <**S**,$\mathbf{\Sigma}$,**L**> est une exception-signature (définition 1, page 97)
- □ **Ok-Frm** est une déclaration de formes *Ok* sur $\mathbf{\Sigma}$-*exc* (définition 2, page 99)
- □ **Ok-Ax** est un ensemble fini d'*Ok*-axiomes sur $\mathbf{\Sigma}$-*exc* (définition 3, page 100)
- □ **Lbl-Ax** est un ensemble fini d'axiomes d'étiquetage sur $\mathbf{\Sigma}$-*exc* (définition 4, page 101)
- □ **Gen-Ax** est un ensemble fini d'axiomes généralisés sur $\mathbf{\Sigma}$-*exc* (définition 5, page 103).

Annexe 3 : exemples d´exception-spécifications

Les booléens

Cette spécification est trivialement simple. En fait, elle ne nécessite aucun traitement d'exceptions, puisque toutes les opérations booléennes, appliquées à des valeurs booléennes *Ok*, retournent une valeur *Ok*. Hormis la déclaration (très simple) de formes *Ok*, elle ne diffère donc pas de la spécification classique (sans traitement d'exceptions) des booléens.

S = { *BOOL* }

Σ = { *True* , *False* , ¬_ , _ ∧_ , _ ∨_ , *if*_ *then*_ *else*_ }
avec les arités évidentes (et nécessaires ici, puisqu'il n'y a qu'une seule sorte).

L = ∅

Ok-Frm : *True* ∈ Ok-Frm
False ∈ Ok-Frm

Ok-Ax :

¬ *True*	=	*False*
¬ *False*	=	*True*
True ∧ *b*	=	*b*
False ∧ *b*	=	*False*
a ∨ *b*	=	¬ (¬*a* ∧ ¬*b*)
if True then a else b	=	*a*
if False then a else b	=	*b*

Lbl-Ax et **Gen-Ax** sont vides, puisqu'aucun traitement d'exceptions n'est requis.

Les entiers naturels

non bornés

Nous présentons ici diverses exception-spécifications des entiers naturels non bornés, avec des “degrés” de traitement d’exceptions progressifs ; ceci afin de démontrer que notre formalisme conduit à des spécifications simples et courtes lorsque l’on se limite aux structures de données présentées en exemples dans les divers autres formalismes de traitement d’exceptions.

1. CAS SANS ETIQUETAGE

L’exemple que nous spécifions ici est est représentatif de ce qui peut être spécifié sans cas de récupération, avec tous les formalismes que nous avions cités en partie B, chapitre I, section 3. Aucun disgnostic d’exception n’est donné (ces formalismes ne disposent pas d’étiquette d’exception), c’est-à-dire que **L**, **Lbl-Ax** et **Gen-Ax** sont vides.

S = { *NAT* }

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

Ok-Frm : $0 \in$ Ok-Frm
$n \in$ Ok-Frm $\Rightarrow$ $succ(n) \in$ Ok-Frm

Ok-Ax :

$$
\begin{aligned}
pred(succ(n)) &= n \\
n + 0 &= n \\
n + succ(m) &= succ(n) + m \\
n - 0 &= n \\
n - succ(m) &= pred(n) - m \\
n \times 0 &= 0 \\
n \times succ(m) &= (n \times m) + n \\
n < 0 &= False \\
0 < succ(m) &= True \\
succ(n) < succ(m) &= n < m \\
eq(0,0) &= True \\
eq(0,succ(m)) &= False \\
eq(succ(n),0) &= False \\
eq(succ(n),succ(m)) &= eq(n,m)
\end{aligned}
$$

Remarquons que, par rapport au formalisme de [GDLE 84], notre composante **Ok-Frm** est ici exactement équivalente à déclarer que les opérations *0* et *succ* sont “safe”. Les algèbres initiales respectivement obtenues avec le formalisme des exception-algèbres et avec les formalisme [GDLE 84] sont alors isomorphes. Néanmoins, notre formalisme permet de déduire de manière *totalement implicite* que les opérations “×”, “<” et “*eq*” sont “safe”.
Par rapport au formalisme des E,R-algèbres de [Bid 84], **Ok-Frm** ne serait pas déclaré dans ce formalisme, mais par contre il faudrait déclarer explicitement que les termes *pred(0)*, et *n–m* lorsque *n* est strictement inférieur à *m*, sont des cas d’erreur. Ceci est automatiquement déduit de manière *implicite* dans notre formalisme.

2. CAS AVEC RECUPERATIONS

Nous allons spécifier ici une récupération des termes négatifs sur *0*. Les composantes <**S**,**Σ**,**L**> et **Ok-Frm** sont les mêmes que dans la section précédente ; nous ajoutons simplement les axiomes généralisés suivants (dans **Gen-Ax**) :

$$pred(0) = 0$$
$$n<m = True \quad \Rightarrow \quad n–m = 0$$

Dans le cas du formalisme de [GDLE 84], ceci est spécifié par les mêmes axiomes ; mais rappelons que ce formalisme ne contient qu’une seule classe d’axiomes, si bien que rien ne permet de distinguer syntaxiquement que ces axiomes relèvent du traitement d’exceptions.
Dans le cas du formalisme de [Bid 84], il suffit d’ajouter le premier axiome parmi les axiomes traitant les valeurs *Ok*, et d’ajouter la déclaration de récupération suivante :

pred(0) est un terme récupéré

Ainsi, ce cas de récupération est explicitement cité, remarquons néanmoins que l’axiome *pred(0)=0* est spécifié au sein des *Ok*-axiomes, bien que ce soit une récupération.

3. CAS AVEC ETIQUETAGE

Pour spécifier la récupération précédente avec le formalisme des exception-algèbres, il est clair que nous préfèrerons écrire une récupération de la forme suivante :

$$n \in NEGATIVE\text{-}VALUE \quad \Rightarrow \quad n = 0$$

avec les étiquetages suivants, spécifiés dans **Lbl-Ax** :

$$pred(0) \in NEGATIVE\text{-}VALUE$$
$$n<m = True \quad \Rightarrow \quad n-m \in NEGATIVE\text{-}VALUE$$

Ceci n'est bien sûr possible qu'à condition de disposer d'un étiquetage des valeurs exceptionnelles. Par conséquent, spécifier ainsi une telle récupération avec les formalisme [GDLE 84] ou [Bid 84] n'est pas possible.
Seul un formalisme utilisant les sous-sortes (cf. OBJ2) pourrait atteindre une spécification de ce genre en créant une sous-sorte de *NAT* nommée $NAT_{NEGATIVE\text{-}VALUE}$. Néanmoins, pour un tel formalisme, les spécifications données dans les sections précédentes ne pourraient pas déduire *implicitement* que les termes *pred(0)* et *n–m* sont exceptionnels. De plus, et surtout, cette récupération de pourrait pas être spécifiée comme une présentation au-dessus de *NAT* sans récupération ; en effet, sans cette récupération la sorte $NAT_{NEGATIVE\text{-}VALUE}$ doit être déclaré comme une sous-sorte de NAT_{erreur}, alors qu'avec cette récupération elle devient une sous-sorte de NAT_{Ok} ; par conséquent, les récupérations induisent des modifications de l'ordre sous-jacent des sous-sortes.

Les entiers naturels

bornés

Récapitulons ici la spécification des entiers naturels bornés déjà développée pas à pas durant la partie B, chapitre III. Il s'agit d'une exception-présentation au-dessus des booléens *BOOL*.

S = { *NAT* }

Σ = { *0, Maxint, crash, succ_ , _ <_ , pred_ , _ +_ , _ −_ , _ ×_ , _ div_* }
avec les arités évidentes.

L = { *NEGATIVE-NUMBER , TOO-LARGE-NUMBER , DIV-BY-0-ERROR* }

Ok-Frm : $succ^{Maxint}(0) \in$ Ok-Frm
$succ(n) \in$ Ok-Frm $\Rightarrow$ $n \in$ Ok-Frm

Ok-Ax :

$$
\begin{aligned}
& & Maxint &= succ^{Maxint}(0) \\
& & n < 0 &= False \\
& & 0 < succ(m) &= True \\
& & succ(n) < succ(m) &= n < m \\
& & pred(succ(n)) &= n \\
& & n + 0 &= n \\
& & n + succ(m) &= succ(n) + m \\
& & n - 0 &= n \\
& & n - succ(m) &= pred(n) - m \\
& & n \times 0 &= 0 \\
& & n \times succ(m) &= (n \times m) + n \\
r < m = True & \Rightarrow & ((n \times m)+r) \ div \ m &= n
\end{aligned}
$$

Lbl-Ax :

$$pred(0) \in NEGATIVE\text{-}NUMBER$$
$$n \in NEGATIVE\text{-}NUMBER \Rightarrow pred(n) \in NEGATIVE\text{-}NUMBER$$
$$n < m = True \Rightarrow (n - m) \in NEGATIVE\text{-}NUMBER$$
$$succ(Maxint) \in TOO\text{-}LARGE\text{-}NUMBER$$
$$n \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow succ(n) \in TOO\text{-}LARGE\text{-}NUMBER$$
$$n \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow (n + 0) \in TOO\text{-}LARGE\text{-}NUMBER$$
$$succ(n)+m \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow n+succ(m) \in TOO\text{-}LARGE\text{-}NUMBER$$
$$n \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow (n \times 1) \in TOO\text{-}LARGE\text{-}NUMBER$$
$$(n \times m + n) \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow (n \times succ(m)) \in TOO\text{-}LARGE\text{-}NUMBER$$
$$(n \text{ div } 0) \in DIV\text{-}BY\text{-}0\text{-}ERROR$$

Gen-Ax :

$$n \in NEGATIVE\text{-}NUMBER \Rightarrow n = crash$$
$$n \text{ div } 0 = crash$$
$$succ(crash) = crash$$
$$pred(crash) = crash$$
$$crash + n = crash$$
$$crash - n = crash$$
$$n - crash = crash$$
$$crash \times n = crash$$
$$crash \text{ div } n = crash$$
$$n \text{ div } crash = crash$$
$$n + m = m + n$$
$$n \times m = m \times n$$
$$n \in TOO\text{-}LARGE\text{-}NUMBER \wedge m \in Ok \Rightarrow n < m = False$$
$$n \in Ok \wedge m \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow n < m = True$$
$$\left.\begin{array}{r} succ(n) \in TOO\text{-}LARGE\text{-}NUMBER \\ succ(m) \in TOO\text{-}LARGE\text{-}NUMBER \end{array}\right\} \Rightarrow succ(n) < succ(m) = n < m$$
$$succ(n) \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow pred(succ(n)) = n$$
$$n \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow n + 0 = n$$
$$n+succ(m) \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow n + succ(m) = succ(n) + m$$
$$n \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow n - 0 = n$$
$$n \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow n - succ(m) = pred(n) - m$$
$$n \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow n \times 0 = 0$$
$$n \times succ(m) \in TOO\text{-}LARGE\text{-}NUMBER \Rightarrow n \times succ(m) = (n \times m) + n$$
$$r < m = True \Rightarrow ((n \times m)+r) \text{ div } m = n$$

Se repporter au chapitre III de la partie B pour les divers commentaires relatifs à cette spécification.

Entiers relatifs

bornés

Il s'agit d'une présentation au-dessus des booléens *BOOL*. Les cas exceptionnels résultent bien sûr de l'application des diverses opérations à des valeurs *Ok* lorsqu'elles retournent une valeur hors de l'intervalle [–*lowerbound*,*upperbound*]. Nous choisissons ici de récupérer les divisions par *0* sur l'une des bornes –*lowerbound* ou *upperbound* ; mais nous ne récupérerons pas les *"n modulo 0"* afin de montrer que, bien que *mod* soit défini au moyen de *div*, il est possible d'interdir la propagation des récupérations de *div* (dans l'algèbre initiale).
Noter que si l'exception-spécification qui suit est un peu longue, ceci est en grande partie dû au fait que *lowerbound* n'est pas supposé égal à *upperbound*. Spécifier un intervalle *Ok* de la forme [–*bound*,+*bound*] serait considérablement plus court ; cependant nous préférons présenter ici le cas général.

S = { *INT* }

Σ = { 0 , *succ_* , *pred_* , *op_* , _ +_ , _ −_ , _ ×_ , _ <_ , *eq(_ ,_)* , _ *div*_ , _ *mod*_ }
avec les arités évidentes.

L = { *TOO-SMALL-INT, TOO-LARGE-INT, DIV-BY-0-ERROR* }

Ok-Frm : $succ^{upperbound}(0) \in$ Ok-Frm
$pred_{lowerbound}(0) \in$ Ok-Frm
$succ(z) \in$ Ok-Frm $\Rightarrow$ $z \in$ Ok-Frm
$pred(z) \in$ Ok-Frm $\Rightarrow$ $z \in$ Ok-Frm

Ok-Ax :

$pred(succ(z)) = z$
$succ(pred(z)) = z$
$op(0) = 0$
$op(succ(z)) = pred(op(z))$
$op(pred(z)) = succ(op(z))$
$z + 0 = z$
$z + succ(y) = succ(z+y)$
$z + pred(y) = pred(z+y)$
$(z+y) - y = z$
$z \times 0 = 0$
$z \times succ(y) = (z \times y) + z$
$z \times pred(y) = (z \times y) - z$
$pred^{lowerbound}(0) < pred^{lowerbound}(0) = False$
$pred^{lowerbound}(0) < succ(z) = True$
$eq(z,y) = \neg \barwedge \neg$
$pred(0) < r < y = True \Rightarrow ((z \times y) + r)\ div\ y = z$
$z\ mod\ y = z - ((z\ div\ y) \times y)$

Lbl-Ax :

$succ^{upperbound+1}(0) \in TOO\text{-}LARGE\text{-}INT$
$pred^{lowerbound+1}(0) \in TOO\text{-}SMALL\text{-}INT$
$z > succ^{lowerbound}(0) \Rightarrow op(z) \in TOO\text{-}SMALL\text{-}INT$
$z < pred^{upperbound}(0) \Rightarrow op(z) \in TOO\text{-}LARGE\text{-}INT$
$succ^{upperbound}(0) + succ(0) \in TOO\text{-}LARGE\text{-}INT$
$pred^{lowerbound}(0) + pred(0) \in TOO\text{-}SMALL\text{-}INT$
$z+y \in TOO\text{-}LARGE\text{-}INT \Rightarrow z+succ(y) \in TOO\text{-}LARGE\text{-}INT$
$z+y \in TOO\text{-}SMALL\text{-}INT \Rightarrow z+pred(y) \in TOO\text{-}SMALL\text{-}INT$
$z+y \in TOO\text{-}LARGE\text{-}INT \Rightarrow pred(z)+succ(y) \in TOO\text{-}LARGE\text{-}INT$
$z+y \in TOO\text{-}LARGE\text{-}INT \Rightarrow succ(z)+pred(y) \in TOO\text{-}SMALL\text{-}INT$
$z < (y + pred^{lowerbound}(0)) = True \Rightarrow z-y \in TOO\text{-}SMALL\text{-}INT$
$z > (y + succ^{upperbound}(0)) = True \Rightarrow z-y \in TOO\text{-}LARGE\text{-}INT$
$(z \times y)+z \in TOO\text{-}LARGE\text{-}INT \Rightarrow z \times succ(y) \in TOO\text{-}LARGE\text{-}INT$
$(z \times y)+z \in TOO\text{-}SMALL\text{-}INT \Rightarrow z \times succ(y) \in TOO\text{-}SMALL\text{-}INT$
$(z \times y)-z \in TOO\text{-}LARGE\text{-}INT \Rightarrow z \times pred(y) \in TOO\text{-}LARGE\text{-}INT$
$(z \times y)-z \in TOO\text{-}SMALL\text{-}INT \Rightarrow z \times pred(y) \in TOO\text{-}SMALL\text{-}INT$
$z\ div\ 0 \in DIV\text{-}BY\text{-}0\text{-}ERROR$
$z\ mod\ 0 \in DIV\text{-}BY\text{-}0\text{-}ERROR$

Gen-Ax : $0\ div\ 0 = succ(0)$

$z<0 = True \Rightarrow z\ div\ 0 = pred^{lowerbound}(0)$
$0<z = True \Rightarrow z\ div\ 0 = succ^{upperbound}(0)$

Naturellement, bien d'autres récupérations peuvent être envisagées.

Tableaux bornés

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*, munie d'un prédicat d'égalité sur la sorte *RANG* (noté *eq*), de deux rangs particuliers *lower-range* et *upper-range*, et d'une relation d'ordre total dans l'algèbre initiale (noté ≤) telle que *lower-range*<*upper-range*. Cette sorte *RANG* sera celle des indices des tableaux (le plus souvent *NAT* dans les exemples).
- Une spécification *ELEM* quelconque.

S = { *ARRAY* }

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

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*.

L = { *OUT-OF-RANGE, NOT-INITIALIZED* }

Ok-Frm : (on considère comme terme *Ok* tout tableau n'ayant reçu que des affectations d'éléments *Ok* à l'intérieur des rangs [*lower-range,upper-range*], même s'il contient des rangs non initialisés).

$$create \in \text{Ok-Frm}$$

$$lower\text{-}range \le i \le upper\text{-}range = True \wedge x \in Ok \wedge t \in \text{Ok-Frm} \implies t[i]:=x \in \text{Ok-Frm}$$

Ok-Ax : $(t[i]:=x)[i] = x$

$eq(i,j) = False \implies (t[i]:=x)[j]:=y = (t[j]:=y)[i]:=x$

$(t[i]:=x)[i]:=y = t[i]:=y$

Lbl-Ax : $create[i] \in NOT\text{-}INITIALIZED$

$eq(i,j)=False \wedge t[i] \in NOT\text{-}INITIALIZED \Rightarrow (t[j]:=x)[i] \in NOT\text{-}INITIALIZED$

$i>upper\text{-}range = True \Rightarrow t[i] \in OUT\text{-}OF\text{-}RANGE$

$i<lower\text{-}range = True \Rightarrow t[i] \in OUT\text{-}OF\text{-}RANGE$

$i>upper\text{-}range = True \Rightarrow t[i]:=x \in OUT\text{-}OF\text{-}RANGE$

$i<lower\text{-}range = True \Rightarrow t[i]:=x \in OUT\text{-}OF\text{-}RANGE$

En ce qui concerne les axiomes généralisés (**Gen-Ax**), on peut imaginer toute récupération utile ... voir Exemple 14, section 1.2, chapitre VI, partie B (pages 137 & 138).

Piles bornées

Nous spécifions les piles bornées comme une présentation *STACK* au-dessus de *NAT* (pour pouvoir spécifier l'opération "hauteur"), et d'une spécification prédéfinie *ELEM* contenant une sorte *ELEM* (les éléments placés dans les piles seront ceux de sorte *ELEM*). Naturellement, il est possible que *ELEM* soit en fait *NAT*. On considère des piles de hauteur maximale *Maxheight*.

S = { *STACK* }

Σ =	*empty :*		$\rightarrow$	*STACK*	(pile vide)
	push :	*ELEM STACK*	$\rightarrow$	*STACK*	(empile un élément)
	pop :	*STACK*	$\rightarrow$	*STACK*	(depile d'un cran)
	top :	*STACK*	$\rightarrow$	*ELEM*	(élément en haut de pile)
	height :	*STACK*	$\rightarrow$	*NAT*	(hauteur)

L = { *UNDERFLOW, OVERFLOW, STACK-IS-EMPTY* }

Ok-Frm =

$x_1 \in$ Ok-Frm $\wedge \cdots \wedge$ $x_{Maxheight} \in$ Ok-Frm $\Rightarrow$ $push(x_1, push(x_2, \ldots, push(x_{Maxheight}, empty) \in$ Ok-Frm

$push(x,X) \in$ Ok-Frm $\Rightarrow$ $X \in$ Ok-Frm

Ok-Ax :	*pop(push(x,X))*	=	*X*
	top(push(x,X))	=	*x*
	height(empty)	=	*0*
	height(push(x,X))	=	*succ(height(X))*

Lbl-Ax :	*pop(empty)* $\in$ *UNDERFLOW*	
	top(empty) $\in$ *STACK-IS-EMPTY*	
X $\in$ *UNDERFLOW*	$\Rightarrow$	*pop(X)* $\in$ *UNDERFLOW*
height(X)=Maxheight	$\Rightarrow$	*push(x,X)* $\in$ *OVERFLOW*
X $\in$ *OVERFLOW*	$\Rightarrow$	*push(x,X)* $\in$ *OVERFLOW*

Tout traitement des piles exceptionnelles peut être imaginé dans **Gen-Ax**. Par exemple, si l'on veut identifier toutes les piles en *UNDERFLOW*, il suffit d'écrire :

$$X \in UNDERFLOW \wedge Y \in UNDERFLOW \Rightarrow X = Y$$

Si l'on veut nommer *crash* cette pile erroné, il suffit d'ajouter la constante *crash* dans la signature, et

de remplacer l'axiome généralisé précédent par :

$$X \in UNDERFLOW \quad \Rightarrow \quad X = crash$$

Si par contre on préfère récupérer les piles *UNDERFLOW* sur la pile vide, il suffit d'écrire :

$$X \in UNDERFLOW \quad \Rightarrow \quad X = empty$$

De même, si l'on veut faire tomber tous les cas d'*OVERFLOW* dans un *crash* irréversible, il suffit d'écrire :

$$X \in OVERFLOW \quad \Rightarrow \quad X = crash$$

Et si l'on veut récupérer l'application de *push* sur une pile pleine en n'effectuant pas ce *push*, il suffit d'écrire :

$$height(X)=Maxheight \quad \Rightarrow \quad push(x,X) = X$$

Bien d'autres traitements des cas exceptionnels peuvent être imaginés.

Chaînes bornées

Nous spécifions les chaînes bornées comme une présentation *STRING* au-dessus de *NAT* (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*. On notera *Maxlen* la longueur maximale des chaînes.

S = { *STRING* }

Σ =	λ :		$\rightarrow$	*STRING*	(chaîne vide)
	add :	*ELEM STRING*	$\rightarrow$	*STRING*	(ajoute un élément)
	concat :	*STRING STRING*	$\rightarrow$	*STRING*	(concaténation)
	length :	*STRING*	$\rightarrow$	*NAT*	(longueur)

L = { *STRING-TOO-LONG* }

Ok-Frm =

$$x_1 \in \text{Ok-Frm} \wedge \cdots \wedge x_{Maxlen} \in \text{Ok-Frm} \Rightarrow add(x_1, add(x_2, \ldots, add(x_{Maxlen}, \lambda) \in \text{Ok-Frm}$$
$$add(x,X) \in \text{Ok-Frm} \Rightarrow X \in \text{Ok-Frm}$$

Ok-Ax :

$$concat(\lambda,u) = u$$
$$concat(add(x,u),v) = add(x,concat(u,v))$$
$$length(\lambda) = 0$$
$$length(add(x,u)) = succ(length(u))$$

Lbl-Ax =

$$length(u)=Maxlen \Rightarrow add(x,u) \in STRING\text{-}TOO\text{-}LARGE$$
$$u \in STRING\text{-}TOO\text{-}LARGE \Rightarrow add(x,u) \in STRING\text{-}TOO\text{-}LARGE$$
$$length(u)+length(v) > Maxlen = True \Rightarrow concat(u,v) \in STRING\text{-}TOO\text{-}LARGE$$

En ce qui concerne les axiomes généralisés (**Gen-Ax**), comme dans le cas des piles, de nombreux traitement d'exceptions peuvent être imaginés. Par exemple on peut amalgamer toutes les chaînes trop longues au moyen de l'axiome généralisé suivant :

$$u \in STRING\text{-}TOO\text{-}LARGE \wedge v \in STRING\text{-}TOO\text{-}LARGE \Rightarrow u = v$$

Ensembles bornés

Nous spécifions les ensembles bornés comme une présentation *SET* au-dessus de *NAT+BOOL* (pour pouvoir spécifier les opérations “cardinal” et “élément de”), 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)). On notera *Maxcard* le nombre maximal d’éléments dans un ensemble.

S = { *SET* }

Σ =	$\emptyset$:		$\rightarrow$	*SET*	(ensemble vide)
	ins :	*ELEM SET*	$\rightarrow$	*SET*	(insertion)
	del :	*ELEM SET*	$\rightarrow$	*SET*	(enlever un élément)
	$\in$:	*ELEM SET*	$\rightarrow$	*BOOL*	(élément de)
	$\cup$:	*SET SET*	$\rightarrow$	*SET*	(union)
	$\cap$:	*SET SET*	$\rightarrow$	*SET*	(intersection)
	$-$:	*SET SET*	$\rightarrow$	*SET*	(différence)
	Δ :	*SET SET*	$\rightarrow$	*SET*	(différence symétrique)
	card :	*SET*	$\rightarrow$	*NAT*	(cardinal)

L = { *SET-OVERFLOW, ELEM-NOT-FOUND* }

Ok-Frm =

$x_1 \in$ Ok-Frm $\wedge \cdots \wedge$ $x_{Maxcard} \in$ Ok-Frm $\Rightarrow$ $ins(x_1, ins(x_2, \ldots, ins(x_{Maxcard}, \emptyset) \in$ Ok-Frm

ins(x,X) $\in$ Ok-Frm $\Rightarrow$ $X \in$ Ok-Frm

Ok-Ax : $ins(x,ins(x,X)) = ins(x,X)$

$ins(x,ins(y,X)) = ins(y,ins(x,X))$

$x \in \varnothing = False$

$x \in ins(x,X) = True$

$x \in ins(y,X) = eq(x,y) \vee x \in X$

$x \in X = False \Rightarrow del(x,ins(x,X)) = X$

$x \in X = True \Rightarrow del(x,ins(x,X)) = del(x,X)$

$eq(x,y) = False \Rightarrow del(x,ins(y,X)) = ins(y,del(x,X))$

$X \cup \varnothing = X$

$X \cup ins(x,Y) = ins(x,X \cup Y)$

$X \cap \varnothing = \varnothing$

$x \in X = True \Rightarrow X \cap ins(x,Y) = ins(x,X \cap Y)$

$x \in X = False \Rightarrow X \cap ins(x,Y) = X \cap Y$

$X - \varnothing = X$

$x \in X = False \Rightarrow X - ins(x,Y) = X - Y$

$x \in X = True \Rightarrow X - ins(x,Y) = del(x,X) - Y$

$X \Delta Y = X \cup Y - X \cap Y$

$card(\varnothing) = 0$

$x \in X = False \Rightarrow card(ins(x,X)) = succ(card(X))$

Lbl-Ax =

$card(X) = Maxcard \wedge x \in X = False \Rightarrow ins(x,X) \in SET\text{-}OVERFLOW$

$X \in SET\text{-}OVERFLOW \Rightarrow ins(x,X) \in SET\text{-}OVERFLOW$

$x \in X = False \Rightarrow del(x,X) \in ELEM\text{-}NOT\text{-}FOUND$

$card(X) + card(X - Y) > Maxcard = True \Rightarrow X \cup Y \in SET\text{-}OVERFLOW$

En ce qui concerne les axiomes généralisés, là encore, de nombreux traitement des cas exceptionnels peuvent être conçus. Citons par exemple la récuparation suivante des termes de la forme *del(x,X)* lorsque *x* n'est pas dans *X* :

$$del(x,X) \in ELEM\text{-}NOT\text{-}FOUND \Rightarrow del(x,X) = X$$

qui correspond à la spécification "classique" de l'opération *del*, lorsque l'on écrivait l'équation :

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

Files bornées

Nous spécifions les files (“fifo”) bornées comme une présentation *QUEUE* au-dessus de *NAT* (pour pouvoir spécifier l’opération “longueur”), et d’une spécification prédéfinie *ELEM* contenant une sorte *ELEM* (les éléments placés dans les files seront ceux de sorte *ELEM*). Naturellement, il est possible que *ELEM* soit en fait *NAT*. On notera *Maxlength* la longueur maximale d’une file.

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)

L = { *UNDERFLOW, OVERFLOW, QUEUE-IS-EMPTY* }

Ok-Frm =

$x_1 \in$ Ok-Frm $\wedge \cdots \wedge$ $x_{Maxlength} \in$ Ok-Frm $\Rightarrow$ $add(x_1, add(x_2, \ldots, add(x_{Maxlength}, new) \in$ Ok-Frm

add(x,X) ∈ Ok-Frm ⇒ *X* ∈ Ok-Frm

Ok-Ax :	*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))*

Lbl-Ax :	*first(new)* ∈ *QUEUE-IS-EMPTY*	
	remove(new) ∈ *UNDERFLOW*	
X ∈ *UNDERFLOW*	⇒	*remove(X)* ∈ *UNDERFLOW*
length(X)=Maxlength	⇒	*add(x,X)* ∈ *OVERFLOW*
X ∈ *OVERFLOW*	⇒	*add(x,X)* ∈ *OVERFLOW*

Des récupérations similaires au cas des piles bornées peuvent être spécifiées dans **Gen-Ax** ; se reporter à l’exception-spécification de *STACK* (annexe 3.6).

ANNEXE 4 :

EXEMPLES D'IMPLEMENTATIONS ABSTRAITES AVEC TRAITEMENT D'EXCEPTIONS

Cet annexe contient divers exemples simples d'implémentations abstraites axec traitement d'exceptions.

Pour les spécifier, nous suivons le formalisme d'implémentation dans le cadre des exception-algèbres, décrit en partie C. Nous nous conformons de plus aux notations décrites dans la partie A, chapitre II, à la fin de la section 2 (pages 49 et 50).

Annexe 4 : exemples d´implémentations abstraites avec exceptions

Annexe 4.1

Implémentation de *STACK*

par *ARRAY* et *NAT*

Pour spécifier l'implémentation des piles bornées au moyen des tableaux bornés et des entiers naturels bornés, nous considérons des tableaux indicés par des entiers naturels bornés (*RANG=NAT*), et les éléments mémorisés dans les tableaux (*ELEM*) sont ceux placés dans les piles. Naturellement, en suivant les méthodes de preuves de correction définies dans la partie C, chapitre V, on peut démontrer sans difficulté majeure que cette implémentation abstraite n'est correcte que si le rang maximal des tableaux (*Maxrange*) et la borne entière (*Maxint*) sont supérieures à la hauteur maximale des piles (*Maxheight*).

L'opération d'abstraction est :

$$<\_\,,\_> : \quad ARRAY \quad NAT \quad -implémente\rightarrow \quad STACK$$

Les formes *Ok* synthétisées peuvent être déclarées comme suit :

$$t \in \text{Ok-Frm} \ \wedge \ 0 \leq i \leq Maxheight = True \quad \Rightarrow \quad <t,i> \in \text{Ok-Frm}$$

La composante cachée est vide.

Les *Ok*-axiomes d'implémentation des opérations sont habituels :

$$\begin{aligned} empty &\approx <t,0> \\ push(n,<t,i>) &\approx <t[i]:=n,succ(i)> \\ pop(<t,succ(i)>) &\approx <t,i> \\ top(<t,succ(i)>) &\approx t[i] \\ height(<t,i>) &\approx i \end{aligned}$$

Les axiomes d'implémentation des étiquettes d'exception peuvent être spécifiés comme suit :

$$pop(<t,0>) \in UNDERFLOW$$
$$X \in UNDERFLOW \quad \Rightarrow \quad pop(X) \in UNDERFLOW$$
$$push(<t,Maxheight>) \in OVERFLOW$$
$$X \in OVERFLOW \quad \Rightarrow \quad push(x,X) \in OVERFLOW$$
$$top(<t,0>) \in STACK\text{-}IS\text{-}EMPTY$$

Les *Ok*-axiomes de représentation de l'égalité sont alors bien connus :

$$<t,0> \;=\; <t',0>$$
$$<t,i>=<t',i> \wedge t[i]=t'[i] \quad \Rightarrow \quad <t,succ(i)> = <t',succ(i)>$$

En ce qui concerne les axiomes généralisés d'implémentation des opérations et les axiomes généralisés de représentation de l'égalité, ceux-ci sont bien sûr dépendants du traitement des cas exceptionnels choisi dans *STACK* et *ARRAY*. Ils seront en particulier vides si aucun traitement exceptionnel n'est spécifié dans *STACK*. Et si l'on choisit de récupérer toutes les piles en *UNDERFLOW* sur la pile vide, il suffit d'ecrire l'axiome généralisé d'implémentation de *pop* suivant :

$$pop(<t,0>) \approx <t,0>$$

et l'axiome généralisé de représentation de l'égalité suivant :

$$X \in UNDERFLOW \quad \Rightarrow \quad X = <t,0>$$

Annexe 4.2

Implémentation de *STRING*

par *ARRAY* et *NAT*

Pour spécifier l'implémentation des chaînes bornées au moyen des tableaux bornés et des entiers naturels bornés, nous considérons des tableaux indicés par des entiers naturels bornés (*RANG=NAT*), et les éléments mémorisés dans les tableaux (*ELEM*) sont ceux consituant les "caracteres" des chaînes. Naturellement, en suivant les méthodes de preuves de correction définies dans la partie C, chapitre V, on peut démontrer sans difficulté majeure que cette implémentation abstraite n'est correcte que si le rang maximal des tableaux (*Maxrange*) et la borne entière (*Maxint*) sont supérieures à la longueur maximale des chaînes (*Maxlen*).

L'opération d'abstraction est :

$$<\_\,,\_> : \quad ARRAY\ NAT \quad -implémente\rightarrow \quad STRING$$

Les formes *Ok* synthétisées peuvent être déclarées comme suit :

$$t \in \text{Ok-Frm} \ \wedge\ 0 \leq i \leq Maxlen = True \quad \Rightarrow \quad <t,i> \in \text{Ok-Frm}$$

Nous utiliserons une composante cachée, afin de définir une opération qui transfert une portion de tableau dans un autre tableau :
l'opération cachée : *transfert(t,i,t',j,k)* a pour rôle de transférer les éléments du tableau *t'* compris entre les rangs *j* et *j+k* (*j+k* exclu), dans le tableau *t* à partir du rang *i* (*i* inclus).
Ceci est spécifié au moyen des *Ok*-axiomes cachés suivants :

$$\begin{aligned} transfert(t,i,t',j,0) &= t \\ transfert(t,i,t',j,succ(k)) &= transfert(t[i]:=t'[j],succ(i),t',succ(j),k) \end{aligned}$$

Les *Ok*-axiomes d'implémentation sont alors :

$$\begin{aligned} \lambda &\approx <t,0> \\ add(x,<t,i>) &\approx <t[i]:=x,succ(i)> \\ concat(<t,i>,<t',j>) &\approx <transfert(t,i,t',0,j),i+j> \\ length(<t,i>) &\approx i \end{aligned}$$

Les axiomes d'implémentation des étiquettes d'exception peuvent alors être spécifiés comme suit :

$$\begin{array}{rcl} & & add(x,<t,Maxlen>) \in STRING\text{-}TOO\text{-}LONG \\ s \in STRING\text{-}TOO\text{-}LONG & \Rightarrow & add(x,s) \in STRING\text{-}TOO\text{-}LONG \end{array}$$

Les *Ok*-axiomes de représentation de l'égalité peuvent être spécifiés comme suit :

$$<t,0> \; = \; <t',0>$$

$$<t,i>=<t',i> \; \wedge \; t[i]=t'[i] \quad \Rightarrow \quad <t,succ(i)> = <t',succ(i)>$$

Naturellement, puisqu'aucun traitement d'exceptions n'est prévu dans notre spécification des chaînes, aucun axiome généralisé d'implémentation des opérations ou de représentation de l'égalité n'est nécessaire ici.

Annexe 4.3

Implémentation de *SET*

par *STRING*

Pour spécifier l'implémentation des ensembles bornés au moyen des chaînes bornées, nous considérons des ensembles et des chaînes d'éléments de sorte *ELEM* quelconque. Naturellement, en suivant les méthodes de preuves de correction définies dans la partie C, chapitre V, on peut démontrer sans difficulté que cette implémentation abstraite n'est correcte que si la longueur maximale des chaînes (*Maxlen*) est supérieure au cardinal maximal des ensembles (*Maxcard*).

L'opération d'abstraction est :

$$<\_> : \quad STRING \quad -implémente\rightarrow \quad SET$$

Les forme *Ok* synthétisées peuvent être déclarées comme suit :

$$length(s) \leq Maxcard = True \wedge s \in \text{Ok-Frm} \quad \Rightarrow \quad <s> \in \text{Ok-Frm}$$

Nous spécifions une composante cachée qui enrichit la spécification *STRING* des opérations *occurs*, *remove* et *delete*.
L'opération *occurs(x,s)* retourne *True* si l'élément *x* est dans la chaîne *s*, et *False* sinon. L'opération *remove(x,s)* retourne la chaîne *s* privée de sa premiere occurrence (éventuelle) de l'élément *x* ; en particulier, si *x* n'est pas dans *s*, alors la chaîne *s* elle même est retournée (*remove* ne correspond donc pas directement à l'opération ensembliste *del* car elle ne retourne aucune erreur dans ce cas). Enfin l'opération *delete(d,s)* retourne la chaîne *s* privée de la première première occurrence (éventuelle) de chacun des éléments contenus dans la chaîne *d* ; en particulier, si *d* ne contient aucun élément en commun avec *s* alors la chaîne *s* est elle même retournée, et si *d* contient tous les éléments de *s* (dans n'importe quel ordre) alors la chaîne vide est retournée.
Ceci est obtenu par les *Ok*-axiomes cachés suivants :

$$
\begin{array}{rcrcl}
 & & occurs(x,\lambda) & = & False \\
 & & occurs(x,add(y,s)) & = & eq(x,y) \vee occurs(x,s) \\
 & & remove(x,\lambda) & = & \lambda \\
 & & remove(x,add(x,s)) & = & s \\
eq(x,y)=False & \Rightarrow & remove(x,add(y,s)) & = & add(y,remove(x,s)) \\
 & & delete(\lambda,s) & = & s \\
 & & delete(add(x,d),s) & = & delete(d,remove(x,s))
\end{array}
$$

Les *OK*-axiomes d'implémentation des opérations sont alors les suivants :

$$
\begin{array}{rcrcl}
 & & \varnothing & \approx & <\lambda> \\
occurs(x,s)=False & \Rightarrow & ins(x,<s>) & \approx & <add(x,s)> \\
occurs(x,s)=True & \Rightarrow & ins(x,<s>) & \approx & s \\
occurs(x,s)=True & \Rightarrow & del(x,<s>) & \approx & <remove(x,s)> \\
 & & x \in <s> & \approx & occurs(x,s) \\
 & & <s> \cup <s'> & \approx & concat(delete(s,s'),s) \\
 & & <s> \cap <s'> & \approx & delete(s,delete(s',s)) \\
 & & <s> - <s'> & \approx & delete(s',s) \\
 & & <s> \Delta <s'> & \approx & concat(delete(s',s),delete(s,s')) \\
 & & card(<s>) & \approx & length(s)
\end{array}
$$

Les axiomes d'implémentation des étiquettes d'exception peuvent être spécifiés comme suit :

$$
\begin{array}{rcl}
length(s)=Maxcard \wedge occurs(x,s)=False & \Rightarrow & ins(x,<s>) \in SET\text{-}OVERFLOW \\
X \in SET\text{-}OVERFLOW & \Rightarrow & ins(x,X) \in SET\text{-}OVERFLOW \\
occurs(x,s)=False & \Rightarrow & del(x,<s>) \in ELEM\text{-}NOT\text{-}FOUND \\
length(s)+length(delete(s,s')) > Maxcard = True & \Rightarrow & <s>\cup<s'> \in SET\text{-}OVERFLOW
\end{array}
$$

L'*Ok*-axiome de représentation de l'égalité est particulièrement simple :

$$<add(x,add(y,s))> = <add(y,add(x,s))>$$

il traduit la commutativité de l'insertion ensembliste.

Annexe 4.4

Implémentation de *SET*

par *ARRAY*

Rappelons que nous avons défini la présentation *ARRAY* au-dessus d'une spécification *RANG* quelconque (munie d'un prédicat d'égalité et une relation d'ordre), et d'une spécification *ELEM* quelconque.

Nous allons considérer des tableaux de booléens, et la spécification *RANG* sera instanciée par un type "intervalle d'entiers" [*min,max*] (rappelons qu'avec traitement d'exceptions, spécifier un tel intervalle ne présente aucune difficulté).
Nous allons implémenter des ensembles (*SET*) dont les éléments seront des éléments de cet intervalle [*min,max*]. Un ensemble *s* sera donc implémenté par un tableau *t* de telle sorte que *n* (de sorte *RANG=[min,max]*) sera élément de *s* si et seulement si *t[r]* égale *True*. Dans ce cas, il est clair qu'aucun cas de "*SET-OVERFLOW*" ne peut être atteint, puisque tous les éléments *Ok* (i.e. ceux compris entre *min* et *max*) peuvent être en même temps élément de l'ensemble implémenté. Le seul cas d'erreur rencontré sera donc une application de l'opération ensembliste *del* alors que l'élément à enlever n'est pas dans l'ensemble.

L'opération d'abstraction est bien sûr :

$$<\_> : \quad ARRAY \quad -implémente\rightarrow \quad SET$$

Les formes *Ok* synthétisées peuvent être déclarées comme suit :

$$t \in \text{Ok-Frm} \quad \Rightarrow \quad <t> \in \text{Ok-Frm}$$

Nous utilisons une composante cachée afin de définir les opérations suivantes :

- *init* sera une constante de type tableau contenant uniformément la valeur *False* entre *min* et *max* (rappelons que le tableau *create* n'est initialisé en aucun rang).
- *OR* sera une opération effectuant le "ou" logique global de deux tableaux (rang par rang). Cette opération crée bien sûr des termes exceptionnels de sorte *ARRAY* chaque fois qu'elle est appliquée à des tableaux contenant un rang non initialisé entre *min* et *max*.
- *AND* sera une opération effectuant le "et" logique global de deux tableaux, selon les mêmes conditions d'application que le *OR* précédent.

□ enfin Δ sera une opération effectuant la différence symétrique de deux tableaux, selon les mêmes conditions d'application que précédemment.

Les *Ok*-axiomes cachés spécifiants ces opérations cachées sont :

$$\begin{aligned}
init &= (..((create[min]:=False)[succ(min)]:=False)..)[max]:=False \\
init\ AND\ t &= init \\
t'[r]:=True\ \ AND\ \ t &= (t'\ AND\ t)[r] := t[r] \\
init\ OR\ t &= t \\
t'[r]:=True\ \ OR\ \ t &= (t'\ OR\ t)[r] := True \\
init\ \Delta\ t &= t \\
t'[r]:=True\ \ \Delta\ \ t &= (t'\ \Delta\ t)[r] := \neg\ t[r]
\end{aligned}$$

Les axiomes d'étiquetage cachés, relatifs à ces opérations, sont :

$$\begin{aligned}
t[r] \in NOT\text{-}INITIALIZED &\Rightarrow t'\ AND\ t \in NOT\text{-}INITIALIZED \\
t'[r] \in NOT\text{-}INITIALIZED &\Rightarrow t'\ AND\ t \in NOT\text{-}INITIALIZED \\
t[r] \in NOT\text{-}INITIALIZED &\Rightarrow t'\ OR\ t \in NOT\text{-}INITIALIZED \\
t'[r] \in NOT\text{-}INITIALIZED &\Rightarrow t'\ OR\ t \in NOT\text{-}INITIALIZED \\
t[r] \in NOT\text{-}INITIALIZED &\Rightarrow t'\ \Delta\ t \in NOT\text{-}INITIALIZED \\
t'[r] \in NOT\text{-}INITIALIZED &\Rightarrow t'\ \Delta\ t \in NOT\text{-}INITIALIZED
\end{aligned}$$

ces axiomes signifient intuitivement que, par exemple pour le premier axiome, *s'il existe un rang r tel que t n'est pas initialisé, alors (t' AND t) répond au même diagnostic*.

Les *Ok*-axiomes d'implémentation peuvent alors être spécifiés comme suit :

		$\emptyset$	$\approx$	$init$
		$ins(r,<t>)$	$\approx$	$t[r]:=True$
$t[r]=True$	$\Rightarrow$	$del(r,<t>)$	$\approx$	$t[r]:=False$
		$r \in <t>$	$\approx$	$t[r]$
		$<t> \cup <t'>$	$\approx$	$t\ OU\ t'$
		$<t> \cap <t'>$	$\approx$	$t\ AND\ t'$
		$<t> - <t'>$	$\approx$	$(t\ \Delta\ t')\ AND\ t$
		$<t> \Delta <t'>$	$\approx$	$t\ \Delta\ t'$
		$card(<init>)$	$\approx$	0
$t[r]:=False$	$\Rightarrow$	$card(<t[r]:=True>)$	$\approx$	$succ(card(<t>))$

Remarquons que les valeurs synthétisées atteintes par les opérations ensemblistes sont celles de la forme <*t*> telles que *t* est partout initialisé entre *min* et *max*.

Les axiomes d'implémentation des étiquettes d'exception peuvent être spécifiés comme suit :

$$t[r]=False \Rightarrow del(r,<t>) \in ELEM\text{-}NOT\text{-}FOUND$$

La représentation de l'égalité est vide puisque deux tableaux ne représentent le même ensemble que s'ils sont égaux.

Enfin, on peut imaginer la récupération suivante, spécifiée dans *SET* :

$$r \in S = False \Rightarrow del(r,S) = S$$

qui se traduit au niveau des axiomes généralisés de l'implémentation par :

$$t[r]=False \Rightarrow del(r,<t>) \approx t$$

Annexe 4.5

Implémentation de *QUEUE*

par *ARRAY* et *NAT*

Récapitulons ici cette implémentation circulaire des files bornées au moyen de tableaux bornés, déjà spécifiée au cours de la partie C.

L'opération d'abstraction est la suivante :

$$<\_ ,\_ ,\_ > : \quad ARRAY \quad NAT \quad NAT \quad -implémente\rightarrow \quad QUEUE$$

Les formes *Ok* synthétisées sont déclarées comme suit :

$$t \in \text{Ok-Frm} \wedge i \in \text{Ok-Frm} \wedge j \in \text{Ok-Frm} \quad \Rightarrow \quad < t, i, j > \in \text{Ok-Frm}$$

Nous utilisons une composante cachée pour définir une opération *Next* telle que *Next(i)* égale *i modulo succ(Maxlength)* (rappelons que *Maxlength* est la longueur maximale d'une file.

$$Next(n) \quad = \quad succ(n) \; - \; (succ(Maxlength) \times (succ(n) \; div \; succ(Maxlength)))$$

Les *Ok*-axiomes d'implémentation sont les suivants :

$$\begin{aligned}
i \leq Maxlength = True \quad &\Rightarrow \quad new \approx <t,i,i> \\
&\quad add(e,<t,i,j>) \approx <t[j]:=e,i,Next(j)> \\
eq(i,j) = False \quad &\Rightarrow \quad remove(<t,i,j>) \approx <t,Next(i),j> \\
eq(i,j) = False \quad &\Rightarrow \quad first(<t,i,j>) \approx t[i] \\
&\quad length(<t,i,j>) \approx Next((j+Maxlength)-i)
\end{aligned}$$

Les axiomes d'implémentation des étiquettes d'exception peuvent être spécifiés comme suit :

$$\begin{aligned}
Next(j) = i \quad &\Rightarrow \quad add(x,<t,i,j>) \in OVERFLOW \\
X \in OVERFLOW \quad &\Rightarrow \quad add(x,X) \in OVERFLOW \\
i = j \quad &\Rightarrow \quad remove(<t,i,j>) \in UNDERFLOW \\
X \in UNDERFLOW \quad &\Rightarrow \quad remove(X) \in UNDERFLOW \\
i = j \quad &\Rightarrow \quad first(<t,i,j>) \in QUEUE\text{-}IS\text{-}EMPTY
\end{aligned}$$

Les axiomes généralisés d'implémentation sont évidemment directement dépendants des récupérations spécifiées dans *QUEUE* ; se référer à l'exemple 5, partie C, chapitre II, section 6

(page 183).

Enfin les *OK*-axiomes de représentation de l'égalité sont :

$$<t,i,i> \;=\; <t',k,k>$$

$$<t,i,j> = <t',k,l> \wedge t[j]=t'[l] \;\Longrightarrow\; <t,i,Next(j)> = <t',k,Next(l)>$$

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[ADJ 76] **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. Egalement : IBM Report RC 6487, Oct. 1976.

[ADJ 80] **Ehrig H., Kreowski H., Thatcher J., Wagner J., Wright J. :** "*Parameterized data types in algebraic specification langages*". Proc. 7th ICALP, July 1980.

[BBC 85] **Bernot G., Bidoit M., Choppy C.:** "*Abstract data types with exception handling : an initial approach based on a distinction between exceptions and errors*". A paraître dans Theoretical Computer Science (1986). Egalement : Rapport de recherche 251, LRI, Orsay, Décembre 1985.

[BBC 86] **Bernot G., Bidoit M., Choppy C. :** "*Abstract implementations and correctness proofs*". Proc. 3rd STACS, January 1986, Springer-Verlag LNCS 210, January 1986. Egalement : Rapport de recherche 250, LRI, Orsay, Décembre 1985.

[BC 85] **Bidoit M., Choppy C. :** "*ASSPEGIQUE : an integrated environment for algebraic specifications*". Formal Methods and Software Developments, Proc. International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin, Mars 1985, Vol. 2 : Colloquium on Software Engineering (CSE), (H. Ehrig, C. Floyd, M. Nivat, J. Thatcher, eds), LNCS 186, Springer Verlag, pp 246-260 Egalement : Rapport de recherche 203, LRI, Orsay, Jan. 1985.

[BCV 85] **Bidoit M., Choppy C., Voisin F. :** "*The ASSPEGIQUE specification environment, Motivations and design*". Proc. of the 3rd Workshop on Theory and Applications of Abstract data types, Bremen, Nov 1984, Recent Trends in Data Type Specification (H.-J. Kreowski ed.), Informatik-Fachberichte 116, Springer Verlag, Berlin-Heidelberg, 1985. Egalement : Rapport de recherche 239, LRI, Orsay, Oct. 1985.

[Ber 84] **Bernot G. :** "*Implémentations de types abstraits algébriques en présence d'exceptions*". Rapport de DEA, LRI, Orsay, Sept. 1984.

[Ber 86] **Bernot G. :** "*Interprétation de quelques résultats élémentaires de la théorie des catégories dans le cadre des types abstraits algébriques*". En préparation.

[BG 83] **Bidoit M., Gaudel M-C. :** "*Spécification des cas d'exceptions dans les types abstraits algébriques: problèmes et perspectives*". Rapport de recherche 146, LRI, Orsay, 1983.

[Bid 82] **Bidoit M. :** "*Algebraic data types: structured specifications and fair presentations*". Proc. AFCET Symposium on Mathematics for Computer Science, Paris, March 1982.

[Bid 84] **Bidoit M. :** "*Algebraic specification of exception handling by means of declarations and equations*". Proc. 11th ICALP, Springer-Verlag LNCS 172, July 1984.

[BW 82] **Broy M., Wirsing M. :** "*Partial abstract data types*". Acta Informatica, Vol.18-1, Nov 1982.

[Cap 86] **Capy F. :** "*Etude des problèmes liés à la consistance et à l'évaluation symbolique dans les E,R-algèbres*". Rapport de DEA, LRI, Orsay, Sept. 1985.

[EKMP 80] **Ehrig H., Kreowski H., Mahr B., Padawitz P. :** "*Algebraic implementation of abstract data types*". Theoretical Computer Science, Oct. 1980.

[EKP 80] **Erig H., Kreowski H., Padawitz P. :** "*Algebraic implementation of abstract data types: concept, syntax, semantics and correctness*". Proc. ICALP, Springer-Verlag LNCS 85, 1980.

[EPE 81] **Engels G., Pletat V., Ehrich H. :** "*Handling errors and exceptions in the algebraic specification of data types*". Osnabruecker Schriften zur Mathematik, July 1981.

[Gau 78] **Gaudel M-C. :** "*Spécifications incomplètes mais suffisantes de la représentation des types abstraits*". Rapport Laboria 320, 1978.

[Gau 80] **Gaudel M-C. :** "*Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation*". Thèse d'état, Nancy, Mars 1980.

[GDLE 84] **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.

[GHM 76] **Guttag J.V., Horowitz E., Musser D.R. :** "*Abstract data types and software validation*". C.A.C.M., Vol 21, n.12, 1978. Egalement : USG ISI Report 76-48.

[Gog 77] **Goguen J.A. :** "*Abstract errors for abstract data types*". Formal Description of Programming Concepts, E.J. NEUHOLD Ed., North Holland, New York 1977.

[GP 79] **Gaudel M-C., Pair C. :** "*Construction de compilateurs basés sur une sémantique formelle*". Actes des journées francophones sur la certification du logiciel, Genève, 1979, pp. 83-101.

[GTW 78] **Goguen J.A., Thatcher J.W., Wagner E.G. :** "*An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*". Current Trends in Programming Methodology, (Yeh ed.), Printice-Hall, vol.IV, p.80, (1978).

[Gut 79] **Guttag J.V. :** "*Notes on type abstraction (Version 2)*". IEEE Transactions on Software Engineering, 1979.

[Hes 85] **Hesselink W.H. :** "*Nondeterminism in data types, a mathematical approach*". A paraître dans TOPLAS, ACM, 1986. Egalement : Computing Science Notes 8506, Groningen University, 1985.

[Hoa 72] **Hoare C.A.R. :** "*Proof of Correctness of Data Representations*". Acta Informatica, vol.1, fasc.4, p.271, (1972).

[Kam 80] **Kamin S. :** "*Final data type specifications : a new data type specification method*". Proc. of the 7th POPL Conference, 1980.

[Loe 81] **Loeckx J. :** "*Algorithmic specifications of abstract data types*". Proc. 8th ICALP 1981.

[McL 71] **Mac Lane S. :** "*Categories for the working mathematician*". Graduate texts in mathematics, 5, Springer-Verlag, 1971.

[Pla 82] **Plaisted D. :** "*An initial algebra semantics for error presentations*". Unpublished Draft, 1982.

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

NOM : BERNOT

PRENOM : Gilles

TITRE : *Une sémantique algébrique pour une spécification différenciée des exceptions et des erreurs ; application à l'implémentation et aux primitives de structuration des spécifications formelles.*

RESUME : Le but de cette thèse est de présenter un nouveau formalisme de traitement d'exceptions dans le cadre des types abstraits algébriques, et de l'utiliser pour traiter l'implémentation abstraite en présence d'exceptions.
La première partie développe une nouvelle sémantique pour l'implémentation abstraite, et permet d'exprimer la correction d'une implémentation en terme de *suffisante complétude* et *consistance hiérarchique*. Ainsi les preuves de correction d'une implémentation abstraite peuvent être traitées par des méthodes classiques telles que les techniques de réécriture ou d'induction structurelle. L'idée majeure de cette approche repose sur une distinction fondamentale entre spécifications *descriptives* et spécifications *constructives*. Des conditions simples et peu restrictives sont fournies pour que la *composition* d'implémentations correctes reste correcte.
La seconde partie développe un nouveau formalisme de traitement d'exceptions : les *exception-algèbres*. Ce formalisme autorise toutes les formes de traitement d'exceptions (messages d'erreur, propagation implicite des exceptions et des erreurs, récupérations d'exceptions), tout en préservant l'existence des modèles initiaux et une approche fonctorielle simple. Nous définissons en particulier une sémantique fonctorielle des *enrichissements*, munie des notions de *consistance hiérarchique* et de *suffisante complétude*.
Plus généralement, la plupart des primitives de structuration des spécifications algébriques peuvent être étendues sans difficulté aux exception-algèbres car les résultats fondamentaux relatifs aux exception-algèbres sont analogues à ceux des types abstraits algébriques "classiques". La troisième partie démontre en particulier que le formalisme d'implémentation abstraite peut être étendu aux exception-algèbres sans difficulté.
Plusieurs exemples d'exception-spécifications et d'implémentations abstraites sont donnés en annexe.

MOTS CLES : spécifications algébriques, traitement d'exceptions, types abstraits algébriques, spécifications hiérarchiques, implémentation abstraite.