UNIVERSITÉ DE NANCY I
UER SCIENCES MATHÉMATIQUES

CENTRE DE RECHERCHE EN
INFORMATIQUE DE NANCY

THESE

POUR L'OBTENTION DU
DOCTORAT DE 3ÈME CYCLE EN INFORMATIQUE

OUTILS D'AIDE A LA CONSTRUCTION ET TRANSFORMATION DE TYPES ABSTRAITS ALGEBRIQUES

SOUTENUE PUBLIQUEMENT LE 12 JUILLET 1984

par

NICOLE LEVY

MEMBRES DU JURY :

PRESIDENT ..... J.P. FINANCE

EXAMINATEURS ..... M.C. GAUDEL
P. MARCHAND
F. OREJAS
A. QUERE

UNIVERSITÉ DE NANCY I
UER SCIENCES MATHÉMATIQUES

CENTRE DE RECHERCHE EN
INFORMATIQUE DE NANCY

THESE

POUR L'OBTENTION DU
DOCTORAT DE 3ÈME CYCLE EN INFORMATIQUE

OUTILS D'AIDE A LA CONSTRUCTION ET TRANSFORMATION DE TYPES ABSTRAITS ALGEBRIQUES

SOUTENUE PUBLIQUEMENT LE 12 JUILLET 1984

par

NICOLE LEVY

MEMBRES DU JURY :

PRESIDENT	.....	J.P. FINANCE
EXAMINATEURS	.....	M.C. GAUDEL
		P. MARCHAND
		F. OREJAS
		A. QUERE

A Célia

Je suis très heureuse de pouvoir remercier ici tous ceux qui m'ont aidée à mener ce travail :

Jean-Pierre FINANCE, Professeur à l'Université de Nancy, qui m'a accueillie dans son équipe, m'a guidée, conseillée et soutenue dans ma recherche. Je tiens à lui exprimer toute ma reconnaissance.

Marie-Claude GAUDEL, Professeur à l'Université de Paris-Sud, pour l'intérêt qu'elle a porté à cette étude, pour ses nombreux conseils et pour sa gentillesse.

Pierre MARCHAND, Professeur à l'Université de Nancy, qui a bien voulu prendre sur son temps pour étudier cette thèse et participer à ce jury.

Fernando OREJAS, Professeur à l'Université de Barcelone, qui m'a apporté une aide précieuse et encouragée à poursuivre dans la voie des types abstraits.

Alain QUERE, Maître Assistant à l'Université de Nancy, initiateur du projet SPES auquel il m'a associée.

Je tiens également à remercier tous les membres de l'équipe Spécification et Programmation et en particulier Eric DUBOIS pour la fructueuse collaboration que nous avons eue,

Jeanine SOUQUIERES qui nous a rejoints dans le projet et y apporte tout son dynamisme.

Je ne puis citer tous les membres du CRIN qui m'ont aidée et qui créent une ambiance de travail si chaleureuse.

Ce rapport a pu être aussi bien présenté grâce à la compétence et au sourire de Marie-Thérèse DRIQUERT.

Enfin sans le soutien quotidien de Jean-Jacques, ce travail n'aurait pas été.

TABLE DES MATIÈRES

INTRODUCTION

INTRODUCTION

Un logiciel informatique est vivant. Les données de notre environnement se modifient en fonction du temps. Le logiciel, qui gère ces données, doit évoluer de même, sous peine de mourir. Son cycle de vie est caractérisé par deux périodes principales : la construction et la maintenance. La construction est divisée en plusieurs étapes, de la rédaction du cahier des charges au codage en passant par la spécification formelle et la programmation.

A chacune de ces étapes il peut être nécessaire de remettre en cause et de modifier les résultats de l'étape précédente.

De nombreuses études ont porté sur la transformation de programmes. En revanche, l'essentiel des transformations concernant les types de données a porté sur l'implémentation.

Parmi les formalismes utilisés pour décrire des spécifications formelles de types, celui des types abstraits algébriques a connu un grand essor depuis une douzaine d'années. Malgré une utilisation encore relativement limitée pour la spécification de problèmes effectifs, la diversité des thèmes abordés montre bien la généralité du pouvoir expressif des types abstraits.

Dans ce travail, nous avons cherché à définir des mécanismes de transformation de types abstraits algébriques, permettant de les optimiser ou de faciliter leur évolution.

Nous nous sommes tout d'abord appliqués à définir un cadre formel pour donner aux types abstraits algébriques une sémantique rigoureuse.

Nous nous sommes ensuite dotés d'un langage d'expression, et de constructeurs de types, ceux-ci étant des types paramétrés.

Enfin, nous avons utilisé la méthode de construction de spécifications développée au CRIN dans le cadre du projet SPES. Le projet SPES a pour but de proposer un langage, un guide méthodologique et des outils informatisés d'aide à la construction et à la transformation de spécifications formelles de problèmes.

La construction d'une spécification se fait de manière descendante et déductive suivant une approche fonctionnelle du problème. L'analyse descen-

dante peut introduire des redondances, soit au niveau des fonctions définies, soit au niveau des structures de données.

Transformer une spécification doit permettre d'optimiser certaines fonctions, d'éliminer la redondance et de permettre l'évolution des structures de données.

Les transformations sont dans notre travail des représentations de types. Nous sommes ainsi assurés que la spécification obtenue par transformation est encore une description formelle du problème initial.

Il s'agit ensuite de déterminer les transformations à appliquer à la spécification. Nous avons donc besoin de critères d'analyse permettant de mettre à jour les défauts d'une spécification. Les transformations sont choisies parmi un ensemble de constructeurs de types utilisés dans la spécification fonctionnelle.

La dernière étape du mécanisme de transformation des types abstraits algébriques sera d'appliquer les transformations choisies.

Ce document est divisé en trois parties :

Dans la première partie nous rappelons les concepts algébriques, définissons les types abstraits algébriques ainsi que les types paramétrés et leurs représentations. Nous démontrons les propriétés essentielles assurant la correction de notre démarche. Nous n'utilisons pas la théorie des catégories.

Dans la deuxième partie nous construisons une spécification fonctionnelle d'un exemple jouet. Nous définissons le langage SPES-TYPES permettant de présenter une telle spécification en termes de types abstraits algébriques. Celle-ci est alors appelée un univers. Les types sont construits à l'aide de constructeurs de types dont nous définissons un ensemble. Nous montrons ensuite comment construire un univers en parallèle d'une spécification fonctionnelle. La définition des types d'un univers se faisant par étapes successives, chaque nouvelle version est dite "issue" de la précédente.

Dans la troisième partie nous abordons les transformations.

Après en avoir précisé le concept, nous en définissons un ensemble correspondant à l'ensemble des constructeurs de types donné dans la deuxième partie. Parmi les transformations nous considérons les transformations de

structure, de généralisation, de simplification et de développement.

Puis nous indiquons des critères d'analyse d'un univers et enfin nous montrons à l'aide de l'exemple jouet comment transformer un univers.

Nous développons notre approche sur un exemple de spécification fonctionnelle : un système d'aide à la construction de spécifications en SPES-TYPES.

La première partie peut paraître formelle au regard des deux parties suivantes. Mais elle nous a paru utile parce qu'elle donne les éléments qui nous permettent de justifier la correction de notre approche.

RAPPELS THEORIQUES SUR LES TYPES ABSTRAITS ALGEBRIQUES

I - RAPPELS THÉORIQUES SUR LES TYPES ABSTRAITS ALGÉBRIQUES

1.- TYPES ABSTRAITS ALGEBRIQUES

1.1.- Approche intuitive

Le concept de type abstrait est issu de deux notions :

- la modularité ([PAR, 72]) qui apparaît sous la forme de "CLASSE" dans le langage SIMULA 67. Elle est obtenue en regroupant les descriptions d'objets et les opérations portant sur eux.
- l'abstraction, obtenue par la complète indépendance entre la description des données et leur implantation.

En 1972, HOARE [HOA, 72] les introduit comme méthode de simplification de preuves de correction de programmes :

"In the development of programs by stepwise refinement, the programmer is encouraged to postpone the decision on the representation of his data until after he has designed his algorithm, and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for the abstract data some convenient and efficient concrete representation in the store of a computer ; and finally programs the primitive operations required by his abstract program in terms of this concrete representation.

If the data representation is proved correct, the correctness of the final concrete program depends only on the correctness of the original abstract program."

Aujourd'hui la terminologie a évolué : les données abstraites sont des types abstraits de données et le programme abstrait une opération définie sur des types. A partir de ces idées de base, plusieurs approches ont été développées. L'approche algébrique a connu des développements très importants. Elle propose de décrire les types de données en termes d'algèbres. C'est-à-dire d'ensembles d'objets et d'opérations ou relations entre ces objets.

L'ensemble des objets est décrit à l'aide d'opérations appelées constructeurs des objets du type qui décrivent un chemin (ou un ensemble de chemins) permettant de parcourir tous les objets du type.

Prenons pour exemple l'ensemble des entiers naturels. Pour décrire un chemin reliant tous les entiers, il faut choisir un point de départ (ou constante) : ici 0 (zéro), et une opération de progression sur un "chemin" : ici "plus un" ou successeur noté suc. En effet, en partant de zéro et en ajoutant un à chaque étape, nous parcourons l'ensemble des entiers. Pour arriver par exemple à l'entier 4 il aura fallu ajouter quatre fois un à zéro, ce qui s'écrit

suc(suc(suc(suc(zéro)))).

Ce terme est appelé la forme normale de l'entier 4.

La théorie des types abstraits algébriques permet de définir des ensembles dont on puisse décrire un chemin passant par tous ses objets. Un tel ensemble est dit finiment engendré (ou minimal).

Dans l'exemple des entiers ci-dessus, nous avons essayé de décrire une formalisation d'un ensemble d'objets connus. Mais à une même description formelle peuvent correspondre plusieurs ensembles. Une interprétation pourrait être celle ne considérant qu'un seul et unique objet, le chemin nous ramenant toujours sur nos pas. En effet si rien ne nous permet de dire que suc(zéro) est égal à zéro, rien ne nous permet non plus d'affirmer le contraire. Il faut alors décider arbitraitement de deux objets dont on ne peut démontrer ni qu'ils sont égaux ni qu'ils sont différents

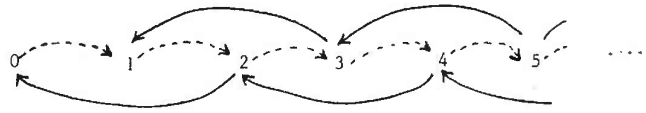
- soit leur différence c'est l'approche dite initiale [ADJ, 78].
- soit leur égalité : c'est l'approche dite terminale [KAM, 75].
- soit une approche intermédiaire [WIR - SAN, 83].

Pour ce travail, nous nous plaçons dans l'approche initiale, celle-ci nous paraissant plus proche de l'idée intuitive que nous avons des types de données. En effet, dans l'approche initiale, les termes engendrés par les constructeurs d'objets sont interprétés chacun comme un objet différent du type défini.

Mais un type n'est pas seulement un ensemble d'objets : ce sont les opérateurs qui le caractérisent. Nous avons vu que les constructeurs

décrivent un chemin parmi les objets. Les modificateurs sont des opérations permettant de relier les objets par "des chemins détournés".

Revenons à l'exemple des entiers, l'opération "moins-deux" relie les entiers selon le graphe ci-dessous : (les flèches ⤻ sont les relations établies par l'opération suc tandis que les flèches ⤳ sont celles établies par moins-deux).



(remarquons que moins-deux n'est pas défini sur tous les objets : que dire de moins-deux de zéro ou de un ?).

Les observateurs sont des opérations permettant de comparer les objets d'un type en terme d'objets d'un autre type défini par ailleurs.

Par exemple le prédicat d'égalité noté eq, à valeur dans le type des booléens ne contenant que deux objets : vrai et faux, nous permet de comparer deux "chemins" :

eq(moins-deux(suc(suc(zéro))),zéro) = vrai

eq(moins-deux(suc(suc(zéro))),suc(zéro)) = faux.

Les modificateurs et les observateurs sont définis à l'aide d'équations.

Le modificateur moins-deux est ainsi défini par
moins-deux(suc(suc(x))) = x pour tout objet x de type entier.

Pour avoir une définition complète de l'opération moins-deux il faut dire ce qui passe sur les objets zéro et suc(zéro). Ici encore plusieurs méthodes ont été proposées ([BID, GAU, 83] présente une étude comparative de différentes méthodes de spécification des cas d'exceptions). Nous choisissons d'introduire un objet appelé indéfini noté indef.

moins-deux (zéro) = indef
moins-deux (suc(zéro)) = indef
indef étant un objet de type entier, il faut aussi préciser les

chemins "suc" et "moins-deux" partant de lui :

moins-deux(indef) = indef :

suc (indef) = indef

Parfois l'ensemble des objets engendrés par les constructeurs est trop vaste, et on aimerait pouvoir se limiter aux objets vérifiant une propriété. Par exemple ne garder que les entiers inférieurs ou égaux à 10.

Nous appelons cette propriété un invariant.

Ainsi, de manière intuitive un type abstrait de données est défini par la donnée :

- d'un ensemble d'opérations parmi lesquelles il y a des constructeurs, des modificateurs et des observateurs, et
- d'un invariant.

D'un point de vue formel il est nécessaire de différencier le niveau syntaxique du niveau sémantique : la spécification d'un type et le modèle choisi.

C'est ce que nous étudions maintenant.

1.2.- Signatures, Algèbres, Spécification et Types Abstraits Algébriques

Nous rappelons ici certaines définitions et résultats désormais classiques. Nous nous sommes basés sur les travaux de J.L. REMY [REM, 82] et de M. BIDOIT [BID, 81].

1.2.1.- Définitions de base

Définitions :

Une signature $<S,\Sigma>$ est la donnée d'un ensemble S de sortes (noms de domaines) et d'une famille de symboles d'opérations $\Sigma = (\Sigma w,s)$, $w \in S^*$, $s \in S$, où S^* est l'ensemble des suites finies sur S (on note $\wedge$ la suite vide).

Le profil d'un symbole est le couple formé de son domaine et de sa sorte :

$\forall f \in \Sigma_{w,s}$ profil(f) = (w,s) que l'on note $f : s \leftarrow w$

domaine(f) = w

sorte(f) = s

La longueur de w est appelée l'arité de f. Un symbole d'arité nulle ($w = \wedge$) est une constante.

Exemple de signature :

$S = \{BOOL\}$

$\Sigma = \Sigma_{\wedge,BOOL}, \Sigma_{BOOL,BOOL}, \Sigma_{BOOL\ BOOL,BOOL}$

$\Sigma_{\wedge,BOOL} = \{vrai, faux\}$

$\Sigma_{BOOL,BOOL} = \{non\}$

$\Sigma_{BOOL\ BOOL,BOOL} = \{et, ou, eq\}$

que nous écrirons

vrai : BOOL ←

faux : BOOL ←

non : BOOL ← BOOL

et : BOOL ← BOOL,BOOL

ou : BOOL ← BOOL,BOOL

eq : BOOL ← BOOL,BOOL

impl : BOOL ← BOOL,BOOL

Définition : Soit $<S,\Sigma>$ une signature. Une $<S,\Sigma>$-algèbre ou Σ-algèbre, est la donnée

- d'une famille d'ensembles, indicée par S : $A = (A_s)_{s \in S}$ appelée support de l'algèbre, et
- d'une famille d'opérations dans A indicée par Σ : $\Sigma_A = (f^A)_{f \in \Sigma}$ telle que si $f : s \leftarrow s_1 \ldots s_n$ alors $f^A : A_s \leftarrow A_{s_1} \ldots A_{s_n}$.

Définition : Un morphisme d'algèbres h est une famille d'applications $(h_s)_{s\in S}$ entre les supports $A = (A_s)_{s\in S}$ et $B = (B_s)_{s\in S}$ de deux $\langle S,\Sigma\rangle$-algèbres (A,Σ_A) et (B,Σ_B), vérifiant la propriété suivante :

$\forall f : s \leftarrow s_1,\dots s_n$ de Σ, $\forall x_1 \in A_{s_1},\dots x_n \in A_{sn}$

$h_s(f^A(x_1\dots x_n)) = f^B(h_{s_1}(x_1),\dots h_{sn}(x_n))$

Si h est une famille de bijections alors h est un isomorphisme de (A,Σ_A) dans (B,Σ_B).

La relation "il existe un unique morphisme h de (A,Σ_A) vers (B,Σ_B)" définit un préordre sur la classe des $\langle S,\Sigma\rangle$-algèbres. L'algèbre minimale pour cette relation, unique à un isomorphisme près, est appelée l'algèbre initiale.

Définition : L'algèbre des termes sur la signature $\langle S,\Sigma\rangle$ est notée $T_{\langle S,\Sigma\rangle}$ ou T_Σ. Son support noté $\text{support}(T_\Sigma)$ (abrégé en T_Σ) est défini par :

- $\forall s \in S \quad \Sigma_{\Lambda,s} \subset \text{support}(T_\Sigma)$
- $\forall f \in \Sigma_{s_1..s_n,s} \quad \forall t_1,\dots t_n \in \text{support}(T_\Sigma)$ tels que $\text{sorte}(t_i) = s_i$ pour $i = 1\dots n$ (où sorte est prolongée aux termes) $f(t_1,\dots t_n) \in \text{support}(T_\Sigma)$.

Les opérations de l'algèbre des termes sont telles que le terme associé aux termes $t_1,\dots,t_n \in \text{support}(T_\Sigma)$, de sortes $s_1\dots s_n$ respectivement, par l'opération f de $\Sigma s_1\dots s_n,s$ est $f(t_1\dots t_n)$.

Propriété : L'algèbre des termes est initiale dans la classe des $\langle S,\Sigma\rangle$-algèbres.

Définition : Une algèbre (A,Σ_A) est une algèbre finiment engendrée si les éléments de son support sont obtenus par composition des opérations. C'est-à-dire si le morphisme h de l'algèbre des termes dans (A,Σ_A) est surjectif.

Remarque : Si Σ ne contient pas de constantes ($\Sigma_{\Lambda,s} = \emptyset$) alors les supports des algèbres finiment engendrées sont vides.

A chaque sorte $s \in S$ on associe un ensemble X_s de variables de sorte s. On note X la famille des ensembles disjoints $(X_s)_{s\in S}$. On définit profil(x) pour tout $x \in X$, comme étant l'unique (s,s) tel que $x \in X_s$.

Définition : L'algèbre libre des termes sur la signature $\langle S,\Sigma\rangle$ est notée $T_{\langle S,\Sigma\rangle}(X)$ ou $T_\Sigma(X)$. Son support, noté $\text{support}(T_\Sigma(X))$ (abrégé en $T_\Sigma(X)$) est défini par :

- $\forall s \in S \quad \Sigma_{\Lambda,s} \cup X_s \subset \text{support}(T_\Sigma(X))$
- $\forall f \in \Sigma s_1\dots s_n,s \quad \forall t_1\dots t_n \in \text{support}(T_\Sigma(X))$ tels que $\text{sorte}(t_i) = s_i$ pour $i = 1,\dots,n$ $f(t_1,\dots,t_n) \in \text{support}(T_\Sigma(X))$

Les opérations de l'algèbre libre des termes sont telles que le terme avec variables associé aux termes avec variables $t_1\dots t_n \in \text{support}(T_\Sigma(X))$ de sortes $s_1\dots s_n$ par l'opération $f \in \Sigma s_1\dots s_n,s$ est $f(t_1\dots t_n)$.

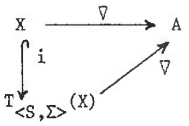
Le profil d'un terme est le couple formé du domaine du terme et de sa sorte :

$\forall t \in T_\Sigma(X)$ $\quad \text{profil}(t) = (\text{domaine}(t), \text{sorte}(t))$

$\text{domaine}(f(t_1\dots t_n)) = \text{domaine}(t_1)\dots\text{domaine}(t_n)$

$\text{sorte}(f(t_1\dots t_n)) = \text{sorte}(f)$

Pour toute $\langle S,\Sigma\rangle$-algèbre (A,Σ_A), une application $\nabla : X \to A$ appelée affectation peut être prolongée de façon unique en un morphisme encore noté ∇ de $T_{\langle S,\Sigma\rangle}(X)$ vers A tel que le diagramme ci-dessous commute : $i : X \to T_{\langle S,\Sigma\rangle}(X)$ est l'injection canonique.



Définition : Soit $<S,\Sigma>$ et $<S',\Sigma'>$ deux signatures munies des ensembles de variables typées $X = (X_s)_{s\in S}$ et $X' = (X'_{s'})_{s'\in S'}$. Un morphisme de signature h est un couple d'applications (h_S,h_Σ) où $h_S : S \to S'$ et $h_\Sigma : \Sigma \longrightarrow \Sigma'$ tel que

- pour tout $f \in \Sigma$ tel que $profil(f) = s_1 \ldots s_n, s$: h est étendu aux termes de $T_\Sigma(X)$: $profil(h_\Sigma(f)) = h_S(s_1)\ldots h_S(s_n), h_S(s)$.
- pour tout $x \in X$ $h(x)$ est un élément de X' vérifiant : $sorte(h(x)) = h_S(sorte(x))$
- pour tout $f(t_1\ldots t_n) \in T_\Sigma(X)$
  $h(f(t_1\ldots t_n)) = h_\Sigma(f)(h(t_1)\ldots h(t_n))$
  donc pour tout terme $t \in T_\Sigma(X)$
  $profil(h(t)) = (h_S(domaine(t)), h_S(sorte(t)))$.

Intuitivement un tel morphisme est une application qui associe aux termes de $T_\Sigma(X)$, des termes de $T_{\Sigma'}(X')$ de profil correspondant.

Définition : Une Σ-équation e est un couple de termes g,d de $T_\Sigma(X)$ de même sorte. Elle est notée $e = [g = d]$ ou $g = d$.

Définitions : Une Σ-algèbre (A,Σ_A) valide une Σ-équation $e = [g=d]$ si et seulement si pour toute affectation $\nabla : X \to A$, $\nabla(g) = \nabla(d)$. A valide un ensemble E d'équations si et seulement si A valide toutes les équations de E. On dit alors que A est un modèle de E.

Définitions : Soient E un ensemble de Σ-équations et $g = d$ une Σ-équation. On dit que E vérifie $g = d$ et on note $E \vdash\!\!\!- g = d$ si et seulement si $g = d$ est valide dans tout modèle de E.

E vérifie un ensemble E' d'équations et on note $E \vdash\!\!\!- E'$ si et seulement si E vérifie toutes les équations de E'.

L'ensemble des équations vérifiées par E est appelé la théorie inductive définie par E.

Définition : Soit E un ensemble de Σ-équations. Une relation $\sim_E$ est une congruence compatible avec E si et seulement si

- $\sim_E$ est une relation d'équivalence sur les termes de T_Σ validant les équations de E et
- $\sim_E$ est compatible avec les opérations de Σ. C'est-à-dire
  $\forall f \in \Sigma s_1\ldots s_n, s, \forall t_1\ldots t_n, t'_1\ldots t'_n \in T_\Sigma$ tels que
  $sorte(t_i) = sorte(t'_i) = s_i$ pour $i = 1\ldots n$,
  $t_1 \sim_E t'_1, \ldots, t_n \sim_E t'_n \Rightarrow f(t_1\ldots t_n) \sim_E f(t'_1,\ldots t'_n)$.

Définition : Soit E un ensemble de Σ-équations.

On note $T_{\Sigma,E}$ l'algèbre quotient de l'algèbre des termes par la plus petite congruence compatible avec E, notée $\equiv_E$: $T_{\Sigma,E} = T_\Sigma/_{\equiv_E}$. Cette algèbre valide E. Elle est initiale dans la classe des Σ-algèbres validant les équations de E.

Définition : Une spécification est la donnée d'un triplet $<S,\Sigma,E>$ où

$<S,\Sigma>$ est une signature, et

E est un ensemble de Σ-équations.

Définition : Soient $<S,\Sigma,E>$ et $<S',\Sigma',E'>$ deux spécifications. Un morphisme de spécifications h est un morphisme de signatures de $<S,\Sigma>$ vers $<S',\Sigma'>$ munies des ensembles de variables typées X et X' tel que E' vérifie les images des équations de E :

$$\forall g = d \in E \qquad E' \vdash\!\!\!- h(g) = h(d).$$

Propriété : La composition de deux morphismes de spécifications est un morphisme de spécifications.

Démonstration : Soient SPEC1 = <S1,Σ1,E1>, SPEC2 = <S2,Σ2,E2> et SPEC3 = <S3,Σ3,E3> trois spécifications et h1 : SPC1 → SPEC2 et h2 : SPEC2 → SPEC3 deux morphismes de spécifications.

Le morphisme de signature, composé de h1 par h2, noté $h2\circ h1 : SPEC1 \to SPEC3$ est défini par le couple $(h2\circ h1_{S1}, h2\circ h1_{\Sigma 1})$ où $h2\circ h1_{S1}(s) = h2_{S2}(h1_{S1}(s))$ pour tout $s \in S1$

$h2\circ h1_{\Sigma 1}(f) = h2_{\Sigma 2}(h1_{\Sigma 1}(f))$ pour tout $f \in \Sigma 1$.

Pour tout $f \in \Sigma 1$ tel que $profil(f) = s_1 \ldots s_n, s$, on a

$$profil(h2\circ h1_{\Sigma 1}(f)) = h2\circ h1_{S1}(s_1)\ldots h2\circ h1_{S1}(s_n), h2\circ h1_{S1}(s) :$$

$$profil(h2\circ h1_{\Sigma 1}(f)) = profil(h2_{\Sigma 2}(h1_{\Sigma 1}(f))))$$

$$= h2_{S2}(profil(h1_{\Sigma 1}(f)))$$

$$= h2_{S2}(h1_{S1}(s_1)\ldots h1_{S1}(s_n), h1_{S1}(s))$$

$$= h2_{S2\circ h1_{S1}}(s_1)\ldots h2_{S2\circ h1_{S1}}(s_n), h2_{S2\circ h1_{S1}}(s)$$

$h2\circ h1$ est donc un morphisme de signature. De plus

$\forall g = d \in E1 \quad E3 \vdash h2\circ h1(g) = h2\circ h1(d)$:

$\forall g = d \in E1 \quad E2 \vdash h1(g) = h1(d)$

et $\forall g' = d' \in E2 \quad E3 \vdash h2(g') = h2(d')$

donc si $E2 \vdash g'' = d''$ alors $E3 \vdash h2(g'') = h2(d'')$

donc $E3 \vdash h2\circ h1(g) = h2\circ h1(d)$

Définition : Un type abstrait algébrique défini par la spécification <S,Σ,E>, est la classe des Σ-algèbres finiment engendrées, validant les équations de E, et isomorphes à l'algèbre initiale $T_{\Sigma,E}$.

Exemple de spécification :

```
TYPE BOOL
     opérations
               vrai : BOOL ←
               faux : BOOL ←

               non : BOOL ← BOOL
               et  : BOOL ← BOOL,BOOL
               ou  : BOOL ← BOOL,BOOL
               eq  : BOOL ← BOOL,BOOL
               impl : BOOL ← BOOL,BOOL

     variable
               b : BOOL

     équations
               non(vrai) = faux
               non(faux) = vrai
              et(vrai,b) = b
              et(faux,b) = faux

              ou(vrai,b) = vrai
              ou(faux,b) = b

              eq(vrai,b) = b
              eq(faux,b) = non(b)

            impl(vrai,b) = b
            impl(faux,b) = vrai
```

Notation : Les opérations et, ou et impl seront notées sous la forme infixée b et b', b ou b' et b impl b'.

Autre exemple de spécification :

TYPE ENTMOD2

opérations

constructeurs zéro : ENTMOD2 ←
suc : ENTMOD2 ← ENTMOD2

modificateurs plus : ENTMOD2 ← ENTMOD2,ENTMOD2
mult : ENTMOD2 ← ENTMOD2,ENTMOD2

variables

em,em' : ENTMOD2

équations

relation entre les constructeurs
suc(suc(em)) = em
définition des modificateurs
plus(zéro,em) = em
plus(suc(em),em') = suc(plus(em,em'))

mult(zéro,em) = zéro
mult(suc(em),em') = plus(mult(em,em'),em')

Définition : Deux spécifications $<S,\Sigma,E>$ et $<S,\Sigma,E'>$ sur une même signature sont équivalentes si et seulement si les congruences $\equiv_E$ et $\equiv_{E'}$ coïncident. C'est-à-dire si $T_{\Sigma,E}$ et $T_{\Sigma,E'}$ sont isomorphes.

Pour que deux spécifications $<S,\Sigma,E>$ et $<S,\Sigma,E'>$ sur une même signature soient équivalentes, il faut et il suffit que

$$E \vdash\!\!\!- E' \quad \text{et} \quad E' \vdash\!\!\!- E$$

Définition : Soit $<S,\Sigma,E>$ une spécification. On définit la combinaison de $<S,\Sigma,E>$ et $<S',\Sigma',E'>$ comme étant la spécification $<S \cup S', \Sigma \cup \Sigma', E \cup E'>$, notée $<S,\Sigma,E>+<S',\Sigma',E'>$.

1.2.2.- Propriétés des spécifications

La construction d'une spécification se fait par étapes successives : partant d'une spécification primitive, on peut rajouter soit des opérations, on parle alors "d'enrichissements", soit également une nouvelle sorte, on parle alors "d'extensions". A chaque étape on doit s'assurer que le type abstrait défini à l'étape précédente n'a pas été modifié soit par adjonction de nouveaux objets non équivalents à ceux définis, soit par modification des relations d'équivalences.

Définition : Une spécification $<S,\Sigma,E>$ est une extension d'une spécification $<S_0,\Sigma_0,E_0>$ si et seulement si

$$S_0 \subset S,\ \Sigma_0 \subset \Sigma \quad \text{et} \quad E \vdash\!\!\!- E_0$$

Définition : Un enrichissement est une extension laissant l'ensemble des sortes inchangé.

Définition : Une spécification SPEC est une restriction d'une spécification SPEC' si et seulement si SPEC' est un enrichissement de SPEC.

Définition : Une spécification structurée (ou hiérarchique [WIR, BROY,81]) est définie comme extension d'une spécification primitive SPEC = $<S,\Sigma,E>$ appelée son environnement, par le triplet

$<\{T\},\Sigma_T,E_T>$ appelé présentation du type T.

T est le type d'intérêt de la spécification $<S \cup \{T\}, \Sigma \cup \Sigma_T, E \cup E_T>$ définie.

Une spécification structurée est consistante vis à vis de son environnement si la relation d'équivalence sur les termes des types de l'environnement n'a pas été modifiée.

Définition : Une spécification $SPEC = \langle S,\Sigma,E\rangle$, extension de la spécification $SPEC_0 = \langle S_0,\Sigma_0,E_0\rangle$ est consistante vis à vis de $SPEC_0$ si et seulement si la restriction de $\equiv_E$ aux Σ_0-termes coïncide avec $\equiv_{E_0}$ c'est-à-dire $\forall t_0,t_0' \in T_{\Sigma_0}$

$$t_0 \equiv_E t_0' \Rightarrow t_0 \equiv_{E_0} t_0'$$

la spécification SPEC est consistante sur les variables vis à vis de $SPEC_0$ si et seulement si

$$\forall t_0,t_0' \in T_{\Sigma_0}(X) \qquad t_0 \equiv_E t_0' \Rightarrow t_0 \equiv_{E_0} t_0' .$$

Une spécification structurée est suffisamment complète vis à vis de son environnement si tout terme de sorte appartenant à l'environnement est équivalent à un terme de l'environnement.

Définition : Une spécification $SPEC = \langle S,\Sigma,E\rangle$, extension de la spécification $SPEC_0 = \langle S_0,\Sigma_0,E_0\rangle$ est suffisamment complète vis à vis de $SPEC_0$ si et seulement si

$$\forall t \in T_\Sigma \quad tq \quad sorte(t) \in S_0$$

$$\exists t_0 \in T_{\Sigma_0} \qquad t \equiv_E t_0$$

SPEC est suffisamment complète sur les variables vis à vis de $SPEC_0$ si et seulement si

$$\forall t \in T_\Sigma(X_0)$$

$$tq \quad sorte(t) \in S_0$$

les variables de t sont d'un type de S_0

$$\exists t_0 \in T_{\Sigma_0}(X_0) \quad t \equiv_E t_0.$$

1.3.- Spécification d'un type abstrait algébrique

1.3.1.- Le problème de la consistance et de la complètude suffisante

Comment savoir si la spécification d'un type abstrait est consistante et suffisamment complète ? Ce problème est en général indécidable [GUT, HOR, 78]. Cependant il existe des méthodes syntaxiques de spécification assurant la consistance et la complètude suffisante de la spécification obtenue.

Une telle méthode a été étudiée en détail par M. BIDOIT [BID, 81]. Nous nous limitons à exposer ici les idées essentielles.

Pour spécifier algébriquement un ensemble d'objets, il faut choisir une famille d'opérations qui génèrent l'ensemble des objets, c'est-à-dire qui décrivent un (ou plusieurs) chemin passant par tous les objets. Ces opérations, appelées les constructeurs des objets du type, sont à valeur dans le type. Chaque classe d'équivalence de termes du type contient un terme composé uniquement de constructeurs. Un tel terme est appelé forme normale des autres objets de la classe. Si le chemin décrit passe une fois et une seule par les objets du type alors il n'y a pas de relation entre les constructeurs (exprimée par des équations du type dont les termes sont en forme normale).

Les opérations d'un type qui ne sont pas des constructeurs sont appelées modificateurs si elles sont à valeur dans le type, et observateurs si elles sont à valeur dans un type de l'environnement.

Pour définir de manière complète les observateurs et les modificateurs d'un type, il faut décrire leur action dans tous les cas, c'est-à-dire ce qui se passe quand on les applique sur l'un ou sur l'autre des constructeurs. Les équations définissant une opération f auront comme terme gauche, un terme de la forme

$$f(c(t_1 \ldots t_i),t_{i+1},..,t_n) \text{ où } c \text{ est un constructeur.}$$

Le terme droit de l'équation ne devra pas introduire de nouvelles variables ou d'opérations du type autres que des constructeurs ou f et ne devra pas être "plus compliqué" que le terme gauche. (De manière

que le système de réécriture de termes associé aux équations orientant celles-ci de gauche à droite, soit à terminaison finie).

M. BIDOIT appelle présentation gracieuse, une présentation $<\{T\}, \Sigma_T, E_T>$ du type abstrait T sur l'environnement SPEC = $<S,\Sigma,E>$, telle que parmi les opérations Σ_T, il y ait une famille de constructeurs et que les opérations E_T puissent être partagées en

- un ensemble d'équations dont les termes sont en forme normale, définissant des relations entre les constructeurs,
- un ensemble d'équations définissant de manière complète les modificateurs et les observateurs.

Exemple de présentation gracieuse :

Le type ENTNAT des entiers naturels présenté sur l'environnement

$<\{BOOL\}, \Sigma_{BOOL}, E_{BOOL}>$ spécifié ci-dessus (§ I.1.2.)

TYPE ENTNAT

type de l'environnement : BOOL

opérations :			notations usuelles
constructeurs :	zéro	: ENTNAT ←	0
	suc	: ENTNAT ← ENTNAT	+1
modificateurs :	plus	: ENTNAT ← ENTNAT,ENTNAT	+
observateurs :	infeg	: BOOL ← ENTNAT,ENTNAT	≤
	eq	: BOOL ← ENTNAT,ENTNAT	=

variables : e,e' : ENTNAT

équations :

pas de relations entre les constructeurs

définition du modificateur :

plus(zéro,e) = e

plus(suc(e),e') = suc(plus(e,e'))

définition des observateurs :

infeg(zéro,zéro) = vrai

infeg(zéro,suc(e)) = vrai
infeg(suc(e),zéro) = faux
infeg(suc(e),suc(e')) = infeg(e,e')
eq(zéro,suc(e)) = faux
eq(suc(e),zéro) = faux
eq(suc(e),suc(e')) = eq(e,e')

Théorème : Soit $<\{T\},\Sigma_T,E_T>$ une présentation gracieuse du type T sur l'environnement SPEC = $<S,\Sigma,E>$. Alors le type abstrait est suffisamment complet vis à vis de SPEC. Si de plus E_T ne contient pas d'équations exprimant une relation entre les constructeurs, alors le type abstrait est consistant vis à vis de SPEC.

Idée de la démonstration :

Le système de réécriture de termes obtenu en orientant de gauche à droite l'ensemble d'équations E_T sans les relations entre les constructeurs, est sans paire critique et à terminaison finie. C'est donc un système de réécriture de termes canonique. Tout terme possède une unique forme normale ne contenant que des constructeurs de T et des opérations de l'environnement ($\in \Sigma$).

- Le type abstrait est consistant si E_T ne contient pas de relations entre les constructeurs puisque le système de réécriture associé est canonique.
- Il est suffisamment complet puisque tout terme a une forme normale (ici le fait qu'elle ne soit pas unique n'a pas d'importance, donc E_T peut contenir des relations entre les constructeurs). Un terme en forme normale ne contenant que des constructeurs et des opérations de l'environnement, un terme d'un type appartenant à l'environnement aura une forme normale ne contenant que des opérations de l'environnement.

Nous renvoyons le lecteur à [BID, 81] pour la démonstration complète de ce théorème.

Remarque : Soit T un type présenté gracieusement. Nous l'enrichissons avec une opération définie comme composée d'opérations du type. Alors T ne perd pas ses propriétés de consistance et de complétude suffisante.

1.3.2.- Le problème des opérations partielles

Nous avons vu que pour spécifier un type abstrait consistant et suffisamment complet vis à vis de son environnement, il faut définir de manière complète tous les modificateurs et observateurs du type. Comment faire si ces opérations sont partielles c'est-à-dire non définies pour certaines valeurs. Prenons par exemple l'opération de modification moins du type ENT spécifié ci-dessus (§ I.1.3.1). Son profil est

moins : ENT ← ENT,ENT.

Elle peut être définie par les équations

moins(zéro,zéro) = zéro

moins(suc(e),zéro) = suc(e)

moins(suc(e),suc(e')) = moins(e,e')

où e et e' sont des variables de type ENT.

Mais que dire de moins(zéro,suc(e)) ?

Plusieurs méthodes ont été proposées. Elles peuvent être regroupées en 2 catégories :

- celles nécessitant l'introduction d'un nouvel arsenal théorique : on peut citer les méthodes fondées sur les algèbres partielles [BRO, WIR, 82] et la méthode fondée sur les algèbres à opérateurs multicibles ([BOI, GUI, PAU, 83]).
- celles n'en nécessitant pas : elles introduisent explicitement une ou plusieurs valeurs erronées ou indéfinies, un prédicat de définition et des équations de propagation des erreurs ([ADJ, 78],[GOG, 78], [REM, 82]).

Notre objectif n'est pas de présenter une nouvelle méthode de traitement des opérations partielles. Nous avons voulu résoudre ce problème de la manière qui nous a parue la plus simple à utiliser et qui ne demande pas l'introduction de nouvelles notions théoriques.

Nous utilisons des constantes "indéfinies", une par sorte. Contrairement aux constantes d'erreurs, un terme comportant en paramètre une constante indéfinie, n'est pas toujours indéfini lui-même. En particulier le type des Booléens ne comporte pas de constante indéfinie. Un prédicat est toujours soit vrai soit faux.

Il faut préciser pour chaque opération, son comportement face à un indéfini comme face à un constructeur supplémentaire. Ceci augmente le nombre d'équations d'une spécification. Pour éviter les inconsistances liées à l'indéfini, nous introduisons des constructeurs cachés, doublant les constructeurs usuels. Ces opérations sont les constructeurs effectifs en ce sens qu'ils apparaissent dans la forme normale des termes. Cependant ils sont cachés en ce sens qu'ils ne peuvent pas être utilisés par un autre type ([PAR, PRI, 73]).

Ils servent à définir les opérations du type. Les constructeurs usuels deviennent des modificateurs selon les critères exposés ci-dessus mais nous les appellerons les constructeurs utilisables.

Introduction d'une constante indéfinie dans une présentation du type T :

Soit $<\{T\},\Sigma_T,E_T>$ une présentation du type T sur l'environnement SPEC = $<S,\Sigma,E>$ $\Sigma_T = C_T \cup M_T \cup O_T$ où C_T est l'ensemble des constructeurs du type T, M_T celui des modificateurs et O_T celui des observateurs.

Nous enrichissons cette présentation d'une constante indéfinie :

$<\{T\},\Sigma'_T,E'_T>$ est la présentation enrichie

$\Sigma'_T = C'_T \cup M'_T \cup O'_T$ est le nouvel ensemble d'opérations où

$C'_T = \{cfn \text{ tq } profil(cfn) = profil(c) \text{ pour tout } c \in C_T\} \cup \{indef\}$

$M'_T = M_T \cup C_T$

$O'_T = O_T$

$\{cfn \text{ tq } profil(cfn) = profil(c) \text{ pour tout } c \in C_T\}$ est l'ensemble des constructeurs cachés doublant les anciens constructeurs de T.

indef est la constante indéfinie de profil $(\wedge,T)$.

D'où

$\Sigma'_T = \Sigma_T \cup \{cfn : profil(c), \forall c \in C_T\} \cup \{indef\}$

E'_T est le nouvel ensemble d'équations. Il est formé des équations de E_T où chaque constructeur $c \in C_T$ est remplacé par le constructeur caché $cfn \in C'_T$, des définitions des anciens constructeurs de C_T en fonction des constructeurs de C'_T (noté $Def\text{-}C_T|_{C'_T}$) et des équations définissant l'action de toutes les opérations de $M_T \cup O_T$ sur la constante indéfinie (noté $Def\text{-}M_T \cup O_T|_{\{indef\}}$) :

$$E'_T = E_T[c \leftarrow cfn \quad \forall c \in C_T] \cup Def\text{-}C|_{C'_T} \cup Def\text{-}M_t \cup O_T|_{\{indef\}}$$

Il est alors possible de définir des opérations partielles. Nous présentons l'exemple du type des entiers ENTNAT étudié au paragraphe I.1.3.1 enrichi d'une constante indéfinie , des constructeurs cachés et de l'opération partielle moins.

Exemple de spécification avec des opérations partielles, ici le modificateur moins

TYPE ENTIER
environnement : BOOL

			notations usuelles
opérations	constructeurs :	zéro : ENTIER ←	0
		suc : ENTIER ← ENTIER	+1
		indef : ENTIER ←	ω
	modificateurs :	plus : ENTIER ← ENTIER,ENTIER	+
		moins : ENTIER ← ENTIER,ENTIER	-
		si-alors-sinon : ENTIER ← ENTIER,ENTIER	
	observateurs :	infeg : BOOL ← ENTIER,ENTIER	≤
		eq : BOOL ← ENTIER,ENTIER	=
	cachées :	zérofn : ENTIER ←	
		sucfn : ENTIER ← ENTIER	

variables e,e' : ENTIER
équations :
définition des constructeurs utilisables :

```
zéro            = zérofn
suc(zérofn)     = sucfn(zérofn)
suc(sucfn(e))   = sucfn(sucfn(e))
suc(indef)      = indef
```


définition des modificateurs :

```
plus(zérofn,e)     = e
plus(sucfn(e),e')  = suc(plus(e,e'))
plus(indef,e)      = indef

moins(zérofn,zérofn) = zérofn
moins(zérofn,sucfn(e)) = indef
moins(sucfn(e),zérofn) = sucfn(e)
moins(sucfn(e),sucfn(e')) = moins(e,e')
moins(e,indef)         = indef
moins(indef,e)         = indef

si-alors-sinon(vrai,e,e') = e
si-alors-sinon(faux,e,e') = e'
```

définition des observateurs :

```
infeg(zérofn,zérofn)       = vrai
infeg(sucfn(e),zérofn)     = faux
infeg(zérofn,sucfn(e))     = vrai
infeg(sucfn(e),sucfn(e'))  = infeg(e,e')
infeg(indef,zérofn)        = faux
infeg(indef,sucfn(e))      = faux
infeg(indef,indef)         = vrai
infeg(zérofn,indef)        = faux
infeg(sucfn(e),indef)      = faux

eq(zérofn,zérofn)          = vrai
eq(zérofn,sucfn(e))        = faux
eq(sucfn(e),zérofn)        = faux
eq(sucfn(e),sucfn(e'))     = eq(e,e')
eq(indef,zérofn)           = faux
eq(indef,sucfn(e))         = faux
eq(zérofn,indef)           = faux
eq(sucfn(e),indef)         = faux
eq(indef,indef)            = vrai
```

Remarques

- Le Type ENTIER est présenté gracieusement.

- infeg et eq sont des prédicats. Ils ne propagent pas l'"erreur" : les équations sont telles, qu'un terme de racine infeg ou eq ne peut être équivalent à indef.

- la définition de eq peut être simplifiée par l'utilisation de l'équation de symétrie :

$$eq(e,e') = eq(e',e)$$

Mais cette équation ne correspond pas aux critères imposés par la présentation gracieuse. (Une telle équation enlèverait la propriété de terminaison finie du système de réécriture associé).

Rajouter à une spécification une constante indéfinie et des constructeurs, modifie l'algèbre des termes et l'algèbre quotient. Le terme indef définit une classe contenant les termes équivalents à indef. Toutes les classes sont enrichies de termes contenant les constructeurs cachés. Les formes normales en particulier contiennent les constructeurs cachés. L'algèbre quotient est enrichie également de classes à un seul élément qui sont les termes formés par application des constructeurs cachés sur la constante indéfinie :

La classe d'un terme t est notée [t]

$$T_{\Sigma',E'} = T_{\Sigma,E} \cup \{[indef]\} \cup \{[cfn(indef...)], \forall cfn \in C'_T\}$$

Les classes [cfn(indef,...)] ne contiennent qu'un élément : cfn(indef...).

Reprenons l'exemple ci-dessus :

La nouvelle forme normale des entiers est :

$$sucfn(sucfn(... (zérofn)...))$$

Les termes $sucfn^n(indef)$ forment les nouvelles classes d'équivalence à un seul élément dans l'algèbre quotient.

Notations : acceptant la surcharge des opérateurs, nous notons pour tous les types

indef la constante indéfinie (le type BOOL n'en comporte pas).

eq le prédicat d'égalité.

cfn le constructeur caché doublant le constructeur c

si-alors-sinon l'opération définie sur tout type T par

si-alors-sinon : T ← BOOL,T,T

si vrai alors t sinon t' = t

si faux alors t sinon t' = t'

où t et t' sont des variables de type T.

L'opération si-alors-sinon est utilisée en notation "dist-fixée" : si b alors t sinon t'.

Dans la suite nous n'introduirons pas de constantes doublant les constructeurs constantes.

1.3.3.- Invariant

Un invariant est une propriété que doivent vérifier tous les objets du type. Un invariant détermine une restriction sur l'ensemble des objets générés par les constructeurs du type.

Exemple d'invariant sur le type des entiers naturels ENTIER :

$$infeg(e,suc^{10}(zéro)).$$

L'introduction d'un invariant va modifier une spécification. La méthode utilisée pour traiter les indéfinis va nous servir aussi pour traiter les invariants. En effet, l'introduction d'un invariant va simplement se traduire par l'adjonction d'une précondition aux équations définissant les constructeurs utilisables en fonction des constructeurs effectifs. De cette manière seuls les objets vérifiant l'invariant seront effectivement construits. Mais la spécification n'a pas changé et les opérations sont définies sur l'ensemble des termes générés par les constructeurs effectifs.

Exemple de spécification avec un invariant :

Le type des entiers naturels inférieurs à 10 (suc^{10}(zéro)).

TYPE ENTINF 10
environnement : BOOL
opérations : les mêmes que celles du type ENTIER
variables : ei,ei' : ENTINF 10
équations :

définition du constructeur utilisable

suc(zéro) = sucfn(zéro)
suc(sucfn(ei)) = si infeg(sucfn(sucfn(ei)),suc^{10}(zéro))
alors sucfn(sucfn(ei))
sinon sucfn(ei)
suc(indef) = indef

définition des modificateurs et des observateurs :
mêmes équations que celles du type ENTIER où on a remplacé les variables e et e' par ei et ei'.

Remarques :

- Dans la suite nous laisserons l'invariant d'un type figurer comme terme booléen, sans effectuer la modification indiquée ci-dessus.

- suc(sucfn(ei)) = sucfn(ei)
si ei est supérieur à 10.
Ici nous préférons "garder l'information" (ici sucfn(ei)) et ne pas effectuer l'opération qui nous "mènerait en erreur". Nous aurons toujours cette politique, ce qui explique pourquoi nous n'avons pas de constantes d'erreurs mais seulement des constantes indéfinies : nous nous arrêtons avant d'aller en erreur et gardons le terme déjà construit. (Les inconsistances sont évitées car les opérations sont définies par leur action sur les constructeurs effectifs).

2.- SPECIFICATIONS PARAMETREES

Pour spécifier une structure de suite d'éléments, il n'est pas nécessaire de connaître avec précision de quels éléments il s'agit. En effet les caractéristiques d'une suite sont les mêmes, qu'il s'agisse d'une suite d'entiers ou d'une suite de caractères. Nous avons envie de pouvoir décrire cette structure indépendamment des éléments qui la composent et de substituer lors d'une utilisation au type des "éléments quelconques" le type des éléments effectifs.

Une spécification paramétrée est une spécification structurée comportant dans son environnement un type appelé paramètre formel dont la spécification ne comprend que les opérations utilisées par le type d'intérêt et pas forcément de constructeurs (elle peut avoir une algèbre initiale vide).

Pour utiliser une spécification paramétrée il faut instancier le type paramètre formel par un type défini : Le type SUITE instancié par le type des caractères donnera le type des suites de caractères.

La propriété demandée à une spécification paramétrée est la protection de paramètre effectif, c'est-à-dire la consistance et la suffisante complétude de la spécification obtenue par instanciation d'une spécification paramétrée, vis à vis de son paramètre effectif. Cette propriété appelée la persistance est stable par composition.

Les types paramétrés sont définis par [ADJ, 82] comme des foncteurs associant à un type d'objets de base, l'extension de ce type par le type des objets structurés, composés sur ces objets de base. Tout en ayant la même approche, nous n'utilisons pas la théorie des catégories.

En particulier notre démonstration de la consistance et de la suffisante complétude d'une spécification obtenue par instanciation d'une spécification paramétrée persistante, peut paraître moins élégante.

2.1.- Définitions

Soient SPEC = $<S,\Sigma,E>$ une spécification appelée environnement, et

SPECP = $<S_p,\Sigma_p,E_p>$ une présentation appelée paramètre formel, sur SPEC.

Définition : On appelle spécification paramétrée par S_p, la spécification définie par

$$SPECTP = SPEC+SPECP+<\{TP[S_p]\},\Sigma_{TP},E_{TP}>.$$

telle que il existe un constructeur c des objets du type TP qui comporte parmi les types de son domaine les sortes S_p : $S_p \subset \bigcup_{c \in C_{TP}}$ domaine(c).

Remarque : Une spécification paramétrée est une spécification structurée comportant dans son environnement une ou plusieurs sortes appelées paramètres formels. Les spécifications associées à ces sortes ne sont généralement pas complètes et peuvent avoir une algèbre initiale vide. Cependant, considérant le type d'intérêt de la spécification paramétrée, nous pouvons utiliser les mêmes critères (de présentations gracieuses) que pour des spécifications non paramétrées pour s'assurer de la consistance et complétude vis à vis de l'environnement.

Exemple de spécification paramétrée : Le type SUITE[P]

Soit la spécification environnement SPEC = <S,Σ,E> où S = {BOOL,ENTIER}.

TYPE paramètre formel P présenté par $<\{P\},\Sigma_p,E_p>$ sur SPEC

environnement :	BOOL,ENTIER
opérations :	Σ_p = {indef, eq}
constructeur	indef : P ←
observateur	eq : BOOL ← P,P
variables :	p,p',p" : P
équations :	E_p comprend les équations suivantes :

Propriétés de l'observateur :

eq(p,p) = vrai
eq(p,p') = eq(p',p)
(eq(p,p') et eq(p',p")) impl eq(p,p") = vrai

TYPE SUITE[P] présenté par $<\{SUITE[P]\}, \Sigma_{SUITE}, E_{SUITE}>$
sur SPEC+<{P},Σ_p,E_p> où P est le type paramètre formel

environnement : BOOL,ENTIER,P

opérations : Σ_{SUITE} comprend les opérations suivantes :

constructeurs :	Lexique
svide : SUITE[P] ←	suite vide
ajout : SUITE[P] ← SUITE[P],P	adjonction en tête
modificateurs :	
sup : SUITE[P] ← SUITE[P],ENTIER	suppression du $n^{ième}$ élément
conc : SUITE[P] ← SUITE[P],SUITE[P]	concaténation de 2 suites
si-alors-sinon : SUITE[P] ← BOOL,SUITE[P],SUITE[P]	
observateurs :	
acc : P ← SUITE[P],ENTIER	accès au $n^{ième}$ élément
rang : ENTIER ← SUITE[P],P	première occurrence d'un élément
taille : ENTIER ← SUITE[P]	nombre d'éléments d'une suite
eq : BOOL ← SUITE[P],SUITE[P]	prédicat d'égalité

variables :
sp,sp' : SUITE[P]
p, p' : P
e, e' : ENTIER

équations : E_{SUITE} comprend les équations suivantes

définition des modificateurs

sup(svide,e)	= svide
sup(ajout(s,p),zéro)	= s
sup(ajout(s,p),suc(e))	= ajout(sup(s,e),p)
conc(s,svide)	= s
conc(s,ajout(s',p'))	= ajout(conc(s,s'),p')
si vrai alors s sinon s'	= s
si faux alors s sinon s'	= s'

définition des observateurs

acc(svide,e)	= indef
acc(ajout(s,p),zéro)	= p
acc(ajout(s,p),suc(e))	= acc(s,e)
rang(svide,p)	= indef
rang(ajout(s,p),p')	= si eq(p,p') alors zéro sinon suc(rang(s,p')
taille(svide)	= zéro
taille(ajout(s,p))	= suc(taille(s))
eq(svide,svide)	= vrai
eq(ajout(s,p),svide)	= faux
eq(svide,ajout(s,p))	= faux
eq(ajout(s,p),ajout(s',p'))	= eq(p,p') et eq(s,s')

Soit SPECTP = SPEC+SPECP+<{TP[S_P]},Σ_{TP},E_{TP}> une spécification paramétrée par la présentation SPECP = <S_P,Σ_P,E_P> du paramètre formel sur l'environnement SPEC = <S,Σ,E>.

Définition : Une présentation SPECE = <S_E,Σ_E,E_E> sur le même environnement, est un paramètre effectif admissible de SPECTP s'il existe un morphisme de spécifications appelé morphisme de passage de paramètres

i : SPEC+SPECP → SPEC+SPECE

qui soit le morphisme identité sur SPEC.

Intuitivement, une spécification est un paramètre effectif admissible d'une spécification paramétrée, si elle comprend des opérations vérifiant les propriétés des opérations du paramètre formel.

Exemple de paramètre effectif admissible

Le type ENTIER spécifié au paragraphe I.1.3.2. est un paramètre effectif admissible de la spécification paramétrée SUITE[P].

En effet soit i est le morphisme de signatures défini par la donnée de (i_S,i_Σ)

où $i_S : S \cup \{P\} \to S \cup \{ENTIER\}$

i_S est l'identité de S

$i_S(P) = ENTIER$

$i_\Sigma : \Sigma \cup \Sigma_P \to \Sigma \cup \Sigma_{ENTIER}$

- $\forall f \in \Sigma \quad i(f) = f$
- $\Sigma_P = \{indef : \leftarrow P, eq : BOOL \leftarrow P,P\}$

$i_\Sigma(indef) = indef$ constante indéfinie de profil indef : ENTIER ←

$i_\Sigma(eq) = eq$ prédicat d'égalité de profil eq : BOOL ← ENTIER,ENTIER.

i est un morphisme de spécification car

$\forall [g = d] \in E_P \quad E_{ENTIER} \vdash i(g) = i(d)$.

$E_P = \{[eq(p,p) = vrai]$

$[eq(p,p') = eq(p',p)]$

$[(eq(p,p') \text{ et } eq(p',p")) \text{impl } eq(p,p") = vrai]\}$

Posons $i(p) = e$, $i(p') = e'$, $i(p") = e"$

$i([eq(p,p) = vrai]) = [eq(e,e) = vrai]$

$i([eq(p,p') = eq(p',p')]) = [eq(e,e') = eq(e',e)]$

$i([(eq(p,p') \text{ et } eq(p',p"))\text{impl} \quad eq(p,p") = vrai])$

$= [(eq(e,e') \text{ et } eq(e',e"))\text{impl } eq(p,p") = vrai]$

A-t-on $E_{ENTIER} \vdash [eq(e,e) = vrai]$

$[eq(e,e') = eq(e',e)]$

$[(eq(e,e') \text{et } eq(e',e"))\text{impl } eq(e,e") = vrai]$?

Pour le démontrer il suffit de faire un raisonnement par récurrence sur les constructeurs des entiers zéro et suc.

En effet, l'ensemble des équations vérifiées par E (la théorie inductive définie par E) est l'ensemble des équations démontrables par induction sur la structure des termes à l'aide des équations de E ([BUR, GOG, 77]).

Démonstration par induction :

A-t-on eq(e,e) = vrai ?
si e = zéro alors eq(zéro,zéro) = vrai
si eq(e,e) = vrai alors eq(suc(e),suc(e)) = eq(e,e) = vrai

A-t-on eq(e,e') = eq(e',e) ?
si e = e' alors on se ramène au cas ci-dessus
si e = zéro et e' = suc(e")
alors eq(zéro,suc(e")) = faux = eq(suc(e"),zéro)
si e = suc(e"), e' = suc(e"') et eq(e",e"') = eq(e"',e")
alors eq(suc(e"),suc(e"')) = eq(e",e"') = eq(e"',e") =
eq(suc(e"'),suc(e"))

A-t-on (eq(e,e') et eq(e',e")) impl eq(e,e") = vrai ?
si e = zéro
alors si e' = zéro
alors eq(e,e') = vrai
si e" = zéro
alors eq(e',e") = vrai et eq(e,e") = vrai
(vrai et vrai) impl vrai = vrai
si e" = suc($e"_o$)
alors eq(e',e") = faux et eq(e,e") = faux
(vrai et faux) impl faux = vrai
si e' = suc(e'_o)
alors eq(e,e') = faux et faux et eq(e',e") = faux
faux impl eq(e,e") = vrai
si e = suc(e_o)
alors si e' = zéro
alors eq(e,e') = faux et faux et eq(e',e") = faux
faux impl eq(e,e") = vrai

si e' = suc(e'_o)
alors eq(e,e') = eq(suc(e_o),suc(e'_o)) = eq(e_o,e'_o)
si e" = zéro
alors eq(e',e") = faux et eq(e_o,e'_o) et faux = faux
faux impl eq(e,e") = vrai
si e" = suc($e"_o$)
alors eq(e',e") = eq(suc(e'_o),suc($e"_o$)) = eq(e'_o,$e"_o$)
eq(e,e") = eq(suc(e_o),suc($e"_o$)) = eq(e_o,$e"_o$)
(eq(e_o,e'_o) et eq(e'_o,$e"_o$)) impl eq(e_o,$e"_o$)
= vrai par induction.

Donc $\forall [g = d] \in E_P \quad E_{ENTIER} \vdash i(g) = i(d)$.

ENTIER est un paramètre effectif admissible de SUITE[P].

Présentation obtenue par instanciation

La présentation obtenue par instanciation de SPECP par le paramètre effectif admissible SPECE, selon le morphisme de passage de paramètres i, est obtenue en substituant dans

- les profils des opérations de Σ_{TP}, les occurrences des sortes de S_P par leur sorte associée par i dans S_E,
- les équations de E_{TP}, les occurrences des opérations de Σ_P par leur opération associée par i de Σ_E et les occurrences des variables d'un type de S_P par une variable du type de S_E associé par i.

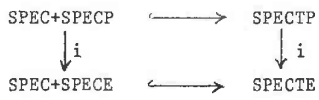
Spécification obtenue par instanciation :

SPECTE = SPEC+SPECE+<{TP[S_E]},Σ_{TE},E_{TE}>

où $\Sigma_{TE} = \Sigma_{TP}[P \leftarrow i(P), \forall P \in S_P]$

$E_{TE} = E_{TP}[X_P \leftarrow X_{i(P)}][f \leftarrow i(f), \forall f \in \Sigma_P]$

Diagramme :



Remarque : C'est un diagramme de "pushout".

Exemple de spécification obtenue par instanciation

type SUITE[P] instancié par le paramètre effectif admissible ENTIER :

Nous avons défini le morphisme de passage de paramètres i :

Nous substituons dans les profils des opérations de SUITE[P], les occurrences de P par $i_S(P)$=ENTIER et dans les équations de E_{SUITE}, l'opération eq du type formel P, par l'opération eq du type ENTIER, la constante indef du type P par la constante indef du type ENTIER et les variables p,p' de type P par les variables n,n' de type ENTIER.

TYPE SUITE [ENTIER] :

environnement : BOOL,ENTIER

opérations — Lexique

constructeurs :

svide : SUITE[ENTIER] ← — suite vide

ajout : SUITE[ENTIER] ← SUITE[ENTIER],ENTIER — adjonction en tête

modificateurs :

sup : SUITE[ENTIER] ← SUITE[ENTIER],ENTIER — suppression du $n^{ième}$ élément

conc : SUITE[ENTIER] ← SUITE[ENTIER],SUITE[ENTIER] — concaténation

si-alors-sinon : SUITE[ENTIER] ← BOOL,SUITE[ENTIER],SUITE[ENTIER]

observateurs :

acc : ENTIER ← SUITE[ENTIER],ENTIER — accès au $n^{ième}$ élément

rang : ENTIER ← SUITE[ENTIER],ENTIER — rang d'un élément

taille : ENTIER ← SUITE[ENTIER] — nombre d'éléments

eq : BOOL ← SUITE[ENTIER],SUITE[ENTIER] — prédicat d'égalité

variables : sn,sn' : SUITE[ENTIER]
n,n',e,e' : ENTIER

équations :

définition des modificateurs :

sup(svide,e) = svide

sup(ajout(sn,n),zéro) = sn

sup(ajout(sn,n),suc(e)) = ajout(sup(sn,e),n)

conc(sn,svide) = sn

conc(sn,ajout(sn',n')) = ajout(conc(sn,sn'),n')

si vrai alors sn sinon sn' = sn

si faux alors sn sinon sn' = sn'

définition des observateurs :

acc(svide,e) = indef

acc(ajout(sn,n),zéro) = n

acc(ajout(sn,n),suc(e)) = acc(sn,e)

rang(svide,n) = indef

rang(ajout(sn,n),n') = si eq(n,n') alors zéro
sinon suc(rang(sn,n'))

taille(svide) = zéro

taille(ajout(sn,n)) = suc(taille(sn))

eq(svide,svide) = vrai

eq(svide,ajout(sn,n)) = faux

eq(ajout(sn,n),svide) = faux

eq(ajout(sn,n),ajout(sn',n')) = eq(n,n') et eq(sn,sn')

2.2.- Propriétés d'une spécification paramétrée

Une spécification paramétrée étant une extension de la spécification paramètre , les propriétés demandées sont la consistance et la complétude suffisante vis à vis de cette spécification. Il faut cependant

remarquer que parfois la présentation du paramètre formel, ne comporte pas de constructeurs (et donc pas de constantes), mais seulement des observateurs tels qu'un prédicat d'égalité ou une relation d'ordre. Dans ce cas la spécification du paramètre formel n'admet pas de termes sans variables : son algèbre initiale est vide ; Les propriétés demandées sont donc la consistance et la complétude suffisante sur les variables. Précisons ceci en introduisant la définition suivante :

Définition : La spécification paramétrée

$$SPECTP = SPEC + SPECP + <\{TP[S_P]\}, \Sigma_{TP}, E_{TP}>$$

est persistante si et seulement si elle est consistante et suffisamment complète sur les variables vis à vis de SPEC+SPECP.

Cette définition de la persistance coïncide avec celle de [ADJ, 82] et de [GAN, 83].

Une spécification obtenue par instanciation d'une spécification paramétrée persistante est consistante et suffisamment complète vis à vis de son environnement.

Théorème : Soit $SPECTP = SPEC + SPECP + <\{TP[S_P]\}, \Sigma_{TP}, E_{TP}>$ une spécification paramétrée par la présentation

$$SPECP = <S_P, \Sigma_P, E_P> \text{ sur l'environnement } SPEC = <S, \Sigma, E>.$$

Alors :

Pour toute présentation $SPECE = <S_E, \Sigma_E, E_E>$ sur l'environnement SPEC, paramètre effectif admissible de SPECTP pour un morphisme i de passage de paramètre, la spécification

$$SPECTE = SPEC + SPECE + <\{TP[S_E]\}, \Sigma_{TE}, E_{TE}>$$

où $\Sigma_{TE} = \Sigma_{TP}[P \leftarrow i(P), \forall P \in S_P]$ et

$$E_{TE} = E_{TP}[X_P \leftarrow X_{i(P)}][f \leftarrow i(f), \forall f \in \Sigma_P]$$

obtenue par instanciation de SPECP par SPECE est consistante et suffisamment complète vis à vis de SPEC+SPECE si et seulement si SPECTP est persistante.

Démonstration :

⇒ Si SPECTE est consistante et suffisamment complète
Alors SPECTP est persistante ?

1.- Supposons SPECTP non consistante sur les variables :

$$\exists t_1, t_2 \in T_{\Sigma \cup \Sigma_P}(X_s)_{s \in SUS_P}$$

$$E \cup E_P \cup E_{TP} \vdash\!\!\!- t_1 = t_2 \quad \text{et} \quad E \cup E_P \not\vdash\!\!\!- t_1 = t_2$$

Soit alors $SPECE = SPECP + <\phi, Var(t_1) \cup Var(t_2), \phi>$

La spécification obtenue en enrichissant SPECP avec les variables de t_1 et de t_2 comme constantes. SPECE est un paramètre admissible de SPECTP pour le morphisme i inclusion. Par hypothèse, SPECTE est consistante :

si $E \cup E_E \cup E_{TE} \vdash\!\!\!- i(t_1) = i(t_2)$ alors $E \cup E_E \vdash\!\!\!- i(t_1) = i(t_2)$

or $E_E = E_P$ et $i(t_1) = t_1$, $i(t_2) = t_2$ donc $E \cup E_P \vdash\!\!\!- t_1 = t_2$

ce qui est contraire à l'hypothèse.

2.- Supposons SPECP non suffisamment complète sur les variables :

$$\exists t \in T_{\Sigma \cup \Sigma_P \cup \Sigma_{TP,s'}}(X_s)_{s \in SUS_P},\ s' \in S \cup S_P$$

$$\text{tq } \nexists\, t' \in T_{\Sigma \cup \Sigma_P}(X_s)_{s \in SUS_P} \text{ avec } E \cup E_P \cup E_{TP} \vdash\!\!\!- t = t'.$$

Soit alors SPECE = SPECP + $\langle\phi, Var(t), \phi\rangle$ la spécification obtenue en enrichissant SPECP avec les variables de t comme constantes. SPECE est un paramètre admissible de SPECTP pour le morphisme i inclusion. Par hypothèse SPECTE est suffisamment complète :

$$i(t) \in T_{\Sigma \cup \Sigma_E \cup \Sigma_{TE}, s'} \qquad s' \in S \cup S_E$$

alors $\exists t' \in T_{\Sigma \cup \Sigma_E}$ tel que $E \cup E_E \cup E_{TE} \vdash\!\!\!- i(t) = t'$

Or $\Sigma_E = \Sigma_P \cup Var(t)$, $E_E = E_P$, $E_{TE} = T_{TP}$, $S_E = S_P$ et $i(t) = t$

donc $t' \in T_{\Sigma \cup \Sigma_P}(X_s)$ $s \in S \cup S_P$ et $E \cup E_P \cup E_{TP} \vdash\!\!\!- t = t'$

ce qui est contraire à l'hypothèse.

⇐ Si SPECTP est persistante

Alors SPECTE est consistante et suffisamment complète ?

Nous avons le diagramme suivant :

$$\begin{array}{ccc} SPEC+SPECP & \hookrightarrow & SPEC+SPECP+SPECTP \\ \downarrow i & & \downarrow i \\ SPEC+SPECE & \hookrightarrow & SPEC+SPECE+SPECTE \end{array}$$

où SPEC = $\langle S, \Sigma, E\rangle$
SPECP = $\langle S_P, \Sigma_P, E_P\rangle$
SPECE = $\langle S_E, \Sigma_E, E_E\rangle$
SPECTP = $\langle\{TP[S_P]\}, \Sigma_{TP}, E_{TP}\rangle$
SPECTE = $\langle\{TP[S_E]\}, \Sigma_{TE}, E_{TE}\rangle$

Pour alléger l'écriture nous posons

SP0 = SPEC+SPECP

SP1 = SPEC+SPECP+SPECTP
SP2 = SPEC+SPECE
SP3 = SPEC+SPECE+SPECTE

avec SP0 = $\langle S0, \Sigma0, E0\rangle$ $S0 = S \cup S_P$ $\Sigma0 = \Sigma \cup \Sigma_P$ $E0 = E \cup E_P$

SP1 = $\langle S1, \Sigma1, E1\rangle$ $S1 = S \cup S_P \cup \{TP[S_P]\}$ $\Sigma1 = \Sigma \cup \Sigma_P \cup \Sigma_{TP}$ $E1 = E \cup E_P \cup E_{TP}$

SP2 = $\langle S2, \Sigma2, E2\rangle$ $S2 = S \cup S_E$ $\Sigma2 = \Sigma \cup \Sigma_E$ $E2 = E \cup E_E$

SP3 = $\langle S3, \Sigma3, E3\rangle$ $S3 = S \cup S_E \cup \{TP[S_E]\}$ $\Sigma3 = \Sigma \cup \Sigma_E \cup \Sigma_{TE}$ $E3 = E \cup E_E \cup E_{TE}$

Le diagramme ci-dessus s'écrit alors :

$$\begin{array}{ccc} SP0 & \hookrightarrow & SP1 \\ \downarrow i & & \downarrow i \\ SP2 & \hookrightarrow & SP3 \end{array}$$

Nous notons X0 l'ensemble des variables de type dans S0
X1 S1
X2 S2
X3 S3

Remarquons que $S3 = S2 \cup i(S1)$ et $\Sigma3 = \Sigma2 \cup i(\Sigma1)$

Nous notons $T_{\Sigma,s}(X)$ l'ensemble des termes sur Σ, avec des variables X, de sorte s.

Nos hypothèses sont

- la complétude suffisante sur les variables de SP1 vis à vis de SP0 :

$$\forall t \in T_{\Sigma1,s}(X0),\ s \in S0 \qquad \exists t' \in T_{\Sigma0}(X0)$$

$$\text{tq } E1 \vdash\!\!\!- t = t'$$

- la consistance sur les variables de SP1 vis à vis de SP0

$$\forall t,t' \in T_{\Sigma0}(X0) \qquad E1 \vdash\!\!\!- t = t' \Rightarrow E0 \vdash\!\!\!- t = t'.$$

Pour démontrer la complétude suffisante et la consistance de SP3 vis à vis de SP2, nous devons utiliser nos hypothèses. Ne pouvant les utiliser que sur des termes ne contenant aucune opération du paramètre effectif $(\Sigma 2\text{-}i(\Sigma 0))$, nous introduisons la notion de "terme normalisé" : un terme de $T_{\Sigma 3}(X2)$ est dit normalisé si tous ses sous-termes de type appartenant au type paramètre effectif (S2), sont des termes du type paramètre effectif $(T_{\Sigma 2}(X2))$.

Définition : Un terme $t \in T_{\Sigma 3}(X3)$ est dit normalisé si et seulement si pour toute adresse d dans t telle que le sous-terme de t d'adresse d noté t(d) de sorte dans S2 soit un terme de $T_{\Sigma 2}(X2)$:

$$\forall d \quad sorte(t(d)) \in S2 \Rightarrow t(d) \in T_{\Sigma 2}(X2)$$

Nous définissons une fonction n de normalisation associant à tout terme de sorte dans le paramètre effectif, un terme du type paramètre effectif.

Cette fonction est étendue aux termes de sorte quelconque.

Proposition : Il existe une fonction n appelée de normalisation qui associe à tout terme t de $T_{\Sigma 3,s}(X2)$, $s \in S2$, un terme de $T_{\Sigma 2}(X2)$. Alors $i(E1) \vdash\!\!\!\!- \; t = n(t)$.

La fonction n est étendue aux termes de sorte quelconque :

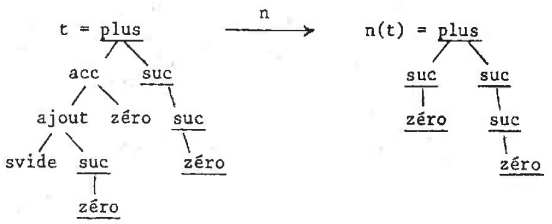
$$\forall t \in T_{\Sigma 3}(X2), \; \bar{n}(t) \text{ est normalisé}$$

où $\bar{n} : T_{\Sigma 3}(X2) \to T_{\Sigma 3}(X2)$ est l'extension de

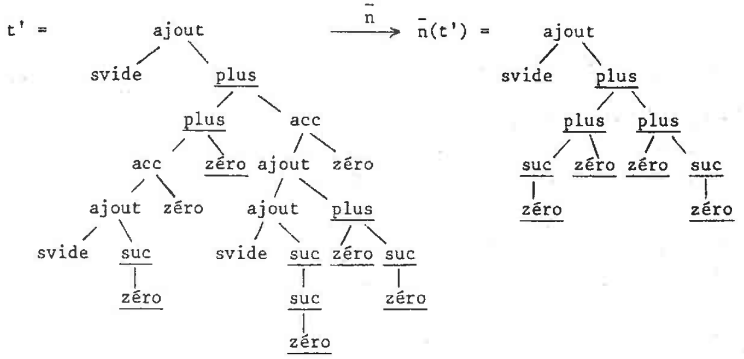
$n = T_{\Sigma 3,s}(X2) \to T_{\Sigma 2}(X2)$, $s \in S2$.

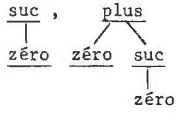
Exemples : Reprenons le type paramétré SUITE[P] et son paramètre effectif admissible ENTIER. Prenons un terme t de sorte ENTIER (noté sous forme d'arbre pour plus de lisibilité).

Nous soulignons les opérations du type paramètre effectif :



t est de sorte ENTIER n(t) appartient à T_{Σ} ENTIER



t' est de sorte SUITE[ENTIER]. Les sous-termes de t' du paramètre effectif :  , ne sont pas modifiés par la normalisation.

Démonstration : Soit un terme $t \in T_{\Sigma 3,s}(X2)$, $s \in S2$.

Définissons par induction n(t) :

- Si t est une constante ou une variable de X2 (de sorte dans S2) soit $t \in \Sigma 2 \cup X2$ alors $n(t) = t$

soit $t \in \Sigma 3\text{-}\Sigma 2 = i(\Sigma 1)$ alors il existe une constante $u \in \Sigma 1$ telle que $t = i(u)$ et u de sorte dans $i(S0)$.
Par complétude suffisante de SP1 vis à vis de SP0, il existe un terme $u' \in T_{\Sigma 0}$ tel que $E1 \vdash\!\!\!- u = u'$.
Donc $i(E1) \vdash\!\!\!- i(u) = i(u')$

$i(E1) \vdash\!\!\!- t = i(u')$

$i(u') \in T_{i(\Sigma 0)}$. Posons $n(t) = i(u')$

Nous avons bien $i(E1) \vdash\!\!\!- t = n(t)$.

- Si $t = f(t_1 \ldots t_n)$, $t \in T_{\Sigma 3,s}(X2)$, $s \in S2$
soit $f \in \Sigma 2$ alors tous les t_i sont de sorte dans $S2$.
Par hypothèse d'induction, ils peuvent être normalisés :

$n(t_i) \in T_{\Sigma 2}(X2)$ et $i(E1) \vdash\!\!\!- t_i = n(t_i)$ $1 \leq i \leq n$

Donc $f(n(t_1),\ldots n(t_n)) \in T_{\Sigma 2}(X2)$ et

$i(E1) \vdash\!\!\!- t = f(n(t_1),\ldots n(t_n))$

Posons alors $n(t) = f(n(t_1),\ldots n(t_n))$

soit $f \in i(\Sigma 1)$. Les t_i peuvent être normalisés en des termes dont tous les sous-termes de sorte dans $S2$ appartiennent à $T_{\Sigma 2}(X2)$.
Remarquons que f appartenant à $i(\Sigma 1)$ tous ses paramètres sont de sorte dans $i(S1)$. Alors tout sous-terme de sorte dans $S2\text{-}i(S0)$ est inclu dans un sous-terme de sorte dans $i(S0)$.
Remplaçons les sous-termes maximaux des $\bar{n}(t_i)$ de sorte dans $i(S0)$ par des variables : le terme obtenu ne contient alors plus que des opérations de $i(\Sigma 1)$ et des variables de $i(X0)$.
Nous pouvons alors appliquer la complétude suffisante sur les variables de SP1 vis à vis de SP0 pour obtenir, modulo les équations de $i(E1)$ un terme de $T_{i(\Sigma 0)}(i(X0))$. Puis en affectant aux variables introduites, les sous-termes de $T_{\Sigma 2}(X2)$, nous obtenons un terme de $T_{\Sigma 2}(X2)$, équivalent à t modulo les équations de $i(E1)$:
Soit $u \in T_{i(\Sigma 1),s}(i(X0))$, $s \in i(S0)$ un terme tel qu'il existe une affectation $a : i(X0) \to T_{\Sigma 2}(X2)$, prolongée par $\bar{a}$ sur les

termes de $T_{\Sigma 3}(X2)$ telle que

$\bar{a}(u) = f(n(t_1)\ldots,n(t_n))$

(u est égal à $f(n(t_1),\ldots n(t_n))$ où les sous-termes maximaux de sorte dans $i(S0)$ ont été remplacés par des variables de $i(X0)$).
$u \in T_{i(\Sigma 1),s}(i(X0))$, $s \in i(S0)$. Par complétude suffisante sur les variables de SP1 vis à vis de SP0, il existe un terme $u' \in T_{i(\Sigma 0)}(i(X0))$ tel que $i(E1) \vdash\!\!\!- u = u'$.
$\bar{a}(u')$ est un terme de $T_{\Sigma 2}(X2)$ et $i(E1) \vdash\!\!\!- \bar{a}(u) = \bar{a}(u')$
or $\bar{a}(u) = f(n(t_1)\ldots n(t_n))$, $i(E1) \vdash\!\!\!- t_i = n(t_i)$ $1 \leq i \leq n$
Donc $i(E1) \vdash\!\!\!- f(t_1\ldots t_n) = \bar{a}(u')$
c'est-à-dire $i(E1) \vdash\!\!\!- t = \bar{a}(u')$
posons alors $n(t) = \bar{a}(u')$. □

Propriétés de la fonction de normalisation :

Propriété 1 : La fonction $\bar{n}$ de normalisation ne modifie dans un terme que les sous-termes de sorte dans $S2$, et dont la racine est une opération de $i(\Sigma 1\text{-}\Sigma 0)$.

Propriété 2 : La fonction $\bar{n}$ de normalisation ne modifie pas dans un terme, les sous-termes appartenant au paramètre effectif : soient $t \in T_{\Sigma 3}(X2)$ et d une adresse dans t telle que $t(d) \in T_{\Sigma 2}(X2)$ ($t(d)$ est le sous-terme de t d'adresse d)
Alors $\bar{n}(t) = \bar{a}(n(t[d \leftarrow x]))$ où $\bar{a} : X2 \to T_{\Sigma 2}$ est une affectation définie par $\bar{a}(x) = t(d)$.

Propriété 3 : Si $t \in T_{i(\Sigma 1)}(i(X0))$
Alors $n(t) \in T_{i(\Sigma 0)}(i(X0))$

Propriété 4 : Le terme normalisé d'un terme sans variables, est un terme sans variables :

Si $t \in T_{\Sigma 3}$ alors $\bar{n}(t) \in T_{\Sigma 3}$

Si $t \in T_{\Sigma 3,s}$, $s \in S2$ alors $n(t) \in T_{\Sigma 2}$.

Nous démontrons maintenant la complétude suffisante et la consistance de SP3 vis à vis de SP2.

Démonstration de la complétude suffisante de SP3 vis à vis de SP2.

Nous voulons démontrer que :

$$\forall t \in T_{\Sigma 3,s}(X2),\ s \in S2$$

$$\exists t' \in T_{\Sigma 2}(X2) \quad tq \quad E3 \vdash\!\!\!- \ t = t'$$

Or si $t \in T_{\Sigma 3,s}(X2)$, alors t peut être normalisé par la fonction n et $n(t) \in T_{\Sigma 2}(X2)$ et nous avons même : $i(E1) \vdash\!\!\!- \ t = n(t)$.

$$\forall t \in T_{\Sigma 3,s}(X2),\ s \in S2 \quad \exists t' \in T_{\Sigma 2}(X2) \quad tq$$

$$i(E1) \vdash\!\!\!- \ t = t'.$$

Démonstration de la consistance de SP3 vis à vis de SP2.

Nous voulons démontrer que :

$$\forall t1,t2 \in T_{\Sigma 2} \qquad E3 \vdash\!\!\!- \ t1 = t2 \Rightarrow E2 \vdash\!\!\!- \ t1 = t2$$

Schéma de la démonstration :

1.- Nous démontrons que $\forall t1,t2 \in T_{\Sigma 2}$

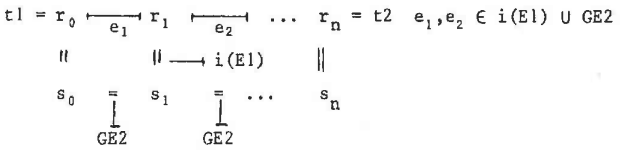
$$E3 \vdash\!\!\!- \ t1 = t2 \Rightarrow i(E1) \cup GE2 \vdash\!\!\!- \ t1 = t2$$

où $GE2 = \{<t,t'>,\ tq\ \ t,t' \in T_{\Sigma 2}$ et $E2 \vdash\!\!\!- \ t = t'\}$.

2.- Puis nous démontrerons que

$$i(E1) \cup GE2 \vdash\!\!\!- \ t1 = t2 \Rightarrow GE2 \vdash\!\!\!- \ t1 = t2$$

en construisant une démonstration de $t1 = t2$ dans GE2, parallèle de celle dans $i(E1) \cup GE2$:



Cette démonstration $GE2 \vdash\!\!\!- \ s_0 = s_1 \ldots = s_n$ sera définie par récurrence. A l'étape j deux cas peuvent se présenter :

1er Cas : $e_j \in i(E1)$

2ème Cas : $e_j \in GE2$

1.- Soient $t1,t2 \in T_{\Sigma 2}$

$E3 \vdash\!\!\!- \ t1 = t2$

Posons $GE2 = \{<t,t'>/t,t' \in T_{\Sigma 2},\ E2 \vdash\!\!\!- \ t = t'\}$

A-t-on $i(E1) \cup GE2 \vdash\!\!\!- \ t1 = t2$?

C'est-à-dire : qu'elle que soit une affectation des variables d'une équation de E2 par des termes de $T_{\Sigma 3}$, a-t-on une équation démontrable à l'aide des équations de $GE2 \cup i(E1)$?

Soit $r = s$ une équation de E2, $r,s \in T_{\Sigma 2}(X2)$

$\forall a : X2 \to T_{\Sigma 3,s}$, $s \in S2$ une affectation

$$i(E1) \cup GE2 \vdash\!\!\!- \ \bar{a}(r) = \bar{a}(s) \ ?$$

où $\bar{a}$ est la prolongation de a sur les termes de $T_{\Sigma 2}(X2)$.

Soit $x \in Var(r) \cup Var(s)$ une variable de l'équation $r = s$.

$a(x)$ est un terme de $T_{\Sigma 3,s}$ de sorte s dans S2.

Par la propriété de complétude suffisante démontrée, il existe un terme tx de $T_{\Sigma 2}$ tel que $i(E1) \vdash\!\!\!- \ a(x) = tx$.

Soit a', l'affectation associant à x ce terme tx :

$a' : X2 \to T_{\Sigma 2}$ définie par $a'(x) = tx$.

Par définition de GE2, $\langle \bar{a}'(r), \bar{a}'(s) \rangle \in GE2$.

Or si $i(E1) \vdash a(x) = tx$ alors $i(E1) \vdash \bar{a}(r) = \bar{a}'(r)$ et $\bar{a}(s) = \bar{a}'(s)$

Nous avons donc :

$$i(E1) \vdash \bar{a}(r) = \bar{a}'(r) \text{ et } \bar{a}(s) = \bar{a}'(s)$$

$$GE2 \vdash \bar{a}'(r) = \bar{a}'(s)$$

Donc $i(E1) \cup GE2 \vdash \bar{a}(r) = \bar{a}(s)$

et ceci quel que soit l'affectation $a : X2 \to T_{\Sigma 3}$.

2.- Soient $t1, t2 \in T_{\Sigma 2}$

$i(E1) \cup GE2 \vdash t1 = t2$

A-t-on $GE2 \vdash t1 = t2$?

$i(E1) \cup GE2 \vdash t1 = t2$ signifie qu'il existe une démonstration où à chaque étape une équation de $i(E1) \cup GE2$ est utilisée (nous notons $r_{j-1} \overset{}{\vdash\!\!\!\mapsto}_{e_j} r_j$ la $j^{\text{ième}}$ étape).

$$t1 = r_0 \overset{}{\vdash\!\!\!\mapsto}_{e_1} r_1 \overset{}{\vdash\!\!\!\mapsto}_{e_2} \ldots r_{j-1} \overset{}{\vdash\!\!\!\mapsto}_{e_j} r_j \ldots \overset{}{\vdash\!\!\!\mapsto}_{e_n} r_n = t2$$

$t1$ et $t2$ appartenant à $T_{\Sigma 2}$, les termes $r_1 \ldots r_{n-1}$ appartiennent à $T_{\Sigma 3,s}$ et sont de sorte s dans S2. r_0 et r_n sont égaux à $t1$ et $t2$ resp.

Nous allons construire une démonstration parallèle

$$t1 = s_0 = s_1 = s_n = t2$$

où $GE2 \vdash s_{j-1} = s_j$ (par application de plusieurs équations éventuellement)

Cette démonstration vérifie les conditions suivantes :

$$a - \quad s_0 = t1, \; s_n = t2$$

$$b - \quad i(E1) \vdash r_j = s_j \qquad 1 \leq j < n$$

$$c - \quad GE2 \vdash s_{j-1} = s_j \qquad 1 \leq j \leq n$$

Pour vérifier b nous posons $s_j = n(r_j) \qquad 0 \leq j \leq n$.

La condition a est vérifiée par la propriété 2 : $s_0 = t1$ et $s_n = t2$.

La condition c est-elle alors vérifiée ?

Démonstration par récurrence :

$r_0, r_n \in T_{\Sigma 2}$ donc $s_0 = n(r_0) = r_0$ et $s_n = n(r_n) = r_n$ par la propriété 1 de la fonction n.

Supposons que nous avons construit les j-1 premières étapes de la démonstration. Construisons la $j^{\text{ième}}$ étape :

$$\begin{array}{ccc} & r_{j-1} & \overset{}{\vdash\!\!\!\mapsto}_{e_j} \; r_j \\ i(E1) \vdash & \| & \\ & s_{j-1} = n(r_{j-1}) & s_j \text{ tel que } s_j = n(r_j) \\ & & \text{et } GE2 \vdash s_{j-1} = s_j \end{array}$$

1er Cas : $e_j \in i(E1)$

Nous utilisons le lemme suivant :

Lemme : $\forall t, t' \in T_{\Sigma 3,s}$, $s \in S2$, deux termes de sorte dans S2, tels qu'il existe une équation e de $i(E1)$ telle que

$$t \overset{}{\vdash\!\!\!\mapsto}_{e} t' \Rightarrow i(E0) \vdash n(t) = n(t').$$

Par définition de la fonction n de normalisation, nous avons

$$i(E1) \vdash t = n(t) \text{ et } t' = n(t').$$

En appliquant ce lemme et en posant

$$s_j = n(r_j), \; i(E1) \vdash r_j = s_j$$

nous avons donc

$$r_{j-1} \overset{}{\vdash\!\!\!\mapsto}_{e_j} r_j, \; e_j \in i(E1) \Rightarrow i(E0) \vdash s_{j-1} = s_j$$

$$\text{et donc } GE2 \vdash s_{j-1} = s_j$$

2ème Cas : $e_j \in GE2$

$e_j = \langle r, r' \rangle$ où $r, r' \in T_{\Sigma 2}$

$r_{j-1} \overset{e_j}{\longmapsto} r_j$ signifie que r est un sous-terme de r_{j-1} et que r' est un sous-terme de r_j.

Soit d l'adresse de r dans r_{j-1} et de r' dans r_j :

$$r_j = r_{j-1}[d \leftarrow r']$$

Par la propriété 2 de la fonction n de normalisation, les sous-termes r et r' de r_{j-1} et r_j resp, ne sont pas modifiés par la normalisation de r_{j-1} et de r_j :

Soient $a1 : x \to r$ une application définie par $a1(x) = r$

$a2 : x \to r'$ une application définie par $a2(x) = r'$

et $u \in T_{\Sigma 3, s}$ tel que $u = r_{j-1}[d \leftarrow x] = r_j[d \leftarrow x]$

$n(r_{j-1}) = a1(n(u))$ et $n(r_j) = a2(n(u))$

r et r' étant des sous-termes de $n(r_{j-1})$ et de $n(r_j)$ resp.

$$GE2 \vdash n(r_{j-1}) = n(r_j)$$

$s_{j-1} = n(r_{j-1})$ et $s_j = n(r_j)$ vérifient donc les conditions demandées.

Nous avons donc construit une démonstration de $t1 = t2$ n'utilisant que des équations de GE2

$$t1 = s_0 = s_1 = \ldots s_{j-1} = s_j \ldots = s_n = t2$$
$$\quad\ \downarrow \quad \downarrow \qquad\quad \downarrow$$
$$\quad GE2 \quad GE2 \qquad GE2$$

Donc $GE2 \vdash t1 = t2$

Or $t1, t2 \in T_{\Sigma 2}$ donc $\langle t1, t2 \rangle \in GE2$ et on a bien

$$E2 \vdash t1 = t2 \qquad \square$$

Il nous reste à démontrer le lemme suivant :

Lemme : $\forall t, t' \in T_{\Sigma 3, s}$, $s \in S2$ tels que

$$\exists e \in i(E1)$$

$$t \overset{e}{\longmapsto} t' \Rightarrow i(E0) \vdash n(t) = n(t')$$

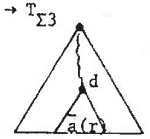
Démonstration : Posons $e = \langle r, r' \rangle$, $e \in i(E1)$ alors $r, r' \in T_{i(\Sigma 1)}(i(X1))$.

$t \overset{e}{\longmapsto} t'$ alors il existe une adresse d dans t et dans t' et une application $a : Var(r) \cup Var(r') \to T_{\Sigma 3}$

tels que $t(d) = \bar{a}(r)$ $\qquad t =$

$t'(d) = \bar{a}(r')$

et $t' = t[d \leftarrow \bar{a}(r')]$



Quelle que soit x une variable de r, $a(x)$ est un sous-terme de t : $a(x) \in T_{\Sigma 3}$ et est de sorte dans $i(S1)$.

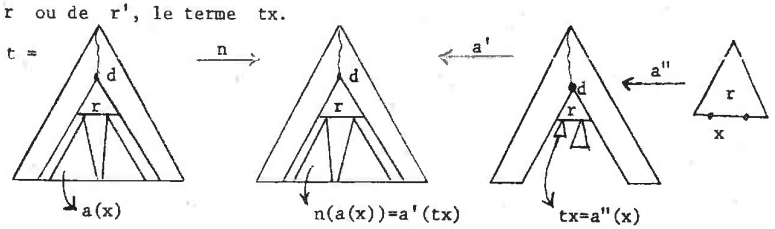
$\bar{n}(a(x))$ est la normalisation de $a(x)$: tout sous-terme de $\bar{n}(a(x))$ de sorte dans $i(S0)$ est un terme de $T_{\Sigma 2}$.

Soit t_x le terme égal à $\bar{n}(a(x))$ où tous les sous-termes maximaux de sorte dans $i(S0)$ sont remplacés par des variables de $i(X0)$ et soit l'affectation associant à ces variables les termes de $T_{\Sigma 2, s}$, $s \in i(S0)$ qu'elles remplacent :

$a' : Var(tx) \to T_{\Sigma 2, s}$, $s \in i(S0)$

tq $a'(tx) = \bar{n}(a(x))$ et $tx \in T_{i(\Sigma 1), s}(i(X0))$, $s \in i(S1)$.

Nous définissons alors l'affectation a'' qui associe à toute variable x de r ou de r', le terme tx.



$a'' : Var(r) \cup Var(r') \to T_{i(\Sigma 1)}(i(X0))$

$a''(x) = tx$ et $a''(r) \in T_{i(\Sigma 1)}(i(X0))$

Nous avons un terme : $a''(r)$, instance de l'un des termes de l'équation e et qui ne possède plus d'opérations de $\Sigma 2 - i(\Sigma 0)$.

Pour appliquer l'hypothèse de consistance de $i(SP1)$ vis à vis de $i(SP0)$, il nous faut un terme de $T_{i(\Sigma 0)}$. Nous pouvons l'obtenir par complétude suffisante à partir d'un terme de $T_{i(\Sigma 1)}(i(X0))$ de sorte dans $i(S0)$. $a''(r) \in T_{i(\Sigma 1)}(i(X0))$ mais n'est pas forcément de sorte dans $i(S0)$. Nous allons donc par le même procédé, enlever d'un sous-terme de t contenant $t(d)$, les opérations de $\Sigma 2 - i(\Sigma 0)$. Il nous faut cependant ne plus toucher au terme $a''(r)$.

Soit d' la plus grande adresse dans t vérifiant

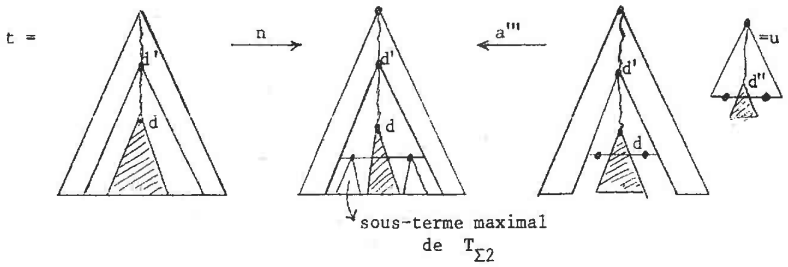
- $d' \leq d$ (c'est-à-dire $t(d)$ inclu dans $t(d')$)
- sorte $(t(d')) \in i(S0)$

$t(d') \in T_{\Sigma 3,s}$, $s \in i(S0)$

$t(d')$ peut être normalisé en un terme de $T_{\Sigma 2}$. Mais il nous faut sauvegarder le sous-terme $t(d)$.

Nous normalisons tous les sous-termes de $t(d')$ ne contenant pas d. Soit u le résultat de cette normalisation et de la substitution des sous-termes maximaux de $T_{\Sigma 2}$ obtenus par des variables (distinctes de celles déjà introduites).

Soit a''' l'affectation associant aux variables les sous-termes qu'elles remplacent. $a''' : i(X0) \to T_{\Sigma 2}$.



d'' est l'adresse du sous-terme d dans le sous-terme d' : $d = d'd''$.

Le terme $v = u[d'' \leftarrow a''(r)]$ est un terme de $T_{i(\Sigma 1)}(i(X0))$ de sorte $i(S0)$, contenant une instance de r.

$v = u[d'' \leftarrow a''(r)]$ $\quad v, v' \in T_{i(\Sigma 1),s}(i(X0))$ $\quad s \in i(S0)$

$v' = u[d'' \leftarrow a''(r')]$

$\langle r, r' \rangle \in i(E1)$ $\quad i(E1) \vdash\!\!\!- v = v'$

Par normalisation nous obtenons deux termes de $T_{i(\Sigma 0)}(i(X0))$ équivalents à v et à v' modulo des équations de $i(E1)$

$$i(E1) \vdash\!\!\!- v = n(v) \ \underline{\text{et}} \ v' = n(v')$$

$$n(v), n(v') \in T_{i(\Sigma 0)}(i(X0))$$

donc $\quad i(E1) \vdash\!\!\!- n(v) = n(v')$

Nous pouvons maintenant appliquer l'hypothèse de consistance de $SP1$ vis-à-vis de $SP0$: $n(v), n(v') \in T_{i(\Sigma 0)}(i(X0))$

$$i(E1) \vdash\!\!\!- n(v) = n(v') \Rightarrow i(E0) \vdash\!\!\!- n(v) = n(v')$$

Nous pouvons maintenant affecter aux variables introduites leurs termes associés par a' et par a'''.

$$i(E0) \vdash\!\!\!- a''' \circ a' \circ n(v) = a''' \circ a' \circ n(v')$$

$n(v)$ et $n(v')$ appartiennent à $T_{i(\Sigma 0)}(i(X0))$

$a''' \circ a' \circ n(v)$ et $a''' \circ a' \circ n(v')$ appartiennent à $T_{\Sigma 2}$.

Enfin

$$\begin{aligned} i(E0) \vdash\!\!\!- t[d' \leftarrow a''' \circ a' \circ n \circ u[d'' \leftarrow a''(r)]] \\ = t[d' \leftarrow a''' \circ a' \circ n \circ u[d'' \leftarrow a''(r')]] \\ = t'[d' \leftarrow a''' \circ a' \circ n \circ u[d'' \leftarrow a''(r')]] \end{aligned}$$

Par construction de la fonction n de normalisation (propriété 2 de n) nous avons

$$n(t) = n(t[d' \leftarrow a''' \circ a' \circ n.u[d'' \leftarrow a''(r)]])$$

$$n(t') = n(t[d' \leftarrow a''', a'.n.u[d'' \leftarrow a''(r')]])$$

et donc $i(EO) \vdash\!\!\!- n(t) = n(t')$

ce qui achève la démonstration du théorème. □

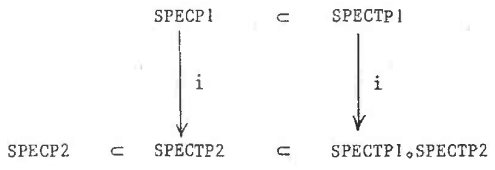
La propriété de persistance est stable par composition des spécifications paramétrées.

Proposition : Soient SPECTP1 et SPECTP2 deux spécifications paramétrées persistantes vis-à-vis de leur paramètre formel SPECP1 et SPECP2 sur le même environnement SPEC.

Si SPECTP2 est un paramètre effectif admissible de SPECTP1,

Alors la spécification SPECTP1∘SPECTP2 paramétrée par SPECP2 obtenue par instanciation de SPECP1 par SPECTP2 est persistante vis-à-vis de SPECP2.

Diagramme :



Démonstration :

SPEC = $<S,\Sigma,E>$

SPECP1 = $<S_{P1},\Sigma_{P1},E_{P1}>$ SPECP2 = $<S_{P2},\Sigma_{P2},E_{P2}>$

SPECTP1 = SPEC+SPECP1+$<\{TP1[S_{P1}]\},\Sigma_{TP1},E_{TP1}>$

SPECP2 = $<S_{P2},\Sigma_{P2},E_{P2}>$

SPECTP2 = SPEC+SPECP2+$<\{TP2[S_{P2}]\},\Sigma_{TP2},E_{TP2}>$

SPECTP2 est un paramètre admissible pour SPECTP1 et i est le morphisme de passage de paramètres.

$$SPECTP1 \circ SPECTP2 = SPEC+SPECP2+<\{TP2[S_{P2}]\},\Sigma_{TP2},E_{TP2}>+$$
$$+<\{TP1[TP2[S_{P2}]],\Sigma_{TP1\circ TP2},E_{TP1\circ TP2}>$$

où $\Sigma_{TP1\circ TP2} = \Sigma_{TP1}[P1 \leftarrow TP2[S_{P2}]]$

$E_{TP1\circ TP2} = E_{TP1}[X_{P1} \leftarrow X_{TP2[S_{P2}]}][f \leftarrow i(f), \forall f \in \Sigma_{TP2}]$

d'après la 1ère proposition, si SPECTP2 est un paramètre admissible alors SPECTP1∘SPECTP2 est consistante et suffisamment complète vis-à-vis de SPECTP2.

SPECTP2 est consistante et suffisamment complète sur les variables vis-à-vis de SPECP2.

Donc SPECTP1∘SPECTP2 est consistante et suffisamment complète sur les variables vis-à-vis de SPECP2. □

Exemple : SUITE[P] est une spécification paramétrée persistante.

Si SUITE[P] est un paramètre admissible de SUITE[P] alors SUITE[SUITE[P]] est une spécification paramétrée persistante.

2.3.- Définition d'opérateurs

Nous voulons disposer parmi les opérations d'un type paramétré, d'opérations qui dépendront d'une opération du paramètre effectif.

Cette opération sera associée à une opération du paramètre formel, spécifiée uniquement par un profil.

Il s'agit d'opérations portant sur l'ensemble des éléments d'un objet structuré : nous définirons par exemple sur le type paramétré SUITE deux opérateurs :

• iter, opérateur d'itération permettant d'appliquer une opération fct du paramètre effectif, à tous les éléments d'une suite vérifiant une propriété définie par un prédicat prop du paramètre effectif.

• somme, opérateur calculant la somme des valeurs entières calculées par une opération fcte du paramètre effectif sur chaque élément d'une suite.

Nous définissons de tels opérateurs comme enrichissement de la spécification paramétrée primitive, sur un enrichissement du paramètre formel :

Soit $SPEC = \langle S,\Sigma,E\rangle$ l'environnement.

$SPECP = \langle S_p,\Sigma_p,E_p\rangle$ le paramètre formel (primitif)

$SPECTP = SPEC+SPECP+\langle\{TP[S_p]\},\Sigma_{TP},E_{TP}\rangle$ la spécification paramétrée (primitive).

$SPECPE = SPECP+\langle\emptyset,F_p,E_{F_p}\rangle$ est un enrichissement du paramètre formel par les opérations F_p

$SPECTPE = SPEC+SPECPE+\langle\{TP[S_p]\},\Sigma_{TP},E_{TP}\rangle +\langle\emptyset,O[F_p],E_{O[F_p]}\rangle$ est un enrichissement de la spécification paramétrée par les opérations $O[F_p]$ dépendant des opérations F_p du paramètre formel enrichi.

L'utilisation de tels opérateurs nécessite la définition d'une extension du morphisme de passage de paramètre, associant une opération du paramètre effectif, aux opérations F_p de l'extension du paramètre formel.

Notations : Pour plus de lisibilité nous utiliserons ces opérateurs indéxés par l'opération du paramètre effectif choisie.

Exemple : $iter_{plus(3,\cdot),infeg(10,\cdot)}(sn)$

est l'opération d'itération définie sur les suites d'entiers appliquant à chaque entier n de la suite sn (représenté par "$\cdot$") vérifiant le prédicat infeg(10,n), l'opération plus(3,n).

Les opérateurs sont définis dans les présentations des types paramétrés (ou constructeurs de types) SUITE, ENSEMBLE et TABLE au paragraphe II.2.2.

3.- REPRESENTATIONS

Les représentations ont été généralement étudiées comme moyen de transformer par étapes successives une spécification en un programme ([ADJ, 78], [FIN 79]).

Chaque représentation précisant certains détails d'implantation, rend la spécification moins "abstraite", tout en en conservant les propriétés. C'est pour cela que la plupart des auteurs parlent d'implantation. Nous abordons les représentations sous un angle différent. Nous les étudions comme moyen de modifier la structure des objets d'un type dans le but d'optimiser ses opérations, de le transformer en vue d'une comparaison à un autre type ou d'une manière générale, d'en donner une vue différente.

De manière intuitive, représenter un type source par un type cible, c'est définir les objets et les opérations du type source à l'aide d'objets et d'opérations du type cible. Nous souhaitons conserver les propriétés du type source décrites par ses équations et bénéficier des propriétés du type cible. Cependant conserver toutes les propriétés limite beaucoup les possibilités et n'est pas toujours nécessaire. Aussi nous n'étudions que des représentations faibles, ne conservant que les propriétés observables, c'est-à-dire les valeurs des termes de sorte dans l'environnement.

Après avoir défini les représentations faibles nous étudions les exemples de représentation des suites par les ensembles et vice versa. Nous définissons au paragraphe 3.2 les représentations faibles paramétrées.

3.1.- Définition des représentations faibles

Soient TS et TC deux types abstraits spécifiés sur le même environnement $SPEC = \langle S,\Sigma,E\rangle$ par $SPEC_{TS} = SPEC+\langle S_{TS},\Sigma_{TS},E_{TS}\rangle$ et

$SPEC_{TC} = SPEC+\langle S_{TC},\Sigma_{TC},E_{TC}\rangle$ respectivement.

Définition : Une représentation faible de TS par TC sur SPEC est un morphisme de signatures d'une restriction consistante et suffisamment complète (vis à vis de SPEC), SPECTSR de SPECTS vers un enrichissement consistant et suffisamment complet (vis à vis de SPEC) SPECTCE de SPECTC :

$r : SPECTSR \rightarrow SPECTCE$

où $SPECTSR = SPEC+<S_{TSR}, \Sigma_{TSR}, E_{TSR}>$

avec $\Sigma_{TSR} \subset \Sigma_{TS}$ et $E_{TS} \vdash\!\!\!- E_{TSR}$

$SPECTCE = SPEC+SPECTC+<\emptyset, \Sigma_{TCE}, E_{TCE}>$

tel que $\forall t1,t2 \in T_{\Sigma \cup \Sigma_{TSR}, s}$ $s \in S$

$E \cup E_{TSR} \vdash\!\!\!- T1 = t2 \leftrightarrow E \cup E_{TC} \cup E_{TCE} \vdash\!\!\!- r(t1) = r(t2)$

Cette définition autorise de représenter

- plusieurs objets par un même objet du type cible
  exemple : les suites de caractères abc, acb, bca, aabc, par l'ensemble de caractères {a,b,c}
- un objet par plusieurs objets du type cible
  exemple : l'ensemble de caractères {a,b,c} par les suites abc, acb, bca, aabc

à condition de conserver le même comportement vis à vis des sortes de l'environnement ("behavioral abstraction"de [SAN, WIR, 83] et [GAN, 83]).

exemple : le prédicat d'appartenance est conservé.

Exemples de représentations faibles :

Nous représentons les suites d'entiers par des ensembles d'entiers et vice versa, respectivement spécifiés comme suit.

Une suite d'entiers est ordonnée et peut comporter plusieurs occurrences d'un élément.

TYPE SUITE-ENT

environnement : BOOL,ENT

opérations

constructeurs	svide : SUITE-ENT	suite vide
	ajout : SUITE-ENT ← SUITE-ENT,ENT	adjonction d'un élément en tête de suite
modificateurs	sup : SUITE-ENT ← SUITE-ENT,ENT	suppression d'un élément donné par son rang
	conc : SUITE-ENT ← SUITE-ENT, SUITE-ENT	concaténation de 2 suites
	supelt : SUITE-ENT ← SUITE-ENT,ENT	suppression de toutes les occurrences d'un élément
observateurs	tête : ENT ← SUITE-ENT	accès au dernier élément ajouté
	taille : ENT ← SUITE-ENT	nombre d'éléments d'une suite
	app : BOOL ← SUITE-ENT,ENT	appartenance d'un élément donné à une suite
	svide? : BOOL ← SUITE-ENT	une suite est-elle vide ?
	eq : BOOL ← SUITE-ENT,SUITE-ENT	prédicat d'égalité

variables : sn,sn' : SUITE-ENT

n,n',e,e' : ENT

équations

définition des modificateurs

sup(svide,e)	= svide
sup(ajout(sn,n),zéro)	= sn
sup(ajout(sn,n),suc(e))	= ajout(sup(sn,n),e)
conc(sn,svide)	= sn
conc(sn,ajout(sn',n))	= ajout(conc(sn,sn'),n)
supelt(svide,n)	= svide
supelt(ajout(sn,n),n')	= si eq(n,n') alors supelt(sn,n') sinon ajout(supelt(sn,n'),n)

Définition des observateurs :

tête(svide)	= indef
tête(ajout(sn,n))	= n
taille(svide)	= zéro
taille(ajout(sn,n))	= suc(taille(sn))
app(svide,n)	= faux
app(ajout(sn,n),n')	= si eq(n,n') alors vrai sinon app(sn,n')
svide?(svide)	= vrai
svide?(ajout(sn,n))	= faux
eq(svide,svide)	= vrai
eq(svide,ajout(sn,n))	= faux
eq(ajout(sn,n),svide)	= faux
eq(ajout(sn,n),ajout(sn',n'))	= eq(n,n') et eq(sn,sn')

Un ensemble d'entiers n'est pas ordonné, et seule une occurrence de chaque élément est prise en compte.

TYPE ENS-ENT

environnement : BOOL,ENT

opérations

constructeur	evide	: ENS-ENT	ensemble vide
	adj	: ENS-ENT ← ENS-ENT,ENT	adjonction d'un élément
modificateurs	supe	: ENS-ENT ← ENS-ENT,ENT	suppression d'un élément
	union	: ENS-ENT ← ENS-ENT,ENS-ENT	union de deux ensembles
observateurs	taillee	: ENT ← ENS-ENT	nombre d'éléments
	appe	: BOOL ← ENS-ENT,ENT	prédicat d'appartenance

observateurs	evide?	: BOOL ← ENS-ENT	un ensemble est-il vide ?
	eqe	: BOOL ← ENS-ENT,ENS-ENT	prédicat d'égalité
variables	en,en'	: ENS-ENT	
	n,n',e,e'	: ENT	

équations

Relation entre les constructeurs

adj(adj(en,n),n')	= adj(adj(en,n'),n)
adj(adj(en,n),n)	= adj(en,n)

Définition des modificateurs

supe(evide,n)	= evide
supe(adj(en,n),n')	= si eq(n,n') alors supe(en,n') sinon adj(supe(en,n'),n)
union(en,evide)	= en
union(en,adj(en',n))	= si appe(en,n) alors union(en,en') sinon adj(union(en,en'),n)

Définition des observateurs

taillee(evide)	= zéro
taillee(adj(en,n))	= si appe(en,n) alors taillee(en) sinon suc(taillee(en))
appe(evide,n)	= faux
appe(adj(en,n),n')	= si eq(n,n') alors vrai sinon appe(en,n')
evide ?(evide)	= vrai
evide?(adj(en,n))	= faux
eqe (evide,en)	= evide?(en)
eqe(adj(en,n),en')	= si appe(en',n) alors eqe(supe(en,n),supe(en',n)) sinon faux

Représentation faible des suites d'entiers par des ensembles d'entiers

Dans un ensemble les éléments ne sont pas ordonnés et seule une occurrence des éléments est prise en compte. En conséquence seules pourront être représentées les opérations du type SUITE-ENT ne prenant en compte ni l'ordre ni les différentes occurrences d'un élément :

Les opérations sup de suppression d'un élément donné par son rang,
tête d'accès au dernier élément ajouté,
taille qui compte le nombre d'éléments (en considérant toutes les occurrences d'un élément)et
eq qui considère que deux suites sont égales si elles ont les mêmes éléments et le même ordre

ne pourront pas être représentées.

Ceci signifie que deux suites différentes telles que

s1 = ajout(ajout(ajout(ajout(svide,zéro),1),2),2)

s2 = ajout(ajout(ajout(svide,zéro),2),1)

seront représentées par des ensembles équivalents (modulo les opérations sur les ensembles) :

eqe(adj(adj(adj(adj(evide,zéro),1),2)2),
adj(adj(adj(evide,zéro),2),1))

Les suites s1 et s2 ont le même comportement vis à vis des observateurs représentés : app et svide ?

Définition du morphisme de signatures rse :

rse est l'identité sur l'environnement commun :

rse(SPECBOOL) = SPECBOOL

rse(SPECENT) = SPECENT

rse associe à la sorte SUITE-ENT, la sorte ENS-ENT

rse(SUITE-ENT) = ENS-ENT

rse associe aux opérations sur les suites, des opérations sur les ensembles de même nom, qui enrichissent SPECENS-ENT :

$\forall f \in \Sigma_{SUITE-ENT}$ tq profil(f) = T,T1...TN

rse(f) = $f \in \Sigma_{ENS-ENTE}$ tq profil(f) = rse(T),rse(T1)..rse(TN)

$$\Sigma_{ENS-ENT} = \Sigma_{ENS-ENT} \cup \left\{ \begin{array}{l} \text{svide : ENS-ENT} \leftarrow ; \\ \text{ajout : ENS-ENT} \leftarrow \text{ENS-ENT,ENT} ; \\ \text{conc : ENS-ENT} \leftarrow \text{ENS-ENT,ENS-ENT} ; \\ \text{supelt : ENS-ENT} \leftarrow \text{ENS-ENT,ENT} ; \\ \text{app : BOOL} \leftarrow \text{ENS-ENT,ENT} ; \\ \text{svide? : BOOL} \leftarrow \text{ENS-ENT} \end{array} \right\}$$

Les définitions de ces opérations enrichissent $E_{ENS-ENT}$:

$$E_{ENS-ENTE} = E_{ENS-ENT} \cup \left\{ \begin{array}{l} \text{svide} = \text{evide} ; \\ \text{ajout(en,n)} = \text{adj(en,n)} ; \\ \text{conc(en,en')} = \text{union(en,en')} ; \\ \text{supelt(en,n)} = \text{supe(en,n)} ; \\ \text{app(en,n)} = \text{appe(en,n)} ; \\ \text{svide?(en,n)} = \text{evide?(en,n)} \end{array} \right\}$$

rse définit bien une représentation faible : vérifions par exemple que le terme t suivant équivalent à vrai, est associé par rse à un terme encore équivalent à vrai.

t = app(conc(ajout(ajout(svide,2),1),ajout(svide,2)),2)

t est équivalent à vrai :

t = app(ajout(conc(ajout(ajout(svide,2),1),svide),2),2)
= vrai

rse(t) est aussi équivalent à vrai :

rse(t) = appe(union(adj(adj(evide,2),1),adj(evide,2)),2)
= appe(union(adj(adj(evide,2),1),evide),2)
car appe(adj(adj(evide,2),1),2)
= appe(adj(evide,2),2) = vrai
= appe(adj(adj(evide,2),1),2)
= appe(adj(evide,2),2)
= vrai

et vice-versa : tous les termes équivalents à rse(t) sont images par rse de termes équivalents à vrai.

Représentation faible des ensembles d'entiers par des suites d'entiers

Nous avons maintenant le processus inverse : deux ensembles équivalents pourront être représentés par des suites différentes modulo les équations des suites d'entiers. Cependant la représentation du prédicat d'égalité permettra d'identifier les suites images de deux ensembles équivalents.

Le morphisme de signatures res est défini sur SPECENS-ENT entier : toutes les opérations peuvent être représentées.

Définition du morphisme de signatures res :

res est l'identité sur l'environnement commun

res(SPECBOOL) = BOOL

res(SPECENT) = ENT

res associe à la sorte ENS-ENT la sorte SUITE-ENT

res(ENS-ENT) = SUITE-ENT

res associe aux opérations sur les ensembles des opérations sur les suites de même nom, qui enrichissent SPECSUITE-ENT :

$\forall f \in \Sigma_{ENS-ENT}$ tq profil(f) = T,T1...TN

res(f) = $f \in \Sigma_{SUITE-ENTE}$ tq profil(f) = res(T),res(T1)...res(TN)

$\Sigma_{SUITE-ENTE} = \Sigma_{SUITE-ENT} \cup \{$evide : SUITE-ENT ← ;
adj : SUITE-ENT ← SUITE-ENT,ENT ;
supe : SUITE-ENT ← SUITE-ENT,ENT ;
union : SUITE-ENT ← SUITE-ENT,SUITE-ENT ;
taillee : ENT ← SUITE-ENT ;
appe : BOOL ← SUITE-ENT,ENT ;
evide? : BOOL ← SUITE-ENT ;
eqe : BOOL ← SUITE-ENT,SUITE-ENT$\}$

Les définitions de ces opérations enrichissent $E_{SUITE-ENT}$:

$E_{SUITE-ENTE} = E_{SUITE-ENT} \cup$

$\{$ evide = svide ;
adj(sn,n) = ajout(sn,n) ;
supe(sn,n) = supelt(sn,n) ;
union(sn,sn') = conc(sn,sn') ;
taillee(svide) = zéro
taillee(ajout(sn,n)) = si app(sn,n) alors taillee(sn)
sinon suc(taillee(sn)) ;
appe(sn,n) = app(sn,n) ;
evide?(sn) = svide?(sn) ;
eqe(svide,sn) = svide?(sn) ;
eqe(ajout(sn,n),sn') = si app(sn',n)
alors eqe(supelt(sn,n),supelt(sn',n))
sinon faux $\}$

res définit bien une représentation faible : vérifions par exemple que le terme t suivant équivalent à vrai, est associé par res à un terme encore équivalent à vrai.

t = eqe(adj(adj(adj(evide,1),2),2),adj(adj(evide,2),1))
= vrai

res(t) = eqe(ajout(ajout(ajout(svide,1),2),2),ajout(ajout(svide,2),1))
= vrai

Bien que ajout(ajout(ajout(svide,1),2),2) ne soit pas équivalent à ajout(ajout(svide,2),1)

En particulier

taille(ajout(ajout(ajout(svide,1),2),2)) = 3
taille(ajout(ajout(svide,2),1)) = 2

Mais nous avons bien

taillee(ajout(ajout(ajout(svide,1),2),2))
= taillee(ajout,ajout(svide,1),2) = 2
taillee(ajout(ajout(svide,2),1)) = 2

Une représentation faible préserve les propriétés de consistance et de complétude suffisante puisque nous demandons que SPECTCE soit un enrichissement consistant et suffisamment complet vis à vis de SPEC.

Il est possible de composer deux représentationsfaibles :

Proposition : Soient T1,T2,T3 trois types spécifiés sur le même environnement SPEC = <S,Σ,E>, respectivement par

SPECT1 = SPEC+<S_{T1},Σ_{T1},E_{T1}>

SPECT2 = SPEC+<S_{T2},Σ_{T2},E_{T2}>

SPECT3 = SPEC+<S_{T3},Σ_{T3},E_{T3}>.

Soient r1 et r2 deux représentations faibles

r1 = SPECT1R → SPECT2E et r2 : SPECT2R → SPECT3E

où SPECT1R = SPEC+<S_{T1},Σ_{T1R},E_{T1R}> avec $\Sigma_{T1R} \subset \Sigma_{T1}$, $E_{T1} \vdash\!\!- E_{T1R}$

SPECT2E = SPECT2+<∅,Σ_{T2E},E_{T2E}>

SPECT2R = SPEC+<S_{T2},Σ_{T2R},E_{T2R}> avec $\Sigma_{T2R} \subset \Sigma_{T2}$, $E_{T2} \vdash\!\!- E_{T2R}$

SPECT3E = SPECT3+<∅,Σ_{T3E},E_{T3E}>

Si r2 peut être prolongé sur une restriction de SPECT2E par le morphisme de signatures

$\overline{r2}$: SPECT2ER → SPECT3EE

où SPECT2ER = SPECT2R+<∅,Σ_{T2ER},E_{T2ER}>

avec $\Sigma_{T2ER} \subset \Sigma_{T2E}$ et $E_{T2E} \vdash\!\!- E_{T2ER}$

SPECT3EE = SPECT3E+<∅,Σ_{T3EE},E_{T3EE}>

tel que $\forall t1,t2 \in T_{\Sigma U \Sigma_{T2R} U \Sigma_{T2ER},r}$ $s \in S$

$E \cup E_{T2R} \cup E_{T2ER} \vdash\!\!- t1 = t2 \Leftrightarrow E \cup E_{T3} U E_{T3E} U E_{T3EE} \vdash\!\!- \overline{r2}(t1) = \overline{r2}(t2)$

Alors le morphisme de signatures $\overline{r2} \circ r1$, composé de $\overline{r2}$ et de r1 est une représentation faible de T1 par T3.

Démonstration : La composée de deux morphismes de signatures étant un morphisme de signatures, il suffit de démontrer que

$$\forall t1,t2 \in T_{\Sigma U \Sigma_{T1RR},s} \quad s \in S$$

$$E \cup E_{T1RR} \vdash\!\!- t1 = t2 \Leftrightarrow EUE_{T3} U E_{T3E} \vdash\!\!- \overline{r2} \circ r1(t1) = \overline{r2} \circ r1(t2)$$

où $\Sigma_{T1RR} \subset \Sigma_{T1R}$ est tel que $\Sigma_{T2R} \cup \Sigma_{T2ER} \vdash\!\!- r1(\Sigma_{T1RR})$

$E_{T1R} \vdash\!\!- E_{T1RR}$

On sait que $\forall t1,t2 \in T_{\Sigma U \Sigma_{T1R},s}$ $s \in S$

$E \cup E_{T1R} \vdash\!\!- t1 = t2 \Leftrightarrow E \cup E_{T2} \cup E_{T2E} \vdash\!\!- r1(t1) = r1(t2)$

Or $T_{\Sigma U \Sigma_{T1RR},s} \subset T_{\Sigma U \Sigma_{T1R},s}, EUE_{T2R} U E_{T2ER} \vdash\!\!- r1(E_{T1RR}), E_{T2R} \subset E_{T2}$ et $E_{T2ER} \subset E_{T2E}$

donc $\forall t1,t2 \in T_{\Sigma U \Sigma_{T1RR},s}$ $s \in S$

$E \cup E_{T1RR} \vdash\!\!- t1 = t2 \Leftrightarrow EUE_{T2R} U E_{T2ER} \vdash\!\!- r1(t1) = r1(t2)$

On sait que $\forall t1,t2 \in T_{\Sigma U \Sigma_{T2R} U \Sigma_{T2ER},s}$ $s \in S$

$E \cup E_{T2R} E_{T2ER} \vdash\!\!- t1 = t2 \Leftrightarrow EUE_{T3} U E_{T3E} U E_{T3EE} \vdash\!\!- \overline{r2}(t1) = \overline{r2}(t2)$

Or $r1(T_{\Sigma U \Sigma_{T1RR},s}) \subset T_{\Sigma U \Sigma_{T2R} U \Sigma_{T2ER},s}$

donc $\forall r1(t1) r1(t2) \in r1(T_{\Sigma U \Sigma_{T1RR},s})$

$E \cup E_{T2R} U E_{T2ER} \vdash\!\!- r1(t1) = r1(t2) \Rightarrow EUE_{T3} U E_{T3E} U E_{T3EE} \vdash\!\!- \overline{r2}(r1(t1)) =$

$= \overline{r2}(r1(t2))$

donc $\forall t1,t2 \in T_{\Sigma \cup \Sigma_{T1RR},s}$ $\quad s \in S$

$$E \cup E_{T1RR} \vdash t1 = t2 \Leftrightarrow E \cup E_{T2R} \cup E_{T2ER} \vdash r1(t1) = r1(t2)$$

$$\Leftrightarrow E \cup E_{T3} \cup E_{T3E} \cup E_{T3EE} \vdash \overline{r2} \circ r1(t1) = \overline{r2} \circ r1(t2) \quad \square$$

3.2.- Représentations faibles paramétrées

Nous généralisons cette notion de représentation, aux spécifications paramétrées. Par instanciation du paramètre formel, on obtient une représentation. Les spécifications source et cible doivent avoir le même paramètre formel. Celui-ci n'ayant pas toujours une algèbre de termes non vide, la propriété demandée est exprimée sur les termes avec variables.

Soient TPS et TPC deux types abstraits paramétrés spécifiés par

$$SPECTPS = SPEC+SPECP+\langle\{TPS[S_p]\},\Sigma_{TPS},E_{TPS}\rangle \text{ et}$$

$$SPECTPC = SPEC+SPECP+\langle\{TPC[S_p]\},\Sigma_{TPC},E_{TPC}\rangle$$

où le paramètre formel $SPECP = SPEC+\langle S_p,\Sigma_p,E_p\rangle$

et l'environnement $SPEC = \langle S,\Sigma,P\rangle$ sont communs.

TPS et TPC sont persistants vis à vis de SPEC+SPECP.

Définition : Une représentation faible paramétrée de TPS par TPC sur SPEC+SPECP est un morphisme de signatures d'une restriction de SPECTPS vers un enrichissement de SPECTPC

$$r : SPECTPSR \to SPECTPCE$$

où $SPECTPSR = SPEC+SPECP+\langle S_{TPS},\Sigma_{TPSR},E_{TPSR}\rangle$

avec $\Sigma_{TPSR} \subset \Sigma_{TPS}$ et $\Sigma_{TPS} \vdash E_{TPSR}$

est une spécification paramétrée persistante

et $SPECTPCE = SPECTPC+\langle\emptyset,\Sigma_{TPCE},E_{TPCE}\rangle$

sont des spécifications paramétrées persistantes

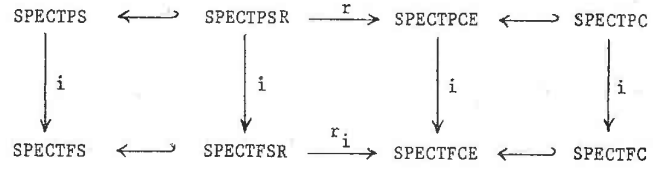
tel que $\forall t1,t2 \in T_{\Sigma \cup \Sigma_P \cup \Sigma_{TPSR},s}(X_s)$, $s \in S$

$$E \cup E_P \cup E_{TPSR} \vdash t1 = t2 \Leftrightarrow E \cup E_P \cup E_{TPC} \cup E_{TPCE} \vdash r(t1) = r(t2)$$

Les transformations que nous étudions dans la 3ème partie, sont des représentations faibles paramétrées.

Par instanciation des spécifications paramétrées sources et cibles par le même paramètre effectif présenté sur SPEC par $SPECF = \langle S_F,\Sigma_F,E_F\rangle$ (et le même morphisme de passage de paramètres

$i : SPEC+SPECP \to SPEC+SPECF$), on obtient une représentation r_i qui rend commutatif le diagramme ci-dessous :



r_i est défini par

$$r_i(SPEC+SPECF) = SPEC+SPECF$$

$r_i(S_{TFS}) = S_{TFC}$

$\forall f \in \Sigma_{TFSR}$

il existe $g \in \Sigma_{TPSR}$ tq $i(g) = f$

alors $r_i(f) = i(r(g))$

$r_i(f) \in \Sigma_{TFC} \cup \Sigma_{TFCE}$

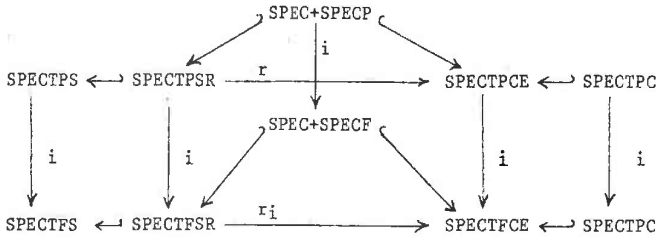
Proposition : Ainsi définie, r_i est une représentation faible :

$\forall t1,t2 \in T_{\Sigma \cup \Sigma_F \cup \Sigma_{TFSR},s}$ $s \in S \cup S_F$

$E \cup E_F \cup E_{TFSR} \vdash t1 = t2 \Leftrightarrow E \cup E_F \cup E_{TFC} \cup E_{TFCE}$

$\vdash r_i(t1) = r_i(t2)$

Démonstration : Nous avons le diagramme suivant :



⇒ Nous allons démontrer que $E \cup E_F \cup E_{TFC} \cup E_{TFCE} \vdash r_i(E_{TFSR})$

Soit $g = d \in E_{TFSE}$ alors il existe $g' = d' \in E_{TPSR}$

tel que $g = i(g')$ et $d = i(d')$

par définition de r on a $E_{TPC} \cup E_{TPCE} \vdash r(g') = r(d')$

et donc $E \cup E_F \cup E_{TFC} \cup E_{TFCE} \vdash i(r(g')) = i(r(d'))$

par définition de r_i on a $r_i(g) = i(r(g'))$ et $r_i(d) = i(r(d'))$

donc $E \cup E_F \cup E_{TFC} \cup E_{TFCE} \vdash r_i(g) = r_i(d)$.

⇐ Soient $t1,t2 \in T_{\Sigma \cup \Sigma_F \cup \Sigma_{TFSR},s}$ $s \in S \cup S_F$

$E \cup E_F \cup E_{FTC} \cup E_{TFCE} \vdash r_i(t1) = r_i(t2)$

$t1,t2 \in T_{\Sigma \cup \Sigma_F \cup \Sigma_{TFSR},s}$ $s \in S \cup S_F$

par complétude suffisante de SPECTFSR vis à vis de SPEC+SPECF

$\exists s1,s2 \in T_{\Sigma \cup \Sigma_F}$ tq $E \cup E_F \cup E_{TFSR} \vdash t1 = s1$ et $t2 = s2$

$r_i(t1), r_i(t2) \in T_{\Sigma \cup \Sigma_F \cup \Sigma_{TFC} \cup \Sigma_{TFCE},s}$ $s \in S \cup S_F$

par complétude suffisante de SPECTFCE vis à vis de SPEC+SPECF

$\exists s'1,s'2 \in T_{\Sigma \cup \Sigma_F}$ tq $E \cup E_F \cup E_{TFC} \cup E_{TFCE} \vdash r_i(t1) = s'1$ et $r_i(t2) = s'2$

r_i est l'identité sur SPEC+SPECF donc $r_i(s1) = s1$ et $r_i(s2) = s2$

Or $E \cup E_F \cup E_{TFSR} \vdash t1 = s1$ et $t2 = s2$

⇒ $E \cup E_F \cup E_{TFC} \cup E_{TFCE} \vdash r_i(t1) = r_i(s1)$ et $r_i(t2) = r_i(s2)$

Donc $E \cup E_F \cup E_{TFC} \cup E_{TFCE} \vdash s1 = s'1$ et $s2 = s'2$

par consistance de SPECFCE vis à vis de SPEC+SPECF

$E \cup E_F \vdash s1 = s'1$ et $s2 = s'2$

puisque $s1,s2,s'1,s'2 \in T_{\Sigma \cup \Sigma_F}$

Donc $E \cup E_F \cup E_{TFSR} \vdash t1 = s'1$ et $t2 = s'2$

Or par consistance de SPECTFCE vis à vis de SPEC+SPECF

$E \cup E_F \cup E_{TFC} \cup E_{TFCE} \vdash r_i(t1) = s'1,\ r_i(t2) = s'2$

$r_i(t1) = r_i(t2)$

$E \cup E_F \vdash s'1 = s'2$

Donc $E \cup E_F \cup E_{TFSR} \vdash t1 = t2$

CONSTRUCTION D'UNIVERS DE TYPES ABSTRAITS ALGEBRIQUES

CHAPITRE II : CONSTRUCTION D'UNIVERS DE TYPES ABSTRAITS ALGÉBRIQUES

Pour construire de manière incrémentale une spécification, nous suivons une démarche déductive et descendante. Un système est spécifié par ses différentes fonctions, celles-ci manipulant des données structurées. Nous obtenons ainsi une spécification dite fonctionnelle. Cependant celle-ci comporte de nombreuses redondances dues à l'approche descendante. D'autre part certains choix de structuration des données doivent être faits dès les premières étapes de la spécification, sans pouvoir être remis en question par la suite. Nous pallions à ces inconvénients en définissant des transformations. De manière à nous assurer de la conservation des propriétés de la spécification initiale, nous utilisons le cadre formel des types abstraits algébriques.

Nous présentons à l'aide d'un exemple, la construction d'une spécification. Cet exemple nous permet de montrer les avantages et les inconvénients de l'approche suivie. Puis nous montrons comment traduire la spécification fonctionnelle en termes de types abstraits algébriques. Ceux-ci sont présentés à l'aide du langage SPES-TYPES que nous définissons. Ils sont construits à l'aide de types de base et de constructeurs de types. Nous en donnons un ensemble au paragraphe 2. Au paragraphe 3 nous montrons comment construire une spécification en termes de types abstraits algébriques par étapes successives, en parallèle de la spécification fonctionnelle. Chaque étape définit un type soit à l'aide d'un constructeur de types et éventuellement d'un invariant, soit en complètant un type déjà défini de nouvelles opérations. Nous définissons puis étudions la relation "être issu de" reliant les différentes versions d'un type en cours de définition.

1.- DEFINITION DU LANGAGE SPES-TYPES

1.1.- Les problèmes liés à la spécification

Spécifier un problème, c'est le définir précisément, indépendamment de

tout élément relatif à sa résolution.

Une spécification doit ([FIN, 84])

- être compréhensible par le demandeur,
- être précise et non ambiguë,
- ne pas comporter d'erreurs (inconsistances) et
- définir complètement tous les aspects du problème (complétude).

Ce qui implique

- le choix de cadres formels pour décrire une spécification ([BUR, GOG, 77], [ABR, 78], [BAU, PEP, WOS, 78]),
- la nécessité de développer des méthodes, des langages et des outils d'aide à la spécification ([JAC, 83]),

ainsi que

- des méthodes de preuves de correction de spécifications.

La notion de types abstraits algébriques de données, basée sur des notions théoriques d'algèbre universelle ([MEL, 71]), bénéficie de travaux effectués dans d'autres domaines tels que l'étude de systèmes de réécriture ([REM, 82]) ou de la logique du premier ordre ([HUE, 77]). Ceux-ci ont permis de définir des méthodes et des systèmes de preuves de correction de spécifications ([MUS, 80b], [HUE, 81], [BAR, 83]). De nombreux langages de spécification ont vu le jour. Certains comme CLEAR ([BUR, GOG, 79]), ASL ([WIR, SAN, 82]), ACT ONE ([EHR, 83]), LARCH ([GUT, HOR, 83]) ne sont pas susceptibles de construire des types abstraits implémentables de manière systématique malgré la définition d'outils puissants de spécification.

D'autres, comme AFFIRM ([MUS, 80a]), OBJ ([GOG, TAR, 79]), SPEC+ ([KAP , 83]) permettent de définir des types abstraits et d'en générer des implémentations.

Le projet SPES encourage à construire une spécification en suivant une démarche déductive et descendante en s'appuyant sur le méta-algorithme développé par E. DUBOIS ([DUB, 84]). Celui-ci a été validé sur des exemples en vraie grandeur (spécification d'un système comptable du

suivi des dépenses informatiques et d'une application relative à la gestion d'un planning des forages, réalisées pour la Société Nationale Elf-Aquitaine). Nous montrons comment appliquer la méthode sur un exemple. Puis nous discutons les avantages et inconvénients de la démarche suivie. Nous proposons de pallier aux inconvénients en appliquant certaines des transformations à la spécification obtenue.

La spécification obtenue bien que décrivant les fonctionnalités du système, peut être exprimée en termes de types abstraits algébriques présentés en SPES-TYPES et définis à l'aide de constructeurs de types. C'est sous cette forme que nous pourrons appliquer les transformations.

1.1.1.- Exemple de spécification fonctionnelle

Le méta-algorithme de spécification permet de décrire en parallèle les fonctions devant être réalisées par le système et les différents types de données manipulés. L'idée est de définir d'abord la relation reliant les résultats attendus et les données du problème.

La définition de cette relation se fait après avoir précisé le type du résultat ainsi que celui des données. Si à ce niveau de décomposition on ne connaît pas toutes les données, il sera possible de compléter par la suite.

Une relation est définie à l'aide de relations intermédiaires qui devront être définies à leur tour, ainsi que d'opérations dont sont munis les types du résultat et des données. On réitère ce processus jusqu'à ce que toutes les relations intermédiaires introduites soient définies. Voici une formalisation de ce processus :

Spécification de l'opération ope : RES ← DON ...
Spécifop(ope) :
Définir RES
Définir DON (autant que possible)

Cas où • ope est une opération primitive
Alors ope(d) = f(d)
où f est une opération d'un constructeur de types
• ope est défini par une analyse par cas
Alors ope(d) = Si pre(d) Alors ope1(d)
Sinon ope2(d)

• ope est la composée d'une opération f et d'autres plus fines
Alors ope(d) = f(ope1(d),...opei(d),...open(d))
définir f

fin du cas.

Pour toutes les opérations opei introduites
Faire Spécifop(opei)

Compléter les arguments manquant dans le profil de ope

fin de Spécifop

Nous allons dérouler ce méta-algorithme sur l'exemple du loueur de bateaux, dérivé de M.A. Jackson ([JAC, 83]).

Enoncé : Un loueur de bateaux note les heures et les numéros de bateaux sur deux cahiers, l'un servant à noter les départs et l'autre les retours.

Nous voulons spécifier les fonctions départb et retourb permettant d'enregistrer les départs et les retours des bateaux au fur et à mesure qu'ils se succèdent, dans les cahiers. Un départ ne pourra être enregistré que si le bateau n'est pas déjà loué et réciproquement un retour ne pourra être enregistré que si le bateau était loué (ou parti).

Pour définir départb nous commençons par préciser
- le type de son résultat : le cahier des départs, notons-le CAHD
- les types de ses données, du moins de celles que nous connaissons : CAHD et le type des éléments du cahier noté LIGNE. Une ligne est composée d'une heure et d'un numéro de bateau.

Comme nous ne savons pas si nous aurons besoin de plus de données, nous laissons trois points "..." dans le profil de départb.

Profil de la fonction départb :

[départb : CAHD ← CAHD,LIGNE ...

Nous allons préciser la structure des objets des types introduits. Nous devons choisir parmi un ensemble de constructeurs de types ceux qui correspondent le plus à notre idée intuitive du modèle de l'univers. Il s'agit d'un choix parmi d'autres mais nous cherchons la souplesse pour pouvoir exprimer nos idées intuitives. Un type est défini alors comme instance d'un constructeur de types, ou type paramétré par des types qu'il nous faudra définir à leur tour.

[TYPE CAHD = SUITE[LIGNE]
[TYPE LIGNE = PC[numbat:NUMBAT,heure:HEURE]

SUITE et PC sont des constructeurs de types (prédéfinis) dont les objets auront respectivement une structure de suite et de produit cartésien.

Les types NUMBAT et HEURE sont des restrictions du type de base des entiers ENT. Cette restriction est exprimée à l'aide d'un invariant (nous pourrions l'exprimer aussi à l'aide d'un constructeur de types "intervalle").

[TYPE NUMBAT = ENT
[TYPE HEURE = ENT invariant : infeg(zéro,h) et infeg(h,23)

où
- h est une variable de type HEURE ce que nous exprimons par h : HEURE.
- infeg est une opération du type ENT
- zéro est une constante du type ENT
- 23 est la notation utilisée pour suc^{23}(zéro), terme sans variable de type ENT
- et est une opération du type BOOL, en notation infixée.

Nous pouvons maintenant définir départb à l'aide d'opérations apportées par les constructeurs de types SUITE et PC et d'opérations intermédiaires qu'il nous faudra définir. Enregistrer un départ c'est ajouter une ligne dans le cahier des départs mais uniquement si le bateau est présent (ou "rentré"). Si ce n'est pas le cas, on ne fait rien.

[départb(cdep,P,...)
[= si rentré(cdep,numbat(ℓ)...)
[alors ajout(cdep,ℓ) sinon cdep
[cdep : CAHD
[ℓ : LIGNE

rentré est un prédicat prenant la valeur vrai si le bateau concerné est de retour après son dernier départ, ou s'il n'est pas parti du tout.

numbat est l'accés à une composante du produit cartésien LIGNE réalisé grâce au nom de cette composante.

ajout est l'opération d'adjonction en tête de suite.

Itérons le processus en définissant le prédicat rentré introduit. D'abord son profil :

```
⌈rentré : BOOL ← NUMBAT ...
```

Le type NUMBAT est déjà défini.

Un bateau est rentré s'il est de retour après son dernier départ ou s'il n'est pas parti du tout :

```
⌈rentré(b...)
     = pas parti(cdep,b) ou pas reparti(cdep,cret,b)
⌊b : NUMBAT
 cret : CAHR
 TYPE CAHR = SUITE[LIGNE] type du cahier des retours
```

pas parti est un prédicat prenant la valeur vrai si le bateau concerné n'est pas enregistré dans le cahier des départs.

par reparti est un prédicat prenant la valeur vrai si le bateau concerné dont un retour a été enregistré (dans le cahier cret des retours) n'a pas de départ postérieur enregistré (dans le cahier cdep des départs).

Il faut maintenant définir les deux opérations introduites pas parti et par reparti.

```
⌈pas parti : BOOL ← CAHD,NUMBAT
⌊pas parti (cdep,b)
     = eq(indef,derdep(cdep,numbat(b))
```

derdep est l'accès (par recherche associative) au dernier départ enregistré (dans le cahier des départs), d'un bateau donné (par son numéro).

numbat est l'accès au numéro de bateau contenu dans une ligne. C'est-à-dire l'accès à la composante nommée "numbat" d'un objet de type produit cartésien LIGNE.

```
⌈pas reparti : BOOL ← CAHD,CAHR,NUMBAT
```



```
⌈par reparti(cdep,cret,b)
⌊    = reldepret(derdep(cdep,numbat(b)),derret(cret,numbat(b)))
```

derret est l'accès au dernier retour enregistré d'un bateau donné.

reldepret est un prédicat indiquant si deux lignes données peuvent correspondre à une même location. C'est-à-dire s'il s'agit du même bateau et si l'heure de la première ligne, correspondant à un départ est inférieure ou égale à l'heure de la deuxième ligne, correspondant à un retour.

Nous avons introduit les opérations intermédiaires derdep, derret et reldepret.

```
⌈derdep : LIGNE ← CAHD,NUMBAT
 derdep(cdep,b)
      = tête(iter_{id,eq(numbat(.),b)}(cdep))
⌊b : NUMBAT
```

```
⌈derret : LIGNE ← CAHR,NUMBAT
 derret(cret,b)
⌊    = tête(iter_{id,eq(numbat(.),b)}(cret))
```

```
⌈reldepret(ℓ,ℓ')
 |    = eq(numbat(ℓ),numbat(ℓ')) et
⌊      infeg(heure(ℓ),heure(ℓ'))
```

iter est l'opérateur d'itération appliquant une opération (ici l'identité) à tous les éléments d'une suite vérifiant une propriété (ici eq(numbat(.),b))

tête est l'accès au dernier élément ajouté à une suite.

heure est la projection sur le champ heure du type produit cartésien LIGNE

ℓ,ℓ' : LIGNE

Nous avons défini toutes les opérations et tous les types introduits. Nous avons précisé tous les arguments des opérations. Nous pouvons à présent compléter ce qui les concerne.

```
rentré : BOOL ← CAHD,CAHR,NUMBAT
rentré(cdep,cret,b)
     = pas parti(cdep,b) ou pas reparti(cdep,cret,b)
```

```
départb : BOOL ← CAHD,CAHR,LIGNE
départb(cdep,cret,ℓ)
     = si rentré(cdep,cret,numbat(ℓ)) alors ajout(cdep,ℓ)
                                      sinon cdep
```

Nous résumons cette spécification à l'aide des blocs suivants. Chaque bloc correspond à un "niveau" de descente dans la spécification . Nous ne répétons pas les définitions (formelles ou informelles) données à un niveau supérieur. Nous ajoutons au premier niveau l'opération retourb permettant d'enregistrer le retour d'un bateau dans le cahier des retours. La spécification de cette opération est semblable à celle de départb.

Structure des blocs de spécification :

liste des noms d'opérations et leurs profils	
liste des définitions formelles des opérations	
définition des types	déclarations de variables
lexique : définitions informelles des opérations et des types introduits	

Spécification des opérations départb et retourb de l'exemple du loueur de bateaux

Niveau 1 :

opérations définies :

départb : CAHD ← CAHD,CAHR, LIGNE

retourb : CAHR ← CAHD,CAHR,LIGNE

définitions :

```
départb(cdep,cret,ℓ)
   = si rentré(cdep,cret,numbat(ℓ))
        alors ajout(cdep.ℓ)
        sinon cdep
retourb(cdep,cret,ℓ)
   = si rentré(cdep,cret,numbat(ℓ))
     alors cret
     sinon ajout(cret,ℓ)
```

Types

```
CAHD = SUITE[LIGNE]
CAHD = SUITE[LIGNE]
LIGNE = PC[numbat:NUMBAT,heure:HEURE]
NUMBAT = ENT
HEURE = ENT invariant:infeg(zéro,h)
                  et    infeg(h,23)
```

Variables

```
cdep : CAHD
cret : CAHR
ℓ : LIGNE
h : HEURE
```

Lexique

départb : enregistrement dans le cahier des départs, du départ d'un bateau

retourb : enregistrement dans le cahier des retours, du retour d'un bateau

rentré : prédicat indiquant si un bateau n'est pas parti du tout ou s'il est rentré après son dernier départ

CAHD est le type du cahier des départs

CAHR est le type du cahier des retours

LIGNE est le type des éléments composant les cahiers

NUMBAT est le type des numéros de bateaux

HEURE est le type des heures de départ ou de retour des bateaux.

Niveau 2 :

opération définie

rentré : BOOL ← CAHD,CAHR,NUMBAT

Définition

rentré(cdep,cret,b)

= pas parti(cdep,b)

ou par reparti(cdep,ret,b)

Lexique

pas parti : prédicat indiquant qu'un bateau n'est pas enregistré dans le cahier des départs.

par reparti : prédicat indiquant qu'un bateau dont un retour a été enregistré, n'a pas de départ postérieur enregistré.

Niveau 3 :

opérations définies

pas parti : BOOL ← CAHD,NUMBAT

pas reparti : BOOL ← CAHD,CAHR,NUMBAT

Définitions

pas parti(cdep,b)

= eq(indef,derdep(cdep,b))

pas reparti(cdep,cret,b)

= reldepret(derdep(cdep,b),derret(cret,b))

Lexique

derdep : accès au dernier départ enregistré d'un bateau

derret : accès au dernier retour enregistré d'un bateau

reldepret : prédicat indiquant si deux lignes peuvent correspondre à une même location : même numéro de bateau et heure de la première ligne inférieure ou égale à l'heure de la deuxième ligne.

Niveau 4 :

opérations définies

derdep : LIGNE ← CAHD,NUMBAT

derret : LIGNE ← CAHR,NUMBAT

reldepret : BOOL ← LIGNE,LIGNE

Définitions

derdep(cdep,b)

= $tête(iter_{id,eq(numbat(.),b)}(cdep))$

derret(cret,b)

= $tête(iter_{id,eq(numbat(.),b)}(cret))$

reldepret(ℓ,ℓ')

= eq(numbat(ℓ),numbat(ℓ')) et infeg(heure(ℓ),heure(ℓ'))

1.1.2.- Discussion de la spécification

L'approche descendante et l'utilisation de constructeurs de types présentent à la fois des avantages et des inconvénients.

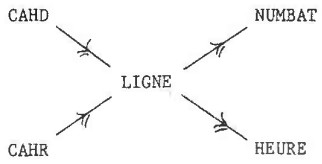
Les avantages :

- L'approche descendante favorise l'obtention d'une spécification complète.
- La spécification en termes de fonctions peut paraître plus naturelle aux utilisateurs. D'autre part, le fait d'associer à chaque définition formelle, une définition informelle favorise la communication entre utilisateurs et spécifieurs.
- Il existe une hiérarchie des fonctions, caractéristique de l'approche descendante, correspondant aux niveaux successifs de raffinements. Ainsi un niveau i de la spécification utilise les opérations définies aux niveaux j strictement plus grands que i (j > i).
  De la même manière les structures de données sont définis hiérarchiquement. Chaque type est défini en fonction de types "plus simples". Par exemple le type du cahier des départs, CAHD, est défini en fonction des éléments du cahier, les lignes de type LIGNE.

Il est possible de définir un ordre (voir § II, 3.1.1.) sur les types définis d'une spécification (nous n'étudions pas les types définis récursivement). En notant "$\succ$" cet ordre nous avons :

CAHD $\succ$ LIGNE LIGNE $\succ$ NUMBAT
CAHR $\succ$ LIGNE LIGNE $\succ$ HEURE

Ce qui peut se traduire par le graphe suivant :



- Les fonctions sont définies comme composées de fonctions intermédiaires (telles que rentré, pas parti, par reparti...) et d'opérations de base associées aux structures de données (telles que ajout ou numbat et heure, les accès aux composantes des produits cartésiens).
  Les structures de données utilisées sont des instances de types paramétrées (SUITE et PC) et de types de base (ENT,BOOL).
  Ainsi la spécification bénéficie-t-elle du cadre formel des types abstraits de données. Il lui assure la correction de ses structures de données et de ses opérations, sans en avoir l'inconvénient majeur qu'est la difficulté d'utilisation pour un informaticien non mathématicien.

Les inconvénients :

- Dès les premiers niveaux de décomposition, le spécifieur est conduit à choisir une formalisation de l'univers à l'aide des constructeurs de types. Les choix ainsi faits auront une répercussion sur les niveaux de décomposition ultérieurs. De fait les fonctions sont définies à l'aide des opérations dont sont munis les types de données.

- le méta-algorithme permet de définir en parallèle une fonction et les données manipulées par cette fonction. Celles-ci sont définies pour répondre le mieux possible aux exigences de la fonction. Aussi il n'y a pas de recherche systématique d'autres structures préalablement définies dans le système. Ce qui entraîne des redondances
  - soit au niveau des fonctions définies
  - soit au niveau des structures de données.
- enfin, des structures de données précises, hiérarchiques, peuvent être une contrainte à l'evolution du système, si l'on veut lui ajouter de nouvelles fonctionnalités.

Pour résoudre les problèmes précédents, nous proposons que cette première étape de construction de la spécification soit suivie d'une deuxième étape de transformations.

Celle-ci doit permettre grâce à une analyse globale des structures de données, de mettre à jour leurs défauts, et grâce à des transformations
- d'optimiser certaines fonctions définies sur des structures de données choisies trop précocement,
- d'éliminer la redondance introduite par une analyse descendante et
- d'adapter les structures de données à de nouvelles contraintes liées à de nouvelles fonctionnalités.

1.1.3.- Proposition d'une solution

Partant de la spécification fonctionnelle, nous allons l'exprimer en termes de types abstraits de données. Cela se fait en associant chaque fonction à un type défini. Nous définissons un langage de présentation de types de données adapté : SPES-TYPES, nous permettant d'avoir les deux vues fonctionnelles et structures de données en même temps. Le cadre formel des types abstraits nous facilite la définition
- de relations permettant de comparer et donc d'analyser les types d'une spécification, ainsi que la définition
- de transformations comme représentations de types abstraits.

Nous présentons dans la troisième partie des transformations permettant
- d'optimiser les opérations définies d'un type
- d'éliminer la redondance
- de simplifier des structures de données
- d'augmenter l'évolutivité de structures de données.

1.2.- Définition de SPES-TYPES

SPES-TYPES possède les caractéristiques suivantes :
- Il est muni d'un ensemble de types de base et de constructeurs de types qui permettront de décrire les structures de données rencontrées.
- Les nouvelles opérations sont définies soit inductivement, soit par composition en utilisant les opérations des constructeurs de types.
- Il est possible de restreindre l'ensemble des objets d'un type à l'aide d'un invariant ainsi que nous l'avons expliqué dans la première partie (§ I, 1.3.3).
- Nous donnerons (§ II, 3.1.2) un critère permettant d'associer chaque opération définie à un type.

Un type défini en SPES-TYPES est un type de base ou une instance d'un type paramétré. Il est possible de restreindre l'ensemble de ses objets à l'aide d'un invariant. Il peut être enrichi par des opérations définies à l'aide d'opérations apportées par le type paramétré ou le type de base.

Remarquons qu'en SPES-TYPES nous spécifions des types et non des spécifications comportant plusieurs types d'intérêt.

Cette hiérarchie très stricte peut paraître très contraignante.

Cependant il n'en est rien. En effet lorsque nous avons besoin de spécifier en même temps les types T1...TN, nous définissons le type produit cartésien de T1...TN. Nous savons alors associer toute opération à un et un seul type.

Le langage SPES-TYPES peut être défini dans des langages de spécification de types abstraits algébriques à condition que ceux-ci autorisent l'utilisation de types paramétrés.

Nous n'utilisons que des équations (et les opérations si-alors-sinon).

Les opérateurs sont surchargés. Certaines opérations sont cachées. Cependant, SPES-TYPES comportant comme constructeurs de types les structures fournies par les langages de programmation, il permet de spécifier tout type calculable.

Après avoir présenté deux exemples de présentations de types, nous définissons le langage SPES-TYPES : d'abord en étudiant les composantes d'une présentation puis en en donnant la grammaire.

1.2.1.- Exemples de présentations de types en SPES-TYPES

Nous présentons les types NOMBRE des nombres et NOTEL des numéros de téléphone.

TYPE NOMBRE = SUITE[CHIFFRE]	
opérations utilisables importées : svide, conc, tête, taille, éq définies : (c) ajoutqueue	déclaration de variables nb : NOMBRE n : CHIFFRE
Définition (c) ajoutqueue : NOMBRE ← NOMBRE,CHIFFRE ajoutqueue(nb,n) = ajoutrg(nb,taille(nb),n)	
Lexique : TYPE NOMBRE : type des nombres formés de suites de chiffres svide : suite vide conc : concaténation de deux suites tête : premier élément d'une suite taille : nombre d'éléments dans une suite eq : égalité entre deux suites ajoutqueue : nouveau constructeur - adjonction en queue (ajoutrg est l'équation d'adjonction d'un élément à un rang donné d'une suite, c'est une opération du constructeur SUITE, importée cachée)	

TYPE CHIFFRE = ENT	
invariant : infeg(zéro,n) et infeg(n,9)	
opérations utilisables : importées : zéro,suc,infeg,eq	déclaration de variables : n : CHIFFRE
Lexique : TYPE CHIFFRE : type des entiers de zéro à neuf.	

TYPE NOTEL = NOMBRE	
invariant : eq(taille(notel),8)	
opérations utilisables importées : svide,conc taille,eq définies : (c) ajoutqueue,ind	déclaration de variable : notel : NOTEL
Définition : ind : CHIFFRE ← NOTEL ind(notel)= tête(notel)	
Lexique : TYPE NOTEL : type des nombres appelés numéros de téléphone ind : indicatif régional d'un numéro de téléphone	

1.2.2.- Constituants d'une présentation de type

Un type T présenté en SPES-TYPES se compose de trois parties
- une expression de type notée expr(T)
- un invariant noté invar(T)
- la présentation d'un ensemble d'opérations noté oper(T).

L'expression d'un type est formée
- soit d'un nom d'un type (défini ou de base)
  exemple : le type NOTEL a comme expression le type défini NOMBRE
  le type CHIFFRE a comme expression le type de base ENT
- soit d'un constructeur de types instancié par des expressions de types.
  exemple : le type NOMBRE a comme expression le constructeur SUITE instancié par l'expression formée d'un type défini CHIFFRE.

Elle définit la structure des objets du type, et leurs propriétés. Elle apporte un ensemble d'opérations sur ces objets. Dans la spécification fonctionnelle, c'est elle qui définissait les types : le type du cahier des départs était défini par

TYPE CAHD = SUITE[LIGNE].

L'expression d'un type permet d'avoir une représentation intuitive de la structure de ses objets (notion de modèle de l'univers). Il faut cependant rappeler qu'il s'agit d'une structure abstraite qui pourra être implantée par la suite de diverses manières. D'autre part, la structure choisie détermine un ensemble de propriétés des objets. Si le type désiré ne correspond pas à ces propriétés, il faudra compenser l'inadéquation de la structure par des définitions d'opérations spécifiques plus compliquées, conduisant à des implantations moins efficaces. On pourra alors transformer cette structure en une autre plus adéquate. Dans la troisième partie de ce travail nous montrons comment se fait cette transformation.

L'invariant d'un type est une propriété que doivent vérifier les objets du type. Il opère une restriction sur l'ensemble des objets engendrés par les constructeurs d'objets. (Voir le paragraphe I.1.3.3). Cette notion d'invariant est très répandue : dans les bases de données on parle de contraintes d'intégrité.

Exemple : Le type NOTEL est muni de l'invariant infeg(taille(notel),8) restreignant l'ensemble des numéros de téléphone aux nombres de taille égale à 8.

L'invariant d'un type doit toujours être plus restrictif que celui du type de son expression : si T' est le type nommé dans l'expression de T, alors invar(T) impl invar(T').

Un type dont on ne précise pas l'invariant, possède celui du type de son expression.

Un type dont l'invariant est égal à "faux" décrit un ensemble vide d'objets. On l'appelle le type VIDE.

La partie opérations de la spécification d'un type T précise l'ensemble de ses opérations. Celui-ci est composé d'opérations importées et d'opérations définies.

Les opérations importées sont apportées par le constructeur de types définissant la structure des objets du type T. Ce sont celles qui sont utilisées (ou utilisables) dans la définition soit d'une opération du type T (opérations importées cachées) soit d'un autre type comme opération du

type T (opérations importées utilisables). Remarquons que les opérations des types paramétres du constructeur de types, appartiennent à l'environnement du type T présenté.

Exemple : les opérations du type NOMBRE importées du constructeur SUITE et utilisables sont svide, conc, tête, taille et eq. L'opération ajoutrg est importée cachée car elle apparaît dans la définition de l'opération ajoutqueue.

Les opérations définies sont les opérations définies du type de l'expression (on dit qu'un type T "hérite" des opérations définies du type de son expression) et celles définies dans la présentation du type T.

Exemple : les opérations définies du type NOTEL sont ajoutqueue, héritée du type NOMBRE, et ind.

Une opération définie peut être cachée. Dans ce cas, elle n'est pas exportable et elle sert exclusivement à la définition d'autres opérations.

L'ensemble des opérations du type T noté oper(T) est présenté par

- la liste des opérations utilisables oputil(T), formée
  . des opérations importées utilisables par un autre type opimputil(T), et
  . des opérations définies (non cachées) opdefutil(T), on a donc

$$oputil(T) = opimputil(T) \cup opdefutil(T).$$

- les définitions des opérations définies (utilisables et cachées) opdef(T). Les opérations définies cachées sont introduites dans ces définitions et définies à leur tour.
- les déclarations des variables utilisées dans la définition des opérations et dans l'invariant.
  Les déclarations de variables ont une portée locale. Il est indispensable de déclarer les variables à cause de la surcharge des opérateurs (un même nom d'opération peut désigner des opérations différentes).

La définition d'une opération est composée d'un profil et d'un ensemble d'équations.

Nous considérons que les opérations d'un type de base c ou d'un constructeur de types c sont toutes exportables : opimputil(c) = oper(c) et que l'ensemble des opérations définies est vide : opdef(c) = ∅. Dans ces conditions nous définissons les ensembles des opérations importées et définies, utilisables d'un type, par rapport à ceux du type de son expression :

Soit un type T présenté en SPES-TYPES. Notons EXPRT le type de son expression. Alors

- l'ensemble des opérations importées utilisables de T est une restriction de l'ensemble des opérations importées utilisables de EXPRT :

$$opimputil(T) \subseteq opimputil(EXPRT)$$

- le type T contient outre les opérations définies dans sa présentation, l'ensemble des opérations définies utilisables de EXPRT :

$$opdefutil(T) \supseteq opdefutil(EXPRT)$$

  Si EXPRT est un constructeur de types instancié, T n'hérite d'aucune opération définie utilisable.

Un type dont on ne précise pas d'opérations

. importées, possède celles du type de son expression

$$opimputil(T) = opimputil(EXPRT)$$

. définies, possède celles du type de son expression

$$opdefutil(T) = opdefutil(EXPRT)$$

L'ensemble des opérations utilisables de T est l'union des opérations importées utilisables et des opérations définies utilisables :

$$oputil(T) = opimputil(T) \cup opdefutil(T)$$

L'ensemble des opérations de T oper(T) contient en outre les opérations cachées (importées ou définies).

Pour faciliter les définitions inductives d'opérations, il est nécessaire de connaître pour tout type, l'ensemble des opérations (utilisables) engendrant les objets du type, c'est-à-dire les constructeurs d'objets. Ce sont

soit les constructeurs d'objets du type de l'expression s'ils sont utilisables, soit des opérations définies. Dans ce cas, nous faisons précéder le nom du nouveau constructeur d'objets d'un "(c)".

Exemple : l'opération ajoutqueue est constructeur utilisable des objets des types NOMBRE et NOTEL.

opdefutil(T)	$\supseteq$	opdefutil(EXPRT)
opimputil(T)	$\subseteq$	opimputil(EXPRT)
invar(T)	impl	invar(EXPRT)

Dans les trois cas, il y a égalité si ce n'est pas précisé dans la définition de T.

Un type défini uniquement par une expression est donc égal au type de son expression :

T est défini uniquement par une expression formée			
	d'un type défini EXPRT	d'un type de base B	d'un constructeur instancié C[T1...TN]
opdefutil(T) =	opdefutil(EXPRT)	oputil(B)	oputil(C)
opimputil(T) =	opimputil(EXPRT)	$\emptyset$	$\emptyset$
invar(T) =	invar(EXPRT)	vrai	vrai

Le type VIDE décrit un ensemble vide d'objets. Il a un invariant égal à faux : invar(VIDE) = faux.

Tout type d'invariant égal à faux est équivalent au type VIDE.

Nous présentons un type par un bloc de la forme suivante :

Nom du TYPE = Expression	
invariant	
liste des opérations utilisables importées définies	déclaration de variables
Lexique	
Définition des opérations définies	

Analysons les présentations des types CHIFFRE, NOMBRE et NOTEL :

Type NOMBRE

expr(NOMBRE) = SUITE[CHIFFRE]

oper(NOMBRE) = {svide,conc,tête,taille,eq,ajoutqueue,ajoutrg}

oputil(NOMBRE) = ope(NOMBRE)\{ajoutrg}

ajoutrg est une opération importée cachée

opimputil(NOMBRE) = {svide,conc,tête,taille,eq}

opdefutil(NOMBRE) = opdef(NOMBRE) = {ajoutqueue}

il n'y pas d'opération définie cachée

constructeurs des objets du type NOMBRE : svide et ajoutqueue

Type NOTEL

expr(NOTEL) = NOMBRE

invar(NOTEL) = eq(taille(notel),8)

on a bien invar(NOTEL) impl invar(CHIFFRE)

oper(NOTEL) = oper(NOMBRE) $\cup$ {ind}

oputil(NOTEL) = {svide,conc,taille,eq,ajoutqueue,ind}

opimputil(NOTEL) = {svide,conc,taille,eq} = opimputil(CHIFFRE)\{tête}

tête est une opération importée cachée, comme ajoutrg

opdefutil(NOTEL) = {ajoutqueue,ind}

il n'y a pas d'opération définie cachée

constructeurs des objets du type NOTEL : svide et ajoutqueue

on a bien opimputil(NOTEL) $\subseteq$ opimputil(NOMBRE)

et opdefutil(NOTEL) $\supseteq$ opdefutil(NOMBRE).

Type CHIFFRE

expr(CHIFFRE) = ENT

invar(CHIFFRE) = infeg(zéro,n) et infeg(n,9)

oper(CHIFFRE) = {zéro,suc,infeg,eq}
= oputil(CHIFFRE)
= opimputil(CHIFFRE)

opdefutil(CHIFFRE) = ∅

constructeurs des objets du type CHIFFRE : ceux du type ENT : zéro et suc.

1.2.3.- Grammaire de SPES-TYPES

Nous donnons la grammaire de la présentation d'un type en SPES-TYPE. Nous précisons cependant le fait qu'il s'agit d'une syntaxe abstraite. En particulier nous ne précisons pas la syntaxe d'un terme, d'un terme à valeur booléenne (terme-bool), d'un lexique. Nous utilisons par convention des identificateurs minuscules pour les noms de variables (nom-var), d'opérations (nom-op),de champ (nom-chp), et des identificateurs majuscules pour les noms de types (NOM-TYPE) et de constructeurs de types (NOM-CONST-TYPES).

Dans la grammaire nous notons :

< - >	un non terminal
a,b	pour a et b
a/b	pour a ou b
a==b*	pour []/b/b,a
a==b+	pour b/b,a

TYPE == NOM-TYPE,<expression>,<invariant>,<ensemble d'opérations>

<expression> == NOM-CONST-TYPES,<paramètres>/NOM-TYPE

<paramètres> == <expression>+/(nom-chp,<expression>)+

<invariant> == <terme-bool>

<ensemble d'opérations> == <op-utilisables>*,<def-op>*,<def-var>*

<op-utilisables> == nom-op

<def-op> == nom-op,<profil>,<lexique>,<équations>+

<profil> == NOM-TYPE, (NOM-TYPE)*

<équation> == <terme>,<terme>

<decl-var> == nom-var,NOM-TYPE

2.- TYPES DE BASE ET CONSTRUCTEURS DE TYPES

Nous avons vu l'importance des types de base et des constructeurs de types dans les définitions de types en SPES-TYPES. Ils doivent être munis d'un grand nombre d'opérations de manière à faciliter la définition des fonctions attendues d'un système. Ils doivent être consistants et suffisamment complets pour les types de base, et persistants pour les constructeurs de types. Nous allons définir un petit ensemble de types de base et de constructeurs de types comportant chacun un ensemble d'opérations permettant d'engendrer par composition un grand nombre d'opérations. Les constructeurs de types que nous avons choisi forment un ensemble minimum. Ils facilitent des vues diverses et aident l'intuition du specifieur. Cependant nous verrons dans la troisième partie que le choix d'un constructeur n'est pas définitif et qu'il est possible d'en changer par transformation.

2.1.- Types de base

Nous allons définir deux types de base :

- le type BOOL des booléens
- le type ENT (ou ENTIER) des entiers.

Une extension possible pourrait être constituée des types de base suivants :

- le type ENTREL des entiers relatifs
- le type CAR des caractères
- le type CHCAR des chaînes de caractères
- le type ENTDEC des nombres décimaux
- les types HEURE et DATE très utilisés en informatique de gestion.

Les deux types BOOL et ENT étant présentés gracieusement et sans relations entre les constructeurs, ils sont consistants et suffisamment complets.

2.1.1.- Le type BOOL des booléens

TYPE BOOL

opérations : constructeurs vrai : BOOL ←
faux : BOOL ←

modificateurs non : BOOL ← BOOL
et : BOOL ← BOOL,BOOL
ou : BOOL ← BOOL,BOOL
eq : BOOL ← BOOL,BOOL
impl : BOOL ← BOOL,BOOL
si-alors-sinon : BOOL ← BOOL,BOOL,BOOL

équations : def-modif

non(vrai)	= faux
non(faux)	= vrai
et(vrai,b)	= b
et(faux,b)	= faux
ou(vrai,b)	= vrai
ou(faux,b)	= b
eq(vrai,b)	= b
eq(faux,b)	= non(b)
impl(b,b')	= ou(non(b),b')
si-alors-sinon(vrai,b,b')	= b
si-alors-sinon(faux,b,b')	= b'

variables : b,b' : BOOL

notations :

les opérations et, ou et impl seront notées sous la forme

infixée : b et b', b ou b', b impl b'

l'opération si-alors-sinon sera notée sous la forme

dist-fixée : si b alors b' sinon b"

2.1.2.- Le type ENT des entiers naturels

TYPE ENT des entiers positifs ou nuls

opérations : constructeurs :

		notations usuelles
zéro	: ENT ←	0
suc	: ENT ← ENT	+1
indef	: ENT ←	ω

modificateurs :

plus	: ENT ← ENT,ENT	+
moins	: ENT ← ENT,ENT	-
max	: ENT ← ENT,ENT	
si-alors-sinon	: ENT ← BOOL,ENT,ENT	

observateurs :

nul	: BOOL ← ENT	= 0?
infeg	: BOOL ← ENT,ENT	≤
eq	: BOOL ← ENT,ENT	=

cachées : sucfn : ENT ← ENT

équations :

définition du constructeur suc à l'aide du constructeur caché sucfn

suc(zéro)	= sucfn(zéro)
suc(sucfn(e))	= sucfn(sucfn(e))
suc(indef)	= indef

définition des modificateurs

plus(zéro,e)	= e
plus(sucfn(e),e')	= suc(plus(e,e'))
plus(indef,e)	= indef
moins(zéro,zéro)	= zéro
moins(zéro,sucfn(e))	= indef
moins(zéro,indef)	= indef
moins(sucfn(e),zéro)	= sucfn(e)
moins(sucfn(e),sucfn(e'))	= moins(e,e')
moins(sucfn(e),indef)	= indef
moins(indef,e)	= indef
max(zéro,zéro)	= zéro
max(zéro,sucfn(e))	= sucfn(e)
max(sucfn(e),zéro)	= sucfn(e)

```
max(sucfn(e),sucfn(e'))     = sucfn(max(e,e'))
max(indef,e)                = indef
max(e,indef)                = indef

si-alors-sinon(vrai,e,e')   = e
si-alors-sinon(faux,e,e')   = e'
```

définition des observateurs

```
nul(zéro)                   = vrai
nul(sucfn(e))               = faux
nul(indef)                  = faux

infeg(zéro,zéro)            = vrai
infeg(zéro,sucfn(e))        = vrai
infeg(sucfn(e),zéro)        = faux
infeg(sucfn(e),sucfn(e'))   = infeg(e,e')
infeg(indef,e)              = eq(indef,e)
infeg(e,indef)              = eq(e,indef)

eq(zéro,zéro)               = vrai
eq(zéro,sucfn(e))           = faux
eq(zéro,indef)              = faux
eq(sucfn(e),zéro)           = faux
eq(sucfn(e),sucfn(e'))      = eq(e,e')
eq(sucfn(e),indef)          = faux
eq(indef,zéro)              = faux
eq(indef,sucfn(e))          = faux
eq(indef,indef)             = vrai
```

variables : e,e' : ENT

notations :

un = suc(zéro)

$n = suc^n(zéro)$

l'opération si-alors-sinon sera notée sous la forme dist-fixée

si b alors e sinon e'

2.2.- Constructeurs de types

Pour exposer notre démarche et ne pas surcharger cette thèse nous nous limitons à la définition de quatre constructeurs de types :

- le constructeur PC (ou PRODCART) de produits cartésiens.
- le constructeur S (ou SUITE) de suites.
- le constructeur E (ou ENS) d'ensembles.
- le constructeur T (ou TABLE) de tables.

Une extension possible pourrait être constituée des constructeurs de types suivants :

- le constructeur SSR de suites ne pouvant pas comporter plusieurs occurrences d'un même élément (suite sans répétitions),
- le constructeur MENS de multi-ensembles,
- le constructeur PCCLE de produits cartésiens comportant une composante identifiante appelée clé,
- le constructeur UNION permettant de réaliser des unions disjointes de types quelconques.

Cet ensemble de constructeurs de types correspond à celui utilisé par E. Dubois [DUB, 84] pour spécifier un système comptable du suivi des dépenses informatiques et une application relative à la gestion d'un planning des forages pour la Société Nationale Elf Aquitaine.

Ceci nous autorise à dire qu'il permet de définir pratiquement toutes les structures d'objets rencontrées en analyse de systèmes d'information. Les fichiers sont des suites (avec ou sans répétitions d'éléments) ou des ensembles (ou multi-ensembles) de produits cartésiens (caractérisés ou non par une clé). Les tables permettent d'associer à une clé (ou indice) un élément (accessible directement à l'aide de cette clé). Quand une fonction rend comme résultat des objets de type différents (opérateurs multicibles), le constructeur UNION sera utilisé pour spécifier le type du résultat.
Ce constructeur est muni d'une opération permettant de connaître le type de tout objet (équivalente à la fonction de répartition de [BOI, GUI, PAV, 83]).

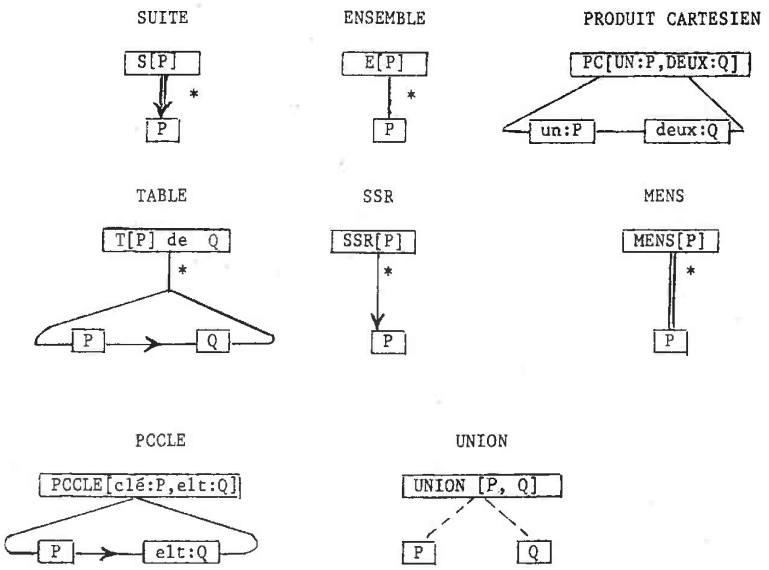
2.2.1.- Caractéristiques des constructeurs de types

Remarques Générales

Notations :

- A chaque constructeur de types nous associons une représentation graphique permettant de visualiser la structure des objets d'un type et les relations $\gtrdot$ entre les types d'un univers ($T \gtrdot T'$ si T se trouve "au-dessus" de T' dans un même arbre).

Représentation graphique des constructeurs de types :



- Les constructeurs de types PC et PCCLE se généralisent à un nombre quelconque de composantes.

- Les constructeurs PC et SUITE sont présentés gracieusement, sans relations entre les constructeurs. Ils sont donc préservants vis à vis de l'environnement comprenant les types BOOL, ENT, P et Q.

- Les constructeurs ENS et TABLE sont présentés gracieusement, avec une relation de commutativité entre les constructeurs. Ils sont donc préservants vis à vis de l'environnement comprenant les types BOOL, ENT, P et Q, modulo la non unicité des formes normales.

- Les constructeurs de types sont tous munis d'un prédicat d'égalité noté eq et de l'opération si-alors-sinon (qui sera notée sous la forme dist-fixée si b alors t sinon t').

- Les constructeurs de types SUITE, ENS et TABLE comportent des opérateurs instanciés par des fonctions.
Ce sont des opérateurs permettant d'appliquer un traitement donné de manière globale à un objet structuré. Nous allons définir deux opérateurs que l'on retrouvera dans les différents constructeurs de types. Il s'agit

- de l'opérateur d'itération $iter_{fct,prop}$ instancié par l'opération fct : Q ← P et le prédicat prop : BOOL ← P qui applique l'opération fct aux éléments de l'objet structuré donné vérifiant la propriété prop.
- de l'opérateur de somme $somme_{fcte}$ instancié par l'opération à valeur entière fcte : ENT ← P calculant la somme des valeurs entières calculées par l'opération fcte sur chaque élément de l'objet structuré donné.

Ces opérations sont définies comme extension des types paramétrés, extension définie sur une extension du type paramètre comportant une opération fct et/ou prop de profil correspondant (cf. § I.2.3).

- Les constructeurs de types sont définis à l'aide des paramètres formels P , Q et/ou R qui comportent une constante indéfinie nommée indef, l'opération si-alors-sinon, et un prédicat d'égalité nommé eq vérifiant les propriétés d'une relation d'équivalence. L'utilisation des opérateurs iter et/ou somme, présuppose que le paramètre effectif comporte des opérations des profils correspondants aux opérations fct, prop et/ou fcte.

Spécification des types paramètres formels utilisés

TYPE P

opérations constructeur indef : P ←

modificateur si-alors-sinon : P ← BOOL,P,P

observateurs eq : BOOL ← P,P

équations

définition du modificateur

si-alors-sinon(vrai,p,p') = p

si-alors-sinon(faux,p,p') = p'

propriétés de l'observateur

eq(p,p) = vrai

eq(p,p') = eq(p',p)

impl(et(eq(p,p'),eq(p',p")),eq(p,p")) = vrai

variables : p,p',p" : P

enrichi éventuellement (pour l'utilisation d'opérateurs) des observateurs suivants :

fct : R ← P

prop : BOOL ← P

fcte : ENT ← P

où R est un type défini

tels que fct(indef) = indef et fcte(indef) = indef

Les paramètres formels Q et R utilisés ont une spécification équivalente.

2.2.2.- Le constructeur PRODUIT CARTESIEN

Un objet d'un type instance du constructeur de produits cartésiens PC, est composé d'un nombre fixé de composantes de types différents correspondant chacune à un champ nommé

- la construction d'un objet de type produit cartésien se fait grâce à l'opération c, paramétrée par les composantes de l'objet.

Exemple : TYPE TPC = PC[un:ENT, deux : BOOL].

Soit tpc un objet de type TPC, tpc est défini par

tpc = c (12,vrai).

Remarque : une des composantes d'un produit cartésien peut être indéfinie (égale à la constante indéfinie "indef").

- la seule modification possible d'un objet de type produit cartésien est la modification d'une de ses composantes. Elle est réalisée par l'opération dont le nom est obtenu en concaténant "modif" et le nom du champ de la composante.

Exemple : modification du champ nommé "un" de tpe défini ci-dessus

tpe' = modifun(tpc,15).

- l'accès aux composantes d'un objet de type produit cartésien se fait à l'aide de l'opération dont le nom est le nom du champ de la composante.

Exemple : un(tpc) = 12 un(tpc') = 15

- deux objets du même type produit cartésien sont égaux s'ils ont les mêmes composantes pour chaque champ.

Exemple : tpc" = c(15,vrai)

tpc' = tpc" car un(tpc') = un(tpc") = 15

deux(tpc') = deux(tpc") = vrai

PC[P,Q] : CONSTRUCTEUR DE TYPES PRODUIT CARTESIEN

se généralise à un nombre quelconque de champs.

P et Q sont les types paramètres formels, ils contiennent les constantes indéfinies indef : P ← indef : Q ←

les prédicats d'égalité eq : BOOL ← P,P eq : BOOL ← Q,Q

opérations de PC[un:P, deux:Q]

constructeur

c : PC[P,Q] ← P,Q — constructeur du produit cartésien à partir des composantes

modificateurs

indef	: PC[P,Q] ←	constante indéfinie
modif.un	: PC[P,Q]←PC[P,Q],P	modification de la valeur du champ nommé "un"
modif.deux	: PC[P,Q]←PC[P,Q],Q	modification de la valeur du champ nommé "deux"
si-alors-sinon	: PC[P,Q]←BOOL,PC[P,Q],PC[P,Q]	

observateurs

un	: P ← PC[P,Q]	accès à la valeur du champ nommé "un"
deux	: Q ← PC[P,Q]	accès à la valeur du champ nommé "deux"
eq	: BOOL ← PC[P,Q],PC[P,Q]	prédicat d'égalité

équations

Def-modif

indef = c(indef,indef)

modif.un(c(p,q),p') = c(p',q)

modif.deux(c(p,q),q') = c(p,q')

si-alors-sinon(vrai,pq,pq') = pq

si-alors-sinon(faux,pq,pq') = pq'

Def-observ

un(c(p,q)) = p

deux(c(p,q)) = q

eq(c(p,q),c(p',q')) = eq(p,p') et eq(q,q')

variables : pq,pq' : PC[P,Q] p,p' : P q,q' : Q

2.2.3.- Le constructeur SUITE

Un objet de type instance du constructeur de suites S (ou SUITE) est composé d'un multi-ensemble d'éléments de même type ordonnés par leur ordre d'adjonction dans la suite. (Une même suite peut comporter deux occurrences d'un même objet).

- la construction d'un objet de type suite se fait en partant de la suite vide "svide" et en ajoutant en tête de la suite des éléments grâce à l'opération "ajout".

  Exemple : TYPE TS = S[ENT]
  ts = ajout(ajout(ajout(svide,zéro),un),deux)

  Remarques : - il est possible de construire directement une suite à partir d'un élément grâce à l'opération "singl" :
  ex : ts1 = singl(un)
  est équivalent à ts1' = ajout(svide,un)
  - il n'est pas autorisé d'ajouter un élément indéfini (c'est-à-dire égal à la constante indéfinie)
  ajout(ts,indef) = ts.

- les opérations de modification d'une suite sont
  - la suppression (sup), l'adjonction (ajoutrg), et la modification (modif) d'un élément de la suite repéré par son rang.
  ex : sup(ts,zéro) = ajout(ajout(svide,zéro),un) (c'est-à-dire suppression de la tête de la suite ts).
  - la suppression (supelt) de toutes les occurrences d'un élément donné de la suite.
  ex : supelt(ts,un) = ajout(ajout(svide,zéro),deux) (c'est-à-dire suppression de l'élément "un" de la suite).

- concaténation de deux suites (conc) ou mise bout à bout :
  ex : conc(ts,ajout(ajout(svide,trois),quatre))
  = ajout(ajout(ajout(ajout(ajout(svide,zéro),un),deux),trois),quatre).

- L'accès aux éléments de la suite se fait par leur rang (position relative par rapport à la tête de la suite) grâce aux opérations "acc" et "tête"

ex : acc(ts,deux) = zéro
tête(ts) = deux (dernier ajouté)

- L'opération "rang" permet de connaître la position de la première occurrence d'un élément donné dans une suite. Si l'élément n'appartient pas à la suite, son rang est indéfini. Le rang de la tête d'uns suite est zéro.
  ex : rang(ts,tête(ts)) = rang(ts,deux) = zéro
  rang(ts,zéro) = deux rang(ts,trois) = indef

- Le prédicat "app" permet de savoir si un élément donné appartient à une suite.
  ex : app(ts,trois) = faux

- Le prédicat "svide?" permet de savoir si une suite est vide. Remarquons qu'une suite indéfinie n'est pas vide : svide?(indef) = faux.

- Deux suites sont égales pour le prédicat "eq" si elles ont des éléments équivalents (pour le prédicat d'équivalence des éléments) ordonnés de la même manière.

- L'opération "taille" permet de connaître le nombre d'éléments d'une suite.

- Le constructeur de types SUITE comprend les deux opérateurs d'itération ($iter_{fct,prop}$) et de somme ($somme_{fcte}$) dont nous avons déjà parlé.

S[P]: CONSTRUCTEUR DE TYPES SUITE

P est le type paramètre (formel)
il contient : une constante indéfinie — indef : P ←
un prédicat d'égalité — eq : BOOL ← P,P
et éventuellement des opérateurs de profil — fct : R ← P
pour définir les opérateurs — prop : BOOL ← P
$iter_{fct,prop}$ et $somme_{fcte}$ — fcte : ENT ← P
où R est un type quelconque

opérations de S[P] : Lexique

constructeurs

svide : S[P] ←	suite vide
ajout : S[P] ← S[P],P	adjonction en tête
indef : S[P] ←	constante indéfinie

cachée

ajoutfn : S[P] ← S[P],P	adjonction effective en tête

modificateurs

singl : S[P] ← P	création d'une suite à partir d'un élément
sup : S[P]←S[P], ENT	suppression d'un élément donné par son rang
supelt : S[P]←S[P],P	suppression de toutes les occurrences d'un élément
ajoutrg : S[P]←S[P],ENT,P	insertion d'un élément à un rang donné
modif : S[P]←S[P],ENT,P	remplacement d'un élément dont le rang est donné
conc : S[P]←S[P],S[P]	concaténation de deux suites
si-alors-sinon : S[P]←BOOL,S[P],S[P]	

observateurs

acc : P ← S[P],ENT	accès à un élément par son rang
tête : P ← S[P]	accès au premier élément (dernier ajouté en date)
rang : ENT ← S[P],P	rang(ou place) d'un élément dans une suite

taille	: ENT ← S[P]	taille (nombre d'éléments) d'une suite
app	: BOOL ← S[P],P	appartenance d'un élément donné à une suite
svide?	: BOOL ← S[P]	une suite est-elle vide ?
eq	: BOOL ← S[P],S[P]	prédicat d'égalité de deux suites

opérateurs

$iter_{fct,prop}$	: S[R] ← S[P]	opérateur appliquant une opération fct : Q ← P aux éléments vérifiant une propriété prop : BOOL ← P
$somme_{fcte}$	: ENT ← S[P]	somme des valeurs entières calculées sur chaque élément par une opération fct' : ENT ← P.

équations

définition du constructeur utilisable

ajout(s,p) = si eq(s,indef) alors indef
sinon si eq(p,indef) alors s
sinon ajoutfn(s,p)

définition des modificateurs

singl(p) = ajout(svide,p)

sup(svide,e) = svide
sup(ajoutfn(s,p),zéro) = s
sup(ajoutfn(s,p),suc(e))
= si eq(e,indef) alors ajoutfn(s,p)
sinon ajoutfn(sup(s,e),p)
sup(indef,e) = indef

supelt(svide,p) = svide
supelt(ajoutfn(s,p),p') = si eq(p,p') alors supelt(s,p')
sinon ajoutfn(supelt(s,p'),p)
supelt(indef,p) = indef

ajoutrg(s,zéro,p) = ajout(s,p)
ajoutrg(svide,suc(e),p) = svide
ajoutrg(ajoutfn(s,p),suc(e),p') = ajoutfn(ajoutrg(s,e,p'),p)
ajoutrg(indef,suc(e),p) = indef
ajoutrg(s,indef,p) = s

modif(s,e,p) = ajoutrg(sup(s,e),e,p)

conc(s,svide) = s
conc(s,ajoutfn(s',p)) = ajoutfn(conc(s,s'),p)
conc(s,indef) = indef

si-alors-sinon(vrai,s,s') = s
si-alors-sinon(faux,s,s') = s'

définition des observateurs

acc(svide,e) = indef
acc(ajoutfn(s,p),zéro) = p
acc(ajoutfn(s,p),suc(e)) = acc(s,e)
acc(ajoutfn(s,p),indef) = indef
acc(indef,e) = indef

tête(s) = acc(s,zéro)

rang(svide,p) = indef
rang(ajoutfn(s,p),p') = si eq(p,p') alors zéro
sinon suc(rang(s,p'))
rang(indef,p) = indef

taille(svide) = zéro
taille(ajoutfn(s,p)) = suc(taille(s))
taille(indef) = indef

app(svide,p) = faux
app(ajoufn(s,p),p') = si eq(p,p') alors vrai
sinon app(s,p')

app(indef,p) = faux

svide?(svide) = vrai
svide?(ajoutfn(s,p)) = faux
svide?(indef) = faux

eq(svide,svide) = vrai
eq(svide,ajoutfn(s,p)) = faux
eq(svide,indef) = faux
eq(ajoutfn(s,p),svide) = faux
eq(ajoutfn(s,p),ajoutfn(s',p')) = eq(p,p') et eq(s,s')
eq(ajoutfn(s,p),indef) = faux
eq(indef,svide) = faux
eq(indef,ajoutfn(s',p')) = faux
eq(indef,indef) = vrai

définition des opérateurs

$iter_{fct,prop}$(svide) = svide

$iter_{fct,prop}$(ajoutfn(s,p)) =

= si prop(p) alors ajout($iter_{fct,prop}$(s),fct(p))
sinon $iter_{fct,prop}$(s)

$iter_{fct,prop}$(indef) = indef

$somme_{fcte}$(svide) = zéro

$somme_{fcte}$(ajoutfn(s,p))

= plus($somme_{fcte}$(s),fcte(p))

$somme_{fcte}$(indef) = indef

variables : s,s' : S[P], p,p' : P, e : ENT

2.2.4.- Le constructeur ENSEMBLE

Un objet de type instance du constructeur ENSEMBLE est composé d'un ensemble d'éléments de même type. Ces éléments ne sont pas ordonnés.

• La construction d'un objet de type ensemble se fait en partant de l'ensemble vide "evide" et en ajoutant les éléments grâce à l'opération "adj".
Exemple : TYPE TE = ENS[ENT]
te = adj(adj(adj(evide,zéro),deux),un)

Remarques : - il est aussi possible de construire directement un ensemble à partir d'un élément grâce à l'opération "single"
ex : tel = single(deux)
est équivalent à tel' = adj(evide,deux)
- l'adjonction à un ensemble d'un élément
• qui lui appartient déjà ou
• qui est indéfini
laisse l'ensemble inchangé :
Si appe(te,e) ou eq(e,indef)
alors adj(te,e) = te

• Les opérations de modification d'un ensemble sont
- la suppression (supe) d'un élément donné de l'ensemble
ex : supe(te,deux) = adj(adj(evide,zéro),un)
- l'opération ensembliste "union" de deux ensembles
ex : union(te,adj(adj(evide,trois),un))
= adj(adj(adj(adj(adj(evide,trois),un),deux),zéro)
(rappelons que les éléments ne sont pas ordonnés).
- l'opération ensembliste d'intersection de deux ensembles "intersec"
ex : intersec(te,adj(adj(adj(evide,zéro),trois),deux)
= adj(adj(evide(zéro),deux)

• L'accès aux éléments d'un ensemble se fait par le prédicat d'appartenance "appe", et s'il s'agit d'un singleton, par l'opération elemsingl.
L'opération "taillee" permet de connaître le nombre d'éléments appartenant à un ensemble.
Le prédicat ensembliste d'inclusion "inclu" permet de savoir si tous les éléments d'un ensemble appartiennent à un premier ensemble.

Le prédicat evide? permet de savoir si un ensemble donné est vide. Un ensemble indéfini n'est pas vide.
Deux ensembles sont égaux pour le prédicat "eq" s'ils contiennent des éléments équivalents (pour le prédicat d'équivalence des éléments).

- Le constructeur de types ENSEMBLE comprend les opérateurs d'itération ($itere_{fct,prop}$) et de sommee ($sommee_{fcte}$) dont nous avons déjà parlé.

E[P] — CONSTRUCTEUR DE TYPES — ENSEMBLE

P est le type paramètre (formel)
il contient une constante indéfinie — indef : P ←
un prédicat d'égalité — eq : BOOL ← P,P
il doit contenir des opérateurs de — fct : R ← P
profils suivants pour pouvoir — prop : BOOL ← P
définir les opérateurs — fcte : ENT ← P
$itere_{fct,prop}$ $sommee_{fcte}$ — où R est quelconque

opérations de E[P]

constructeurs

evide : E[P] ←	ensemble vide
adj : E[P] ← E[P],P	adjonction d'un élément
indef : E[P] ←	constante indéfinie

cachée

adjfn : E[P] ← E[P],P	adjonction effective d'un élément

modificateurs

single : E[P] ← P	création d'un ensemble à partir d'un élément
supe : E[P] ← E[P],P	suppression d'un élément
union : E[P] ← E[P],E[P]	union de deux ensembles
intersec : E[P] ← E[P],E[P]	intersection de deux ensembles
si-alors-sinon : E[P] ← BOOL,E[P],E[P]	

observateurs

elemsingl : P ← E[P]	élément d'un singleton
taillee : ENT ← E[P]	taille(nombre d'éléments)d'un ensemble
inclu : BOOL ← E[P],E[P]	inclusion d'un ensemble dans un autre
appe : BOOL ← E[P],P	appartenance d'un élément à un ensemble
evide? : BOOL ← E[P]	un ensemble est-il vide ?
eq : BOOL ← E[P],E[P]	prédicat d'égalité entre deux ensembles

opérateurs

$itere_{fct,prop}$: E[R] ← E[P]	opérateur appliquant une opération fct : R ← P aux éléments vérifiant une propriété prop : BOOL ← P
$sommee_{fcte}$: ENT ← E[P]	opérateur calculant la somme des valeurs entières calculées sur chaque élément par l'opération fcte : ENT ← P

équations

relation adjfn(adjfn(ep,p),p') = adjfn(adjfn(ep,p'),p)

définition du constructeur caché

```
adj(ep,p) = si eq(ep,indef) alors indef
            sinon si eq(p,indef) ou appe(ep,p)
                  alors ep
                  sinon adjfn(ep,p)
```

définition des modificateurs

```
single(p) = adj(evide,p)

supe(evide,p) = evide
supe(adjfn(ep,p),p') = si eq(p,p') alors ep
                       sinon adjfn(supe(ep,p'),p)
supe(indef,p) = indef

union(evide,ep) = ep
union(adjfn(ep,p),ep') = adj(union(ep,ep'),p)
union(indef,ep) = indef
```

intersec(evide,ep) = si eq(ep,indef) alors indef
sinon evide

intersec(adjfn(ep,p),ep')
= si eq(ep',indef) alors indef
sinon si appe(ep',p) alors adjfn(intersec(ep,ep'),p)
sinon intersec(ep,ep')

intersec(indef,ep) = indef

si-alors-sinon(vrai,ep,ep') = ep

si-alors-sinon(faux,ep,ep') = ep'

Définition observateurs

elemsingl(evide) = indef
elemsingl(adjfn(evide,p)) = p
elemsingl(adjfn(adjfn(ep,p),p')) = indef
elemsingl(indef) = indef

taillee(evide) = zéro
taillee(adjfn(ep,p)) = suc(taillee(ep))
taillee(indef) = indef

inclu(ep,evide) = non(eq(indef,ep))
inclu(ep,adjfn(ep',p')) = appe(ep,p') et inclu(ep,ep')
inclu(ep,indef) = vrai

appe(evide,p) = faux
appe(adjfn(ep,p),p') = si eq(p,p') alors vrai sinon appe(ep,p')
appe(indef,p) = faux

evide?(evide) = vrai
evide?(adjfn(ep,p)) = faux
evide?(indef) = faux

eq(evide,ep) = evide?(ep)

eq(adjfn(ep,p),ep') = appe(ep',p) et eq(ep,supe(ep',p))

eq(indef,evide) = faux

eq(indef,adjfn(ep,p)) = faux

eq(indef,indef) = vrai

Définition des opérateurs

$itere_{fct,prop}(evide) = evide$

$itere_{fct,prop}(adjfn(ep,p)) = si\ prop(p)\ alors\ adj(itere_{fct,prop}(ep),fct(p))$
$sinon\ itere_{fct,prop}(ep)$

$itere_{fct,prop}(indef) = indef$

$sommee_{fcte}(evide) = zéro$

$sommee_{fcte}(adjfn(ep,p)) = plus(fcte(p),sommee_{fcte}(ep))$

$sommee_{fcte}(indef) = indef$

Variables : ep,ep' : E[P] p,p' : P

2.2.5.- Le constructeur TABLE

Un objet de type instance du constructeur TABLE est une représentation d'une fonction définie sur un ensemble d'indices (ou de clés) éléments d'un type donné, à valeur dans un ensemble d'éléments d'un autre type et associant à chaque indice un élément. Ni les indices ni les éléments ne sont ordonnés.

- La construction d'un objet de type table se fait à partir de la table vide "tvide" et en insérant les éléments associés à des indices grâce à l'opération "insert".

  ex : TYPE TT = T[ENT] de BOOL
  tt = insert(insert(insert(tvide,un,vrai),trois,faux),zéro,vrai)

  remarques : - l'insertion d'un élément dont l'indice associé appartient déjà à la table est équivalente à la modification de l'élément associé à l'indice appt(tt,un) ⇒ insert(tt,un,faux)
  = mod(tt,un,faux)
  - il n'est pas possible d'insérer dans une table un élément dont l'indice est indéfini insert(tt,indef,vrai) = tt,
  - cependant il est possible d'associer un élément indéfini à un indice
  insert(tt,quatre,indef) est défini.

- Les modifications possibles d'une table sont
  - la suppression d'un élément repéré par son indice (supt)
    ex : supt(tt,un) = insert(insert(tvide,trois,faux),zéro,vrai)
  - la modification d'un élément repéré par son indice (mod)

ex : mod(tt,zéro,faux) = insert(insert(insert(tvide,un,vrai),trois,faux),
zéro,faux)

où tt est défini ci-dessus.

- L'accès aux éléments d'une table se fait par leur indice : grâce à l'opération "acct".

ex : acct(tt,trois) = faux

L'opération "taillet" permet de connaître le nombre d'indices accessibles d'une table.

ex : taillet(tt) = trois

Le prédicat "appt" indique d'un indice s'il est accessible dans une table.

ex : appt(tt,trois) = vrai

Le prédicat "tvide?" indique si une table donnée est vide.

Une table indéfinie n'est pas vide.

Deux tables sont égales pour le prédicat "eq" si elles ont le même ensemble d'indice accessible et que les éléments associés à chaque indice dans chacune des tables sont égaux (pour un prédicat d'équivalence "eq" du type des éléments).

- Le constructeur de types TABLE comprend les opérateurs d'itération ($itert_{fct,prop}$), et de somme ($sommet_{fcte}$) dont nous avons déjà parlé. Précisons seulement le profil des opérations

fct : R ← PC[P,Q]

prop : BOOL ← PC[P,Q]

fcte : ENT ← PC[P,Q]

pour un type d'expression TABLE[P] de Q.

T[P] de Q — CONSTRUCTEUR DE TYPES — TABLE

P et Q sont des types paramètres (formels)

il contient : une constante indéfinie — indef : P

un prédicat d'égalité — eq : BOOL ← P,P

PC[P,Q] contient des opérateurs de profils suivants : — fct : R ← PC[P,Q]

prop : BOOL ← PC[P,Q]

pour pouvoir définir les opérateurs — fct' : ENT ← PC[P,Q]

où R est quelconque

$itert_{fct,prop}$ $sommet_{fcte}$

Opérations de T[P] de Q

Constructeurs

tvide : T[P] de Q ←	table vide
insert : T[P] de Q ← T[P] de Q,P,Q	insertion d'un élément
indef : T[P] de Q ←	constante indéfinie

cachée

insertfn : T[P] de Q ← T[P] de Q,P,Q	insertion effective d'un élément

modificateurs

supt : T[P] de Q ← T[P] de Q,P	suppression d'un élément donné par son indice
mod : T[P] de Q ← T[P] de Q,P,Q	modification d'un élément
si-alors-sinon : T[P] de Q ← BOOL,T[P] de Q,T[P] de Q	

observateurs

acct : Q ← T[P] de Q,P	accès à un élément donné par son indice
taillet : ENT ← T[P] de Q	taille (nombre d'indices accessibles) d'une table
appt : BOOL ← T[P] de Q,P	appartenance d'un indice donné dans une table
tvide? : BOOL ← T[P] de Q	la table est-elle vide ?
eq : BOOL ← T[P] de Q,T[P] de Q	prédicat d'égalité

opérateurs

$itert_{fct,prop}$: T[P] de R ← T[P] de Q — opérateur appliquant une fonction fct : R ← PC[P,Q] aux éléments vérifiant une propriété prop : BOOL ← PC[P,Q]

$sommet_{fcte}$: ENT ← T[P] de Q — somme des valeurs entières calculées sur chaque élément par l'opération fcte : ENT ← PC[P,Q]

équations

relation : insertfn(insertfn(tb,p,q),p',q') = insertfn(insertfn(tp,p',q'),p,q)

Définition du constructeur caché

```
insert(tb,p,q)
  = si eq(tb,indef) alors indef
    sinon si eq(p,indef) alors tb
          sinon si appt(tb,p)
                alors insertfn(supt(tb,p),p,q)
                sinon insertfn(tb,p,q)
```

Définition des modificateurs

```
supt(tvide,p) = tvide
supt(insertfn(tb,p,q),p') = si eq(p,p') alors tb
                            sinon insertfn(supt(tb,p'),p,q)
supt(indef,p) = indef

mod(tb,p,q) = insert(supt(tb,p),p,q)

si-alors-sinon(vrai,tb,tb') = tb
si-alors-sinon(faux,tb,tb') = tb'
```

Définition des observateurs

```
acct(tvide,p) = indef
acct(insertfn(tb,p,q),p') = si eq(p,p') alors q
                            sinon acct(tb,p')
acct(indef,p) = indef
```



```
taillet(tvide) = zéro
taillet(insertfn(tb,p,q)) = suc(taillet(tb))
taillet(indef) = indef

appt(tvide,p) = faux
appt(insertfn(tb,p,q),p') = si eq(p,p') alors vrai
                            sinon appt(tb,p')
appt(indef,p) = faux

tvide?(tvide) = vrai
tvide?(insertfn(tb,p,q)) = faux
tvide?(indef) = faux

eq(tvide,tb) = tvide?(tb)
eq(insertfn(tb,p,q),tb') = appt(tb',p) et
      eq(acct(tb',p),q) et eq(tb,supt(tb',p))
eq(indef,tvide) = faux
eq(indef,insertfn(tb,p,q)) = faux
eq(indef,indef) = vrai
```

Définition des opérateurs

$itert_{fct,prop}(tvide) = tvide$

$itert_{fct,prop}(insertfn(tb,p,q)) =$

= si prop(c(p,q)) alors insert($itert_{fct,prop}(tb)$,p,fct(c(p,q)))
  sinon $itert_{fct,prop}(tb)$

$itert_{fct,prop}(indef) = indef$

$sommet_{fcte}(tvide) = zéro$

$sommet_{fcte}(insert(tb,p,q)) = plus(sommet_{fcte}(tb),fcte(c(p,q)))$

$sommet_{fcte}(indef) = indef$

variables : tb,tb' : T[P] de Q p,p' : P q,q' : Q

3.- UNIVERS DE TYPES

Nous avons vu comment se construisait une spécification fonctionnelle. Nous avons défini un langage de présentation de types qui doit nous permettre d'exprimer une spécification fonctionnelle en termes de types abstraits.

Nous l'appelons univers de types. Il se construit de manière incrémentale. Chaque étape précise un peu plus un des types de l'univers. Il y a cependant des règles à respecter : en effet chaque nouvelle version d'un type doit être issue de la précédente.

Nous étudions dans un premier temps comment construire un univers de types. Puis nous approfondissons l'étude de la relation "être issu de".

3.1.- Définition et construction d'un univers

Nous voulons construire une spécification en termes de types abstraits algébriques, appelée univers, équivalente à une spécification fonctionnelle donnée. Pour cela il faut

- associer chaque fonction ou opération définie à un et un seul type, et
- définir pout tout type l'ensemble de ses opérations utilisables.

Nous avons besoin pour cela de définir deux relations sur les types : une relation d'ordre notée "$\rangle$" pour la règle d'association, et une relation "utilise".

Il est possible de définir l'univers en parallèle de la construction de la spécification fonctionnelle. A chaque étape soit un nouveau type est introduit soit une nouvelle opération.

3.1.1.- Relation d'ordre "$\rangle$" et relation "utilise"

Les relations "$\rangle$" et "utilise" sont définies sur des ensembles de types appelés univers.

Définition : Un univers est un ensemble formé de types présentés en SPES-TYPES, spécifiant un problème.

La relation d'ordre "$\rangle$" compare les types en ne considérant que leur expression. Mais ce qui nous intéresse c'est l'expression "en terme de types de

base ou de constructeurs de types instanciés", ce que nous appelons la structure d'un type. Cette notion de structure d'un type est très utile. C'est elle qui apparaît dans les graphes des types d'un univers. C'est aussi elle qui facilitera la recherche d'un type dans une bibliothèque. Elle exprime la structure intuitive des objets des types manipulés.

Définition : La structure d'un type T est

- l'expression de T si celle-ci est un type de base B, ou un constructeur de types C instancié par T1...TN et
- la structure du type de l'expression de T si celui-ci est un type défini T'.

Structure (T) = Si expr(T) = T' où T' est un type défini
alors Structure(expr(T))
Si expr(T) = C[T1...TN] ou expr(T) = B
où C est un constructeur de types et B un type de base.
alors expr(T).

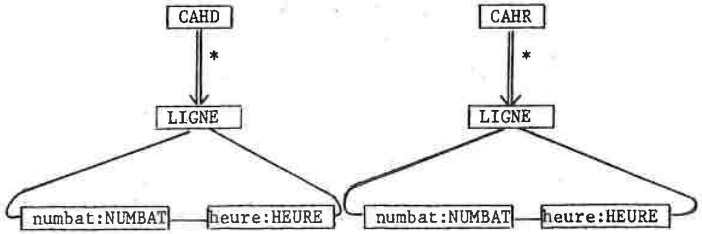
Remarque : Un type pouvant être défini uniquement par une expression, la structure d'un type, est une définition de type. Nous parlerons du type structure d'un type donné.

Exemples :

- Reprenant l'exemple des types NOMBRE et NOTEL étudiés au paragraphe II.1.2.1., la structure de NOMBRE et de NOTEL est la même, c'est l'expression de NOMBRE, une suite de chiffres : SUITE[CHIFFRE].
- Reprenons l'exemple du loueur de bateaux étudié au paragraphe II.1.1.1. Les types de la spécification sont

  CAHD,CAHR d'expression SUITE[LIGNE]
  LIGNE d'expression PC[numbat:NUMBAT,heure:HEURE]
  NUMBAT d'expression ENT
  HEURE d'expression ENT

Pour tous ces types, leur structure est leur expression. Le graphe de ces types est le suivant :



La relation d'ordre "$\succ$" relie les types apparaissant dans un même arbre d'un graphe de types. Un type donné sera plus grand pour "$\succ$" qu'un type se situant "en dessous" de lui dans un arbre.

Définition : Soit u un univers. La relation notée "$\succ$" est définie comme étant la fermeture transitive stricte de la relation "est un paramètre effectif de la structure de" ou "est égal à"

T $\succ$ T1 ssi T1 "est un paramètre effectif de la structure de" T
ou T = T1
ou il existe TP tel que
TP "est un paramètre effectif de la structure de" T et
TP $\succ$ T1
sinon T et T1 sont incomparables pour "$\succ$"

Rappel : Soit E un ensemble, et R une relation sur E. La fermeture transitive stricte de R est la plus petite relation transitive contenant R.

Notation : Si T $\succ$ T1 on dit alors que T est plus grand ou égal pour "$\succ$" que T1.

Exemples :

- NOMBRE $\succ$ CHIFFRE, NOTEL $\succ$ CHIFFRE
  NOMBRE et NOTEL sont incomparables pour $\succ$

- CAHD $\succ$ LIGNE $\succ$ NUMBAT
  CAHR $\succ$ LIGNE $\succ$ HEURE

Propriété :

"$\succ$" est une relation d'ordre :
Elle est une réflexive et transitive par définition.
Les types récursifs n'étant pas admis, elle est antisymétrique.
Elle est partielle sur un univers.

Remarque : Les types de base sont minimaux pour "$\succ$" : si B est un type de base, il n'existe pas de type T différent de T tel que B $\succ$ T.

La relation "utilise" permet de connaître l'ensemble des opérations d'une spécification utilisant une opération donnée, dans les équations les définissant.

Définition : Soient op1 et op2 deux opérations d'une spécification. On dit que op1 utilise op2 si et seulement si op2 apparaît dans les équations définissant op1.

Par extension on dit qu'un type T1 utilise les opérations op1...opn d'un type T2 si et seulement si les opérations définies de T1 utilisent les opérations op1...opn.

Exemples : Prenons la spécification du loueur de bateaux.
L'opération départb utilise dans sa définition les opérations rentré et numbat.
L'opération rentré utilise les opérations pasparti et pas reparti ainsi que l'opération de conjonction ou.

Remarque : Une opération utilise les opérations intermédiaires introduites lors de l'analyse déductive.

3.1.2.- Construction d'un univers à partir d'une spécification fonctionnelle

Une spécification fonctionnelle est constituée d'un ensemble de fonctions définies par des équations et d'un ensemble de types définis par une expression et un invariant. Pour obtenir une spécification en termes de types abstraits algébriques présentés en SPES-TYPES, il suffit d'associer chaque fonction à un et un seul type puis de définir pour chaque type l'ensemble de ses opérations utilisables, importées et définies, ensemble précisé en SPES-TYPES pour chaque type et non précisé dans la spécification fonctionnelle.

Pour associer une opération à un type, nous nous guidons sur les types de son profil. Ayant suivi une démarche déductive, il semble naturel d'associer une opération au type le plus grand pour $\succ$ de son profil :

Exemple : l'opération pas parti de la spécification du loueur de bateaux, a le profil suivant :

pas parti : BOOL ← CAHD,NUMBAT

Rappelons que l'expression de CAHD est une suite de lignes, chaque ligne étant composée d'un numéro de bateaux et d'une heure. L'opération pas parti est un prédicat permettant de savoir si un bateau donné n'a pas de départ enregistré dans le cahier des départs de type CAHD. Il est bien naturel d'associer pas parti à CAHD.

Cette règle ne suffit pas à couvrir tous les cas. En effet il se peut qu'il y ait plusieurs types "plus grands pour $\succ$" que tous les autres dans le profil d'une opération. Dans ce cas nous n'étudions que les types du domaine de l'opération : nous aurons alors des observateurs, au lieu de constructeurs d'objets à partir d'autres objets. Deux cas peuvent alors se présenter:

- soit il y a un type du domaine de l'opération qui soit plus grand que tous les autres. Dans ce cas nous lui associons l'opération.
- soit il y en a plusieurs. Dans ce cas nous associons l'opération au produit cartésien de ces types. Il faut alors modifier la définition de l'opération en remplaçant les variables et les termes de type d'une des composantes du produit cartésien par la projection correspondante.

Règle d'association d'une opération à un type

Soit fct : $T_0 \leftarrow T_1 \ldots,T_n$ une opération définie par les équations equat

Si $\exists T_i$, $0 \leq i \leq n$ tel que $T_i \succ T_j \quad \forall j = 0,\ldots n$

Alors Cas 1 : fct est associée à T_i

Sinon Si $\exists T_i$, $1 \leq i \leq n$ tel que $T_i \succ T_j \quad \forall j = 1,\ldots n$

Alors Cas 2 : fct est associée à T_i

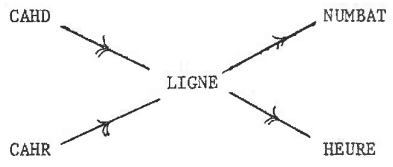
Sinon Cas 3 : Soit $\{T_{i_1}\ldots T_{i_m}\} \subset \{T_1\ldots T_n\}$ tel que les propriétés suivantes soient vérifiées

- Pour tout $T_j, j \in \{1\ldots n\}$, tel qu'il existe $k \in \{1\ldots m\}$ avec $T_k \succ T_j$ alors $T_j \notin \{T_{i_1}\ldots T_{i_m}\}$
- $\forall j,k \in \{i_1,\ldots,i_m\}$ alors T_j et T_k sont incomparables pour $\succ$

Alors fct est associée à $PC[t_{i_1}:T_{i_1},\ldots,t_{i_m}:T_{i_m}]$ où $t_{i_1}\ldots t_{i_m}$ sont les noms des composantes du produit cartésien. fct est alors définie par les équations equat $[t_1,\ldots t_m \leftarrow t$ et $t_j \leftarrow T_{i_j}(t)$ $\forall t_1\ldots t_m$ sous termes de equat tq sorte$(t_j) = T_{ij}$ $\forall j = 1\ldots m$ et sorte$(t) = PC[t_{i_1}:T_{i_1},\ldots,t_{i_m}:T_{i_m}]$.

Remarque : Selon cette règle, les opérations définies d'un type T n'utilisent que des opérations associées à des types plus petits ou incomparables pour $\succ$ que T.

Exemples : Associons les opérations définies dans la spécification fonctionnelle du loueur de bateaux étudiée au paragraphe II.1.1.1., aux types ordonnés par $\succ$, de la manière suivante :



Etudions le cas de l'opération départb : CAHD ← CAHD,CAHR,LIGNE

- CAHD et CAHR sont plus grands que LIGNE pour $\succ$ et ils sont incomparables. C'est dans le cas (3) qui s'applique. départb est associée au type DOUBLE-CAH défini par :

TYPE DOUBLE-CAH = PC[cahdep : CAHD, cahret : CAHR]

Nouvelle définition de départb :

départb : DOUBLE-CAH ← DOUBLE-CAH,LIGNE
départb(dcah,ℓ)
= si rentré(dcah,numbat(ℓ))
alors modif-cahdep(dcah,ajout(cahdep(dcah),ℓ))
sinon dcah

De la même manière, l'opération retourb est associé à DOUBLE-CAH et définie par :

retourb : DOUBLE-CAH ← DOUBLE-CAH,LIGNE
retourb(dcah,ℓ)
= si rentré(dcah,numbat(ℓ))
alors dcah
sinon modif-cahret(dcah,ajout(cahret(dcah),ℓ))

Il en va de même pour l'opération rentré associée à DOUBLE-CAH et définie par :

rentré : BOOL ← DOUBLE-CAH,NUMBAT
rentré(dcah,b) = pas parti(cahdep(dcah),b)
ou pas reparti(dcah,b)

et pour l'opération pas reparti définie par

pas reparti : BOOL ← DOUBLE-CAH,NUMBAT
par reparti(dcah,b) =
= reldepret(derdep(cahdep(dcah),b),derret(cahret(dcah),b))

L'opération pas parti : BOOL ← CAHD, NUMBAT est associée par application du cas (1), au type CAHD.

L'opération derdep : LIGNE ← CAHD,NUMBAT est associée par application du cas (1), au type CAHD.

L'opération derret : LIGNE ← CAHR,NUMBAT est associée par application du cas (1), au type CAHR.

L'opération reldepret : BOOL ← LIGNE,LIGNE est associée par application du cas (1), au type LIGNE.

Définition des ensembles d'opérations utilisables des types d'un univers

L'ensemble des opérations utilisables d'un type T doit contenir toutes les opérations du type T utilisées par les autres types, ainsi que les opérations définissant une des fonctionnalités du problème.

De plus, par convention, les constructeurs des objets d'un type doivent toujours être utilisables.

Exemple : Définissons les ensembles d'opérations utilisables des types de la spécification fonctionnelle (modifiée par l'exemple ci-dessus) du loueur de bateaux.

TYPE DOUBLE-CAHIER :
opérations importées utilisables : c(svide,svide)
opérations définies utilisables : départb,retourb
constructeurs : c(svide,svide),départb,retourb

TYPE CAHD :
opérations importées utilisables : svide,ajout
opérations définies utilisables : pas parti, derdep

TYPE CAHR :
opérations importées utilisables : svide,ajout
opérations définies utilisables : derret

TYPE LIGNE :
opérations importées utilisables : numbat,heure,c,eq,indef
opération définie utilisable : reldepret

TYPE NUMBAT :
opérations importées utilisables : zéro,suc,eq

TYPE HEURE :

opérations importées utilisables : zéro,suc,infeg

Remarques sur cet exemple :

- les opérations rentré et pas reparti, associées à DOUBLE-CAHIER ne sont pas des fonctionnalités du système (ici le système ne comporte que deux fonctionnalités : départb et retourb) et ne sont pas utilisées par d'autres types.
- les opérations eq et indef du type LIGNE sont utilisées par le type CAHD dans la définition de pas parti.
- c est le constructeur des objets d'un produit cartésien.
- l'opération eq du type NUMBAT est utilisée par le type CAHD dans la définition de derdep, par le type CAHR dans la définition de derret et par le type LIGNE dans la définition de reldepret.
- l'opération infeg du type HEURE est utilisée par le type LIGNE dans la définition de reldepret.
- les opérations c, svide, ajout, zéro, suc sont utilisables parce que constructeurs des objets de leurs types respectifs.

3.1.3.- Construction d'un univers en parallèle d'une spécification fonctionnelle

La définition d'un univers peut se faire en parallèle de la construction de la spécification fonctionnelle. En suivant la démarche proposée par le méta-algorithme, celle-ci se construit de manière déductive et à chaque étape on définit soit un type soit une opération.

Adjonction d'un type dans un univers :

Lors de son adjonction dans un univers, un type

- est défini par une expression et éventuellement par un invariant,
- importe toutes les opérations utilisables de son expression (en effet on ne sait pas lesquelles seront utilisées en conséquence, on laisse toutes les possibilités), et
- ne possède aucune opération définie propre (suivant une démarche déductive on ne définit des opérations qu'après avoir défini le ou les types de son profil auxquels elle sera associée).

Remarque : Dans le cas où le type de l'expression est un type défini, le type hérite de l'ensemble des opérations utilisables importées et définies. L'invariant devra être plus restrictif que l'invariant du type de l'expression.

Adjonction d'une opération dans un univers :

Une opération ajoutée à un univers enrichit l'ensemble des opérations définies du type auquel elle est associée. Elle n'est utilisable que si utilisée par une opération associée à un autre type. Elle peut "remplacer" une opération importée et donc impliquer une restriction de l'ensemble des opérations importées.

Une fois la spécification fonctionnelle achevée, il est nécessaire de préciser l'ensemble des opérations importées utilisables des types de l'univers associé.

Comme nous l'avons déjà vu, cet ensemble ne doit contenir que les opérations utilisées par d'autres types, ainsi que les constructeurs (si ceux-ci sont les constructeurs importés). Cet ensemble est donc une restriction de l'ensemble des opérations importées utilisables obtenu lors de la définition de la spécification fonctionnelle.

Nous pouvons résumer ces étapes de la manière suivante :

- adjonction de T dans l'univers u
  T est défini par l'expression EXPRT et l'invariant invar(T)
  opimputil(T) = opimputil(EXPRT),
  opdefutil(T) = opdefutil(EXPRT)
  (si EXPRT est un type de base ou un constructeur instancié,
  opimputil(T) = oper(EXPRT)
  opdefutil(T) = ∅).
- adjonction de fct dans l'univers u
  fct est associé au type T de l'univers

opdef(T) est enrichi de fct

Si ∃fct' associée à T', T' ≠ T tq fct' utilise fct

Alors opdefutil(T) est enrichi de fct.

Sinon fct est une opération cachée de T.

Si fct "remplace" une opération fct' importée de T

(fct remplace une opération fct' importée de T si fct = fct')

alors opimputil(T) est restreint de fct'.

- Une fois la définition de la spécification fonctionnelle terminée, il faut restreindre les ensembles des opérations importées utilisables de tous les types de l'univers.

3.1.4.- Exemple

Univers u_o déduit de la spécification fonctionnelle du loueur de bateaux (§ 2.1.1.1.).

Le type DCAH des doubles cahiers est introduit comme indiqué ci-dessus.

L'univers u_o est composé des types suivants :

DOUBLE-CAH	CAHD	LIGNE	NUMBAT
	CAHR		HEURE

TYPE DOUBLE-CAH = PC[cahdep : CAHD, cahret : CAHR]

opérations utilisables	déclarations de variables
importées : c(svide,svide)	dcah : DOUBLE-CAH
définies : (c)départb,(c)retourb	ℓ : LIGNE
	b : NUMBAT

définitions

(c) départb : DOUBLE-CAH ← DOUBLE-CAH,LIGNE

départb(dcah,ℓ)

= si rentré(dcah,numbat(ℓ))

alors modifcahdep(dcah,ajout(cahdep(dcah),ℓ))

sinon dcah

(c) retourb : DOUBLE-CAH ← DOUBLE-CAH,LIGNE

retourb(dcah,)

= si rentré(dcah,numbat(ℓ))

alors dcah

sinon modifcahret(dcah,ajout(cahret(dcah),ℓ))

rentré : BOOL ← DOUBLE-CAH,NUMBAT

rentré(dcah,b) = pas parti(cahdep(dcah),b)

ou pas reparti(dcah,b)

pas reparti : BOOL ← DOUBLE-CAH,NUMBAT

pas reparti(dcah,b) = reldepret(derdep(cahdep(dcah),b),

derret(cahret(dcah),b))

Lexique

TYPE DOUBLE CAH : type des doubles cahiers composés d'un cahier des départs (cahdep) et d'un cahier des retours (cahret)

départb : enregistrement dans le cahier des départs du double-cahier, du départ d'un bateau

retourb : enregistrement dans le cahier des retours du double cahier, du retour d'un bateau

rentré : prédicat indiquant si un bateau n'est pas parti du tout ou s'il est rentré après son dernier départ

pas reparti : prédicat indiquant si un bateau dont un retour a été enregistré, n'a pas de départ postérieur enregistré

TYPE CAHD = SUITE[LIGNE]

opérations utilisables	déclarations de variables
importées : svide,ajout définies : par parti,derdep	cdep : CAHD b : NUMBAT

définitions

pas parti : BOOL ← CAHD,NUMBAT

par parti(cdep,b) = eq(indef,derdep(cdep,b))

derdep : LIGNE ← CAHD,NUMBAT

derdep(cdep,b) = tête($iter_{id,eq(numbat(.),b)}$(cdep))

Lexique

TYPE CAHD = type du cahier des départs du double cahier

pas parti : prédicat indiquant qu'un bateau n'est pas enregistré dans le cahier des départs

derdep : accès au dernier départ enregistré d'un bateau

TYPE CAHR = SUITE[LIGNE]

opérations utilisables	déclaration de variables
importées : svide,ajout définie : derret	cret : CAHR b : NUMBAT

définition

derret : LIGNE ← CAHR,NUMBAR

derret(cret,b) = tête($iter_{id,eq(numbat(.),b)}$(cret))

Lexique

TYPE CAHR = type du cahier des retours du double-cahier

derret : accès au dernier retour enregistré d'un bateau

TYPE LIGNE = PC[numbat : NUMBAT, heure : HEURE]

opérations utilisables	déclaration de variables
importées : numbat,heure,c,eq, indef définie : reldepret	ℓ,ℓ' : LIGNE

définition

reldepret : BOOL ← LIGNE,LIGNE

reldepret(ℓ,ℓ') = eq(numbat(ℓ),numbat(ℓ'))
et infeg(heure(ℓ),heure(ℓ'))

Lexique

TYPE LIGNE = type des éléments des types CAHD et CAHR

reldepret : prédicat indiquant si deux lignes peuvent correspondre à une même location : même numéro de bateau et heure de la première ligne inférieure ou égale à celle de la seconde.

TYPE NUMBAT = ENT

opérations utilisables importées : zéro,suc,eq

Lexique :

TYPE NUMBAT = type des numéros de bateaux

TYPE HEURE = ENT

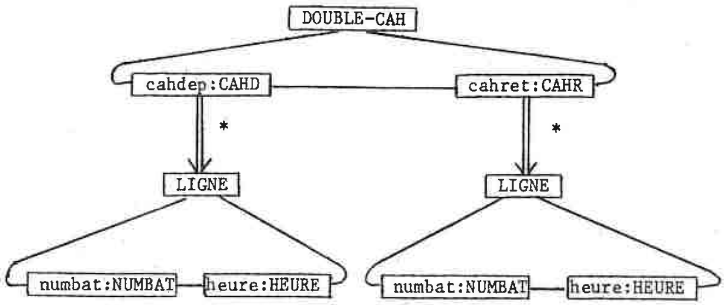
invariant : infeg(zéro,h) et infeg(h,23)

opérations utilisables	déclaration de variable
importées : zéro,suc,infeg	h : HEURE

Lexique

TYPE HEURE : type des heures de départ et de retour des bateaux

Graphe des types de l'univers $\mathcal{U}_o$:



3.2.- Etude de la relation "être issu de"

Nous avons vu que lors de la définition d'un univers, un type est défini par une expression, un invariant (toujours plus restrictif que celui du type de son expression), une restriction de l'ensemble des opérations importées et un enrichissement de l'ensemble des opérations définies (du type de son expression). Nous disons qu'il est issu de ce type. Ainsi les différentes "versions" d'un type au cours de la définition d'un univers, sont issues les unes des autres.

Etudions plus formellement cette relation ainsi que la relation "être parents" et la relation d'équivalence "$\equiv$" qui en découlent.

Soient T1 et T2, deux types présentés en SPES-TYPES, différents du type VIDE.

Définition : T1 et T2 sont équivalents, ce que l'on note $T1 \equiv T2$ si et seulement si leurs types structures sont équivalents, ils ont le même ensemble d'opérations utilisables et leurs invariants sont équivalents :

- structure(T1) = structure(T2)
  ou Si structure(T1) = $C[T1_1,\ldots,T1_n]$
  et structure(t2) = $C[T2_1,\ldots,T2_n]$
  où C est un constructeur de types
  Alors $T1_i \equiv T2_i$ pour $i = 1,\ldots,n$
- oputil(T1) = oputil(T2)
- invar(T1) eq invar(T2)

Propriété : La relation "$\equiv$" est une relation d'équivalence.

Démonstration : Evidente puisque l'égalité syntaxique notée =, et eq sont des relations d'équivalence.

Exemple : Dans l'exemple du loueur de bateaux, les types CAHD et CAHR ne sont pas équivalents car ils n'ont pas le même ensemble d'opérations définies utilisables.
Cependant en enrichissant CAHR de l'opération
pas rentré : BOOL ← CAHR,NUMBAT

définie par

pas rentré(cret,b) = eq(indef,derret(cret,b)),

les types CAHD et CAHR sont alors équivalents.

Remarque : On peut généraliser cette définition en introduisant une relation d'équivalence sur les opérations. Nous ne le faisons pas ici.

Nous pouvons maintenant définir la relation "être issu de" :

Définition : T2 est issu de T1, noté $T1 \rightsquigarrow T2$ si et seulement si

- leurs types structures sont équivalents,
- l'ensemble des opérations importées utilisables de T2 est une restriction de celui de T1,
- l'ensemble des opérations définies utilisables de T2 est un enrichissement de celui de T1,
- l'invariant de T2 est plus restrictif que celui de T1.

T1 $\rightsquigarrow$ T2		
structure (T1)	$\equiv$	structure(T2)
opimputil(T1)	$\supseteq$	opimputil(T2)
opdefutil(T1)	$\subseteq$	opdefutil(T2)
invar(T1)	impl	invar(T2)

Propriétés

1.- Soit $\mathcal{C}$ l'ensemble des types présentés en SPES-TYPES.
Soit $\mathcal{C}|_{\equiv}$ le quotient de $\mathcal{C}$ par la relation d'équivalence $\equiv$.
Soit [T] la classe d'équivalence du type T.

On définit la relation $\overset{\equiv}{\rightsquigarrow}$ sur les classes de $\mathcal{C}$ par

[T1] $\overset{\equiv}{\rightsquigarrow}$ [T2] si et seulement si $\forall T1' \in [T1], T2' \in [T2]$ $T1' \rightsquigarrow T2'$.

La relation $\overset{\equiv}{\rightsquigarrow}$ est une relation d'ordre sur $\mathcal{C}|_{\equiv}$.

Démonstration : Structure (T1') $\equiv$ structure (T2') $\forall T1' \in [T1], \forall T2' \in [T2]$
$\supseteq$, $\subseteq$ et impl sont des relations d'ordre.

Notation : Si [T1] $\overset{\equiv}{\rightsquigarrow}$ [T2] alors on dit que [T1] est plus petit que [T2].

2.- Tout type T est issu du type de son expression :

$expr(T) \rightsquigarrow T$

Le type VIDE est issu de tout type.

Démonstration : Notons EXPRT le type de l'expression de T.

- Par définition de structure :
  structure(T) = EXPRT si EXPRT est un type de base ou un constructeur de types instancié alors structure(EXPRT) = EXPRT = structure(T)
  structure(T) = structure(EXPRT) si EXPRT est un type défini
- opimputil(T) est une restriction de opimputil(EXPRT)
- opdefutil(T) est un enrichissement de opdefutil(EXPRT)
- l'invariant de T est toujours plus restrictif que celui de EXPRT.

Le type VIDE a un invariant égal à faux et faux impl invar(T) = vrai.

3.- Tout type T est issu de son type structure.

Exemple : Soit l'univers contenant les types suite croissante d'entiers SCROIS et suite croissante d'entiers de taille limitée SCROISLIM spécifiés par :

TYPE SCROIS = SUITE ENT	
opérations utilisables importées : toutes sauf ajout, ajoutième et modifième définies : (c)ajcr	déclarations de variables sc : SCROIS e : ENT
(c)ajcr : SCROIS $\times$ SCROIS,ENT ajcr(sc,e) = ajoutième(sc,rang(sc,tête($iter_{id,infeg(e,.)}$(sc))),e)	
Lexique : TYPE SCROIS est le type des suites croissantes d'entiers ajcr: nouveau constructeur adjonction d'un élément préservant l'ordre croissant de la suite	

TYPE SCROISLIM = SCROIS	
invariant : infeg(taille(scℓ),10)	
opérations utilisables importées : toutes sauf ajout, ajoutième,modifième définies : (c)ajcr,peutonaj	déclarations de variables scℓ: SCROISLIM
peutonaj : BOOL ← SCROISLIM peutonaj(scℓ) = infeg(taille(scℓ),9)	
Lexique : TYPE SCROISLIM est le type des suites croissantes d'entiers de taille limitée à 10 éléments. peutonaj est un prédicat indiquant si l'on peut ajouter un élément de plus à la suite	

Le type SCROISLIM est issu du type SCROIS

SCROIS ⇝ SCROISLIM

Définition : T1 et T2 sont parents noté T1 ≁ T2 si et seulement si
- ils ont des types structure équivalents
- leurs invariants ne sont pas incompatibles.

T1 ≁ T2
structure(T1) ≡ structure(T2) invar(T1) et invar(T2) ≠ faux

Remarques : - cette relation ne prend pas en compte les opérations des types.
- dire que deux invariants sont incompatibles signifie que l'ensemble des objets (du type défini par l'expression) vérifiant les deux invariants est vide (ce qui est exprimé par invar(T1) et invar(T2) = faux).

Exemple : Reprenons l'univers comprenant les types SCROIS et SCROISLIM en lui ajoutant le type SDECROISLIM des suites décroissantes d'entiers de taille limitée à 12.

TYPE SDECROISLIM = SUITE[ENT]	
invariant : infeg(taille(sdℓ),12)	
opérations utilisables importées : toutes sauf ajout,ajoutrg modif définies : (c)ajdecr	déclarations de variables sdℓ : SDECROISLIM e : ENT
(c) ajdecr : SDECROISLIM ← SDECROISLIM,ENT ajdecr(sdℓ,e)=ajoutrg(sdℓ,rang(sdℓ,tête(iter$_{id,infeg(.,e)}$(sdℓ))),e)	
Lexique : TYPE SDECROISLIM est le type des suites décroissantes d'entiers de taille limitée à 12 adjdecr : nouveau constructeur, adjonction d'un élément préservant l'ordre décroissant de la suite	

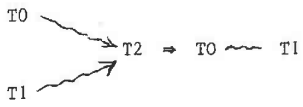
Les types SCROIS, SCROISLIM et SPECROISLIM sont parents.

Propriétés : Soient T0, T1 et T2 trois types différents du type VIDE.

1.- "sont parents" est une relation réflexive et symétrique.

2.- Si T1 est issu de T2 alors ils sont parents.
T2 ⇝ T1 ⇒ T2 ≁ T1

3.- Si T2 est issu de T0 et de T1 alors T0 et T1 sont parents.



Démonstrations :

1 et 2 : évidentes

3 : T2 est issu de T0 et de T1

donc - structure(T2) ≡ structure(T0)

et structure(T2) ≡ structure(T1)

d'où structure(T0) ≡ structure(T1)

- invar(T2) impl invar(T0)

et invar(T2) impl invar(T1)

d'où quand invar(T2) est vrai

invar(T0) et invar(T1) est vrai

donc T0 et T1 sont parents.

Remarque : Si T1 et T0 sont issus de T2

T1 et T0 ne sont pas forcément parents.

Contre-exemple : Soient T0, T1, T2 trois types issus de ENTIER

invar(T0) = infeg(0,x) et infeg(x,4) $(0 \leq x \leq 4)$

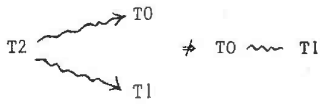
invar(T1) = infeg(6,x) et infeg(x,10) $(6 \leq x \leq 10)$

invar(T2) = infeg(0,x) et infeg(x,10) $(0 \leq x \leq 10)$

On a bien invar(T0) impl invar(T2)

et invar(T1) impl invar(T2)

mais invar(T0) et invar(T1) = faux



$\not\Rightarrow$ T0 ⇝ T1

(la relation n'est pas transitive pour cette même raison).

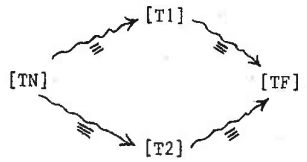
Soit $\mathcal{C}$ l'ensemble des types présentés en SPEC-TYPES.

Soit $\mathcal{C}|_{\equiv}$ le quotient de $\mathcal{C}$ par la relation d'équivalence $\equiv$.

La relation d'ordre $\mathcal{C}|_{\equiv}$ définit un treillis complet sur l'ensemble des (classes de) types de types structures équivalents :

$\forall [T1],[T2] \in \mathcal{C}|_{\equiv}$ tq structure([T1]) ≡ structure([T2])

$\exists$[TN] et [TF] tels que



Définitions :

Le noyau de [T1] et [T2] est la plus grande classe de types dont [T1] et [T2] soient issus.

La fusion de [T1] et [T2] est la plus petite classe de types qui soit issue de [T1] et de [T2].

Propriétés :

La classe du type structure est la plus petite des classes de types de types structures équivalents :

$\forall T1 \in \mathcal{C}$ [structure(T1)] $\overset{\equiv}{\rightsquigarrow}$ [T1]

La classe du type VIDE est la plus grande des classes de types de types structures équivalents :

$\forall T1 \in \mathcal{C}$ [T1] $\overset{\equiv}{\rightsquigarrow}$ [VIDE].

Nous allons caractériser le noyau et la fusion de deux types :

Définition : Soient T1 et T2 $\in \mathcal{C}$ tels que structure(T1) ≡ structure(T2). Le noyau de T1 et T2 noté TN = noyau(T1,T2) est défini par :

- l'expression de TN est celle de T1
- les opérations importées utilisables de TN sont celles de T1 et celles de T2
- les opérations définies utilisables de TN sont celles appartenant à l'intersection de celles de T1 et de celles de T2
- l'invariant de TN est la disjonction de ceux de T1 et de T2

TN = noyau(T1,T2)	
expr(TN)	= expr(T1)
opimputil(TN)	= opimputil(T1) ∪ opimputil(T2)
opfedutil(TN)	= opdefutil(T1) ∩ opdefutil(T2)
invar(TN)	= invar(T1) ou invar(T2)

Remarque : Le noyau de deux types T1 et T2 est un type à partir duquel "on peut définir" T1 et T2. Il possède donc plus d'opérations importées, moins d'opérations définies et son invariant étant moins restrictif que ceux de T1 et de T2, il possède plus d'objets.

Exemple : Reprenons l'univers composé des types des suites croissantes et décroissantes

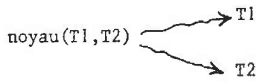
SCROIS,SCROISLIM et SDECROISLIM

TYPE noyau (SCROISLIM, SDECROISLIM) = SUITE[ENTIER]	
invariant : infeg(taille(sncrde),12)	
opérations utilisables	déclaration de variable
importées : toutes sauf ajout, ajoutrg et modif	sncrde : noyau(SCROISLIM, SDECROISLIM)

Ainsi défini, le noyau de deux types possède bien les propriétés souhaitées.

Propriétés :

1.- T1 et T2 sont issus de noyau(T1,T2)



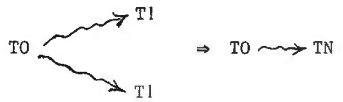
Démonstration : Notons TN le noyau de T1 et de T2 : TN = noyau(T1,T2)

Il suffit de démontrer que T1 est issu de TN,

- expr(TN) = expr(T1)
  or expr(T1) ≡ expr(T2) par définition de parents
- opimputil(TN) = opimputil(T1) ∪ opimputil(T2)
  donc opimputil(T1) ⊆ opimputil(TN)
- opdefutil(TN) = opdefutil(T1) ∩ opdefutil(T2)
  donc opdefutil(T1) ⊇ opdefutil(TN)
- invar(TN) = invar(T1) ou invar(T2)
  donc invar(T1) impl invar(TN)

donc T1 est issu de TN et T2 aussi.

2.- Le noyau de T1 et de T2 est le plus petit type dont T1 et T2 soient issus



Démonstration : Notons TN le noyau de T1 et de T2 : TN = noyau(T1,T2)

T1 et T2 sont issus de T0 . TN l'est-il aussi ?

- expr(T0) ≡ expr(T1) et expr(T0) ≡ expr(T2)
  donc expr(T0) ≡ expr(TN)
- opimputil(T1) ⊆ opimputil(T0)
  opimputil(T2) ⊆ opimputil(T0)
  donc opimputil(TN) ⊆ opimputil(T0)
- opdefutil(T1) ⊇ opdefutil(T0)
  opdefutil(T2) ⊇ opdefutil(T0)
  donc opdefutil(TN) ⊇ opdefutil(T0)
- invar(T1) impl invar(T0)
  invar(T2) impl invar(T0)
  donc invar(TN) impl invar(T0)

3.- Si T2 est issu de T1 alors leur noyau est équivalent à T1.

T1 ⇝ T2 ⇒ T1 ≡ noyau(T1,T2)

Démonstration : Notons TN le noyau de T1 et de T2 TN = noyau(T1,T2)

- expr(TN) ≡ expr(T1)
- opimputil(TN) = opimputil(T1) ∪ opimputil(T2)
  or opimputil(T2) ⊆ opimputil(T1)
  donc opimputil(TN) = opimputil(T1)
- opdefutil(TN) = opdefutil(T1) ∩ opdefutil(T2)
  or opdefutil(T2) ⊇ opdefutil(T1)
  donc opdefutil(TN) = opdefutil(T1)
- invar(TN) = invar(T1) ou invar(T2)
  or invar(T2) ⇒ invar(T1)
  donc invar(TN) = invar(T1)

Définition de la fusion de deux types

Le type fusion de T1 et T2, noté | TF = fusion(T1,T2) | est défini par :

- l'expression de TF est celle de T1
- les opérations importées utilisables de TF sont celles appartenant à l'intersection des ensembles d'opérations importées utilisables de T1 et de T2.
- les opérations définies utilisables de TF sont l'union de celles de T1 et de celles de T2.
- l'invariant de TF est la conjonction de ceux de T1 et de T2.

TF = fusion(T1,T2)	
expr(TF)	= expr(T1)
opimputil(TF)	= opimputil(T1)∩ opimputil(T2)
opdefutil(TF)	= opdefutil(T1) ∪ opdefutil(T2)
invar(TF)	= invar(T1) et invar(T2)

Remarque : La fusion de deux types T1 et T2 "est plus définie" que T1 et que T2. Il possède moins d'opérations importées, plus d'opérations définies et son invariant étant plus restrictif que ceux de T1 et de T2, il possède moins d'objets.

Exemple : Reprenons l'univers composé des types des suites croissantes et décroissantes :

TYPE fusion(SCROISLIM,SDECROISLIM) = SUITE[ENTIER]	
invariant : infeg(2,taille(sucrde)) et infeg(taille(sucrde),10)	
opérations utilisables importées : toutes sauf ajout,ajoutrg et modif définies : ajcr,ajdcr, peutonaj	déclaration de variable sucrde : fusion(SCROISLIM, SDECROISLIM)

Ainsi définie, la fusion de deux types possède bien les propriétés souhaitées.

Propriétés :

1.- TF = fusion(T1,T2) est issu de T1 et de T2.

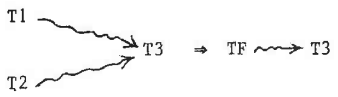
Démonstration :

Démontrons que T est issu de T1

- expr(TF) ≡ expr(T1)
- opimputil(TF) = opimputil(T1) ∩ opimputil(T2)
  donc opimputil(TF) ⊆ opimputil(T1)
- opdefutil(TF) = opdefutil(T1) ∪ opdefutil(T2)
  donc opdefutil(TF) ⊇ opdefutil(T1)
- invar(TF) = invar(T1) et invar(T2)
  donc invar(TF) impl invar(T1)

donc TF est issu de T1 et par symétrie de T2.

2.- Le type fusion de T1 et de T2 est le plus grand type issu de T1 et de T2 (pour la relation "est issu")



Démonstration :

T3 est issu de T1 et de T2

- expr(T3) ≡ expr(T1) donc expr(T3) ≡ expr(TF)
- opimputil(T3) ⊆ opimputil(T1)
  opimputil(T3) ⊆ opimputil(T2)
  donc opimputil(T3) ⊆ opimputil(T1) ∩ opimputil(T2)
  opimputil(T3) ⊆ opimputil(TF)
- opdefutil(T3) ⊇ opdefutil(T1)
  opdefutil(T3) ⊇ opdefutil(T2)
  donc opdefutil(T3) ⊇ opdefutil(T1) ∪ opdefutil(T2)
  opdefutil(T3) ⊇ opdefutil(TF)

- invar(T3) impl invar(T1)
  invar(T3) impl invar(T2)
  donc invar(T3) impl (invar(T1) et invar(T2))
  invar(T3) impl invar(TF)

3.- Si T2 est issu de T1 alors leut type fusion est équivalent à T2 :

$$T1 \rightsquigarrow T2 \Rightarrow T2 \equiv fusion(T1,T2)$$

Démonstration :

TF = fusion(T1,T2)
- expr(T0) = expr(T1) ≡ expr(T2) donc expr(TF) ≡ expr(T2)
- opimputil(T2) ⊆ opimputil(T1)
  donc opimputil(T1) ∩ opimputil(T2) = optimputil(T2)
- opdefutil(T2) ⊆ opdefutil(T1)
  donc opdefutil(T1) ∪ opdefutil(T2) = opdefutil(T2)
- invar(T2) impl invar(T1)
  donc invar(t2) et invar(T1) = invar(T2)

donc TF ≡ T2

Conclusion

Nous avons défini un langage permettant de spécifier un type de manière progressive : chaque étape permet de le préciser ou particulariser un peu plus. Ce langage facilite l'utilisation de bibliothèques de types. En effet nous avons défini des critères de comparaison de types très simples à mettre en oeuvre. Rechercher un type dans une bibliothèque c'est à partir d'une description "succinte" du type désiré (à savoir une expression dont on importe toutes les opérations, quelques opérations définies et un invariant), rechercher parmi un ensemble de types ceux qui sont issus de cette description. Si la recherche s'avère vaine on peut l'élargir aux types parents voire uniquement de structure équivalente à la description. Si seulement alors la recherche est fructueuse, on pourra fusionner le type trouvé à la description du type souhaité.

TRANSFORMATIONS

CHAPITRE III : TRANSFORMATIONS

Jusqu'à présent nous nous sommes préoccupés de construire une spécification, sans songer à son efficacité.

L'étape de construction doit être suivie d'une étape de transformation. Celle-ci doit permettre d'optimiser une spécification et de la rendre plus apte à évoluer.

Nous définissons un ensemble de transformations dépendant des constructeurs de types choisis,ainsi que des critères d'analyse d'une spécification. Un schéma de mécanisme de transformation intègre ces différents outils. Nous l'appliquons sur un exemple : la spécification du système SPES-TYPE d'aide à la construction d'univers de types.

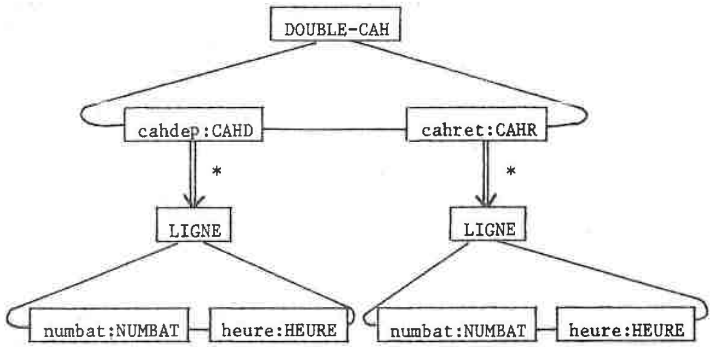
1.- TRANSFORMATION D'UN TYPE

Après avoir défini la notion de transformation et donné un schéma de présentation, nous définissons un ensemble de transformations correspondant aux constructeurs de types introduits. Nous étudions des transformations de structure, de généralisation, de simplification et de développement.

1.1.- Définition de la notion de transformation

1.1.1.- Pourquoi transformer ?

Nous avons étudié comment construire et présenter en termes de types abstraits, la spécification d'un problème. Nous allons étudier les inconvénients liés à l'approche descendante et au choix d'une famille de constructeurs de types pour structurer les données. Pour cela nous allons compléter l'exemple d'univers étudié au paragraphe II.3.1., du loueur de bateaux. Nous y avons présenté l'univers u_o composé des types structurés de la manière suivante :



Les opérations définies utilisables :

- du type DOUBLE-CAH des doubles cahiers sont departb et retourb.
- du type CAHD du cahier des départs sont pasparti et derdep.
- du type CAHR du cahier des retours sont derret.
- du type LIGNE des éléments des cahiers est reldepret

Le type DOUBLE-CAH est muni des opérations définies cachées rentré et pasparti.

Ces structures de données, de par la construction de la spécification, sont adaptées aux fonctionnalités departb et retourb. Pour définir une nouvelle fonctionnalité, nous devons utiliser la structure déjà définie. Celle-ci ne sera alors pas forcément la plus adéquate aux nouvelles opérations définies. Pour illustrer ceci nous enrichissons l'univers u_o d'une nouvelle fonctionnalité : l'opération batloués permettant de connaître l'ensemble des bateaux loués à une heure donnée.

Appliquons le méta-algorithme pour spécifier la fonction batloués :

Type du résultat :

[ENS-BAT = ENS[NUMBAT]

Le résultat est un ensemble de (numeros de) bateaux. Les données ne sont pas encore identifiées sauf l'heure de la période, de type HEURE.

Définition de batloués : ensemble des bateaux loués à une heure donnée.

Il s'agit donc du sous-ensemble des locations possèdant la propriété : "loué à une heure donnée".

Type des données :

[ENS-LOC = ENS[LOC]

Les données sont de type ensemble de locations

[batloué : ENS-BAT ← ENS-LOC, HEURE

[batloué(ensloc,h) = itere$_{id,louéheure(.,h)}$ (ensloc)

où louéheure(.,h) est un prédicat indiquant à partir d'une location si le bateau était loué à l'heure h.

ensloc est une variable de type ENS-LOC.

Définition de louéheure

[louéheure : BOOL ← LOC,HEURE

[type des données :

LOC = PC[bat:NUMBAT, dep:HEURE, ret : HEURE]

[louéheure(loc,h) = infeg(dep(loc),h) et infeg(h,ret(loc))

Nous avons introduit une donnée intermédiaire : ensloc l'ensemble des locations. Il faut à présent la définir en fonction des données du problème, à savoir le double-cahier.

[ensloc = calculloc(dcah)

Définition de calculloc

[calculloc : ENS-LOC ← DOUBLE-CAH

calculloc calcule la liste des locations à partir du double cahier dcah. Pour cela il associe à chaque départ (du cahier des départs) un retour (du cahier des retours). Appelons assdepret l'opération réalisant cette association. Puis il faut construire à l'aide de l'opération consloc, l'élément de type location à partir des enregistrements de départ et de retour.

[calculloc(dcah) = iter$_{consloc(.,assdepret(cahret(dcah),.)),vrai}$(cahdep(dcah))

[Définition de assdepret : LIGNE ← CAHR,LIGNE

C'est le dernier enregistrement entré dans le cahier des retours pouvant correspondre à la même location que la ligne donnée. Il s'agit donc de la relation reldepret déjà étudiée

[assdepret(cret,ℓ) = tête(iter$_{id,reldepret(ℓ,.)}$(cret))

[Définition de consloc : LOC ← LIGNE,LIGNE

[consloc(ℓd,ℓr) = c(numbat(ℓd),heure(ℓd),heure(ℓr))

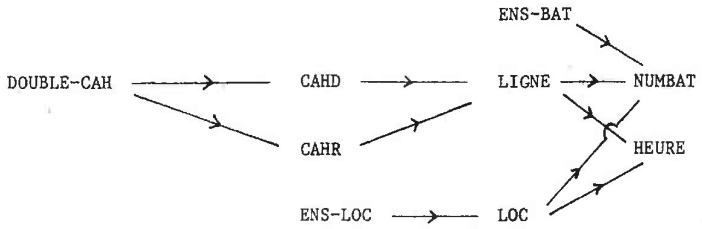
Remarque : Si un bateau est parti et non rentré, l'opération assdepret donnera comme résultat la constante indef de type LIGNE et heure(indef) = indef. Donc dans ce cas la location aura une composante indéfinie.

Le spécifieur n'a pas eu à se préoccuper de ce cas. Les opérations de

base des constructeurs étant définies de manière complète, les opérations définies par composition le sont aussi.

Nous pouvons associer chacune des opérations définies à un type en appliquant la règle du § II.3.1.

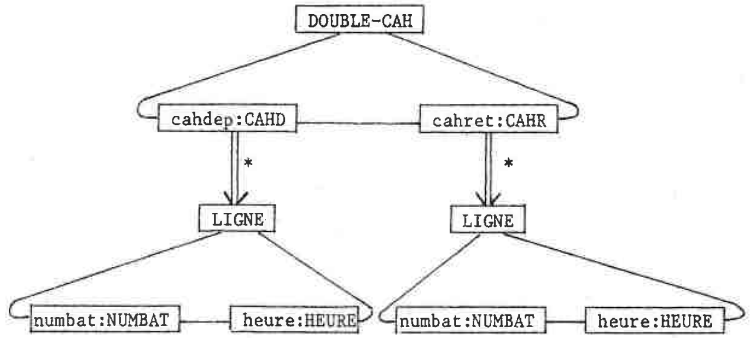
Les types sont préordonnés suivant le graphe ci-dessous :

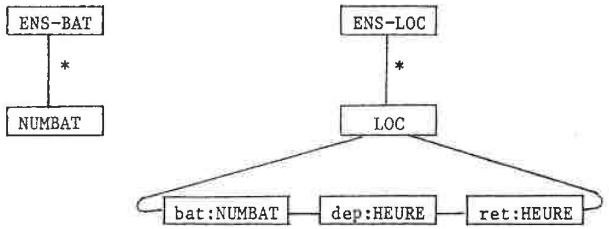


batloués : ENS-BAT ← ENS-LOC
est associée à ENS-LOC par la règle (2)
louéheure : BOOL ← LOC,HEURE est associée à LOC par la règle (1)
calculloc : ENS-LOC ← DOUBLE-CAH est associée à DOUBLE-CAH par la règle (2)
assdepret : LIGNE ← CAHR,LIGNE est associée à CAHR par la règle (1)
consloc : LOC ← LIGNE,LIGNE est associée à LIGNE par la règle (2)

Etude de la spécification de l'univers u_1 obtenu par enrichissement de l'univers u_0 :

L'univers u_1 est composé des types structurés de la manière suivante :





opérations définies utilisables de :

DOUBLE-CAH	:	departb	: DOUBLE-CAH ← DOUBLE-CAH,LIGNE
		retourb	: DOUBLE-CAH ← DOUBLE-CAH,LIGNE
		calculloc	: ENS-LOC ← DOUBLE-CAH
CAHD	:	pasparti	: BOOL ← CAHD,NUMBAT
		derdep	: LIGNE ← CAHD,NUMBAT
CAHR	:	derret	: LIGNE ← CAHR,NUMBAT
		assdepret	: LIGNE ← CAHR,LIGNE
LIGNE	:	reldepret	: BOOL ← LIGNE,LIGNE
		consloc	: LOC ← LIGNE,LIGNE
ENS-LOC	:	batloués	: ENS-BAT ← ENS-LOC
LOC	:	louéheure	: BOOL ← LOC,HEURE

Remarques :

- La démarche descendante suivie pour la définition de l'opération batloués nous a conduit à introduire une donnée intermédiaire : l'ensemble des locations. Cette donnée est redondante avec la donnée du double-cahier puisqu'une opération (calculloc) permet de calculer l'ensemble des locations uniquement à partir du double-cahier.

[Il y a donc redondance de structure.

- Lors du déroulement du méta-algorithme, la définition d'une opération nécessite la définition de la structure du type du résultat. C'est ainsi que nous avons défini le type CAHD, du cahier des départs. Puis par application de la règle (3) d'association d'une opération à un type, nous avons introduit la structure du double-cahier. Cette structure n'est pourtant pas optimale pour les opérations départb et retourb. En effet, avant d'enregistrer un départ ou un retour il faut vérifier qu'il n'y a pas d'erreur, à savoir si le bateau n'est pas déjà parti (s'il s'agit d'un départ), et s'il n'est pas déjà rentré (s'il s'agit d'un retour).
  Pour répondre à ces interrogations, il est nécessaire de rechercher la dernière location de ce bateau.

[Il y a donc inadéquation d'une structure de données trop précocement définie, aux opérations définies postérieurement.

- Lors de la spécification de nouvelles fonctionnalités enrichissant l'univers u_o, suivant la démarche proposée par le méta-algorithme, nous n'avons pas eu l'occasion de remettre en question les premiers choix faits. Cet enrichissement s'est "greffé" sur un univers et n'a pu que s'en accomoder.

[Il y a donc manque d'évolutivité.

Nous pouvons résumer les inconvénients liés à l'approche descendante et à la description en parallèle des fonctions du système et des structures de données, par

- le risque d'inadéquation aux opérations définies par la suite des structures de données choisies trop précocement,

- l'introduction de structures redondantes,
- le manque d'évolutivité.

Pour pallier à ces inconvénients, nous allons définir des transformations de types. Nous distinguons les transformations de structure, de généralisation, de simplification et de développement de structures.

1.1.2.- Définition et présentation des transformations

Les transformations de types que nous allons introduire permettent de modifier la structure d'un type tout en préservant ses propriétés.

De manière précise, une transformation est une représentation faible paramétrée. Elle décrit la transformation des objets d'un type source et de ses opérations en termes d'objets et d'opérations d'un type cible. Le type source est issu du type défini par l'expression source. Il n'est muni que des opérations qui peuvent être transformées. En effet, les structures n'étant pas toutes équivalentes (ou isomorphes), il est nécessaire de prendre des précautions lors de la transformation d'un type. Etudions à l'aide d'un exemple les problèmes qui se posent.

Nous voulons définir des transformations permettant de modifier la structure d'un type donné:par exemple celle permettant d'obtenir un ensemble à partir d'une suite.

Les deux structures ont chacune leurs caractéristiques :

- une suite est un multi-ensemble d'éléments de même type, ordonnés par leur ordre d'adjonction dans la suite. Remarquons qu'une suite peut comporter deux occurrences d'un même objet,
- dans un ensemble, les éléments ne sont pas ordonnés et un élément n'a d'une seule occurrence.

Lors de la transformation d'une suite en un ensemble, il y a donc

- oubli de l'ordre des éléments et
- conservation d'une seule des occurrences des éléments appartenant à la suite.

Pour que la transformation soit effectivement une représentation faible paramétrée, il faut qu'aucune des opérations du type source de structure de

suite, n'utilise ni l'ordre des éléments ni le fait qu'ils puissent avoir plusieurs occurrences.

Les opérations du constructeur SUITE utilisant l'ordre sont les suivantes :

- les modificateurs sup, ajoutrg, modif
- les observateurs acc, rang, tête et éq.

L'opération sup ne supprime qu'une occurrence d'un élément. L'opération taille compte toutes les occurrences de tous les éléments d'une suite.

Pour pouvoir transformer en un ensemble, une suite, et ceci sans perte d'information, il faut qu'aucune des opérations ci-dessus, ne soit utilisable sur la suite.

> Le type source d'une transformation est issu de celui défini par l'expression source, par restriction de l'ensemble des opérations importées utilisables.

Parfois il n'est pas suffisant de restreindre l'ensemble des opérations. En effet les objets du type source peuvent devoir vérifier une propriété ou invariant.

> Le type source d'une transformation peut comporter un invariant restreignant l'ensemble des objets transformables.

Examinons le problème de la représentation de l'égalité. Trois cas peuvent se produire :

- l'égalité de l'expression du type source ne peut être transformé.

  C'est le cas de la transformation d'une suite en un ensemble. Ceci signifie que des objets différents pour le prédicat d'égalité du type source seront transformés en des objets égaux pour le prédicat d'égalité du type cible.

- la transformée de l'égalité du type source est l'égalité du type cible. Ceci signifie qu'à chaque classe d'objets du type source est associée une et une seule classe d'objets du type cible.

- l'égalité du type source est transformée en une relation d'équivalence moins fine que l'égalité du type cible. C'est le cas pour la transformation d'un ensemble en une suite. Ceci signifie que certains objets équivalents pour le prédicat d'égalité du type source, seront transformés en des objets différents pour le prédicat d'égalité du type cible. Ces transformations sont les représentations faibles de [FIN, 79].

Définition :

Une transformation r est une représentation faible paramétrée d'un type (paramétré) source TS vers un type (paramétré) cible TC.

Une transformation r est définie par la donnée
- du type source TS
- du type cible TC
- de l'association à chaque opération du type source, d'une opération du type cible

Le type source TS
- est issu de l'expression source expr(TS)
- par restriction des opérations importées utilisables :
  oputil(TS) $\subset$ oputil(expr(TS))
- par la définition d'un invariant invar(TS).

Le type cible TC
- est issu de l'expression cible expr(TC)
- par adjonction d'opérations définies utilisables, transformées des opérations utilisables de TS :
  oputil(TC) = oputil(expr(TC)) $\cup$ r(oputil(TS)).

L'association entre les opérations du type source et celles du type cible se fait implicitement :

les opérations du type source et du type cible associées ont le même nom. Les exceptions sont précisées dans la définition du type cible.

Exemple : Dans la transformation tts d'une table en une suite de couples, le prédicat d'égalité du type source eq n'est pas transformé

en le prédicat d'égalité eq du type cible mais en une relation d'équivalence définie du type cible eqt = tts(eq).

Un des objectifs recherchés lors de la transformation d'un type, est une meilleure adéquation de ses opérations à sa structure, c'est-à-dire une optimisation de la définition de ses opérations.

Prenons l'exemple d'un type dont la structure est un ensemble de couples. Supposons que les opérations d'accès définies sur ce type sont des recherches associatives d'un élément ayant pour première composante une valeur donnée. Ce type n'est pas adapté à sa structure. L'accès aux éléments sera optimisé par la transformation en structure de table ayant pour indice la première composante du couple. Nous dirons que l'opération d'accès aux éléments a bénéficié de la transformation. Dans la transformation inverse (d'une structure de table en une structure d'ensemble de couples) l'opération d'accès aux éléments d'une table pâtira (en ce sens qu'elle nécessitera d'une itération supplémentaire lors de son implantation). L'utilisation par un type d'opérations bénéficiant d'une transformation, ou au contraire en pâtissant sont des critères d'application d'une telle transformation sur ce type. Nous les précisons pour chaque transformation.

Schéma de présentation d'une transformation :

Les transformations sont présentées selon le schéma général suivant :

Transformation R

du type source	en	le type cible
graphe de l'expression du type source	$\xrightarrow{R}$	graphe de l'expression du type cible

explication intuitive du mécanisme de transformation.

critères d'application :

- utilisation de certaines opérations
- non utilisation de certaines autres opérations

TYPE SOURCE-R = expr(TS)	
invar(TS)	
oputil(TS)	variables

TYPE CIBLE-R = expr(TC)	
invar(TC)	
oputil(TC)	variables
définition des opérations définies utilisables	

1.2.- Etude de quelques transformations

Parmi les transformations que nous allons présenter, nous distinguons :

- les transformations de structure,
- les généralisations,
- les simplifications et
- les développements.

Nous présentons l'ensemble des transformations correspondant aux quatre constructeurs de types définis. Chaque constructeur correspond à une structure ayant les propriétés suivantes :

- structure de suite : association d'un nombre variable d'éléments de même type, ordonnés. Un élément peut avoir plusieurs occurrences. L'accès aux éléments est séquentiel suivant l'ordre de la suite.
- structure d'ensemble : association d'un nombre variable d'éléments de même type, non ordonnés. Un élément ne peut avoir qu'une seule occurrence. L'accès aux éléments se fait à l'aide du prédicat d'appartenance.

- structure de table : fonction définie sur un ensemble d'indices, éléments d'un type donné et à valeur dans un ensemble d'éléments d'un autre type. La fonction associe à chaque indice, un élément. Le nombre d'indices est variable. Ni les indices ni les éléments ne sont ordonnés. L'accès aux éléments se fait directement par leur indice.

- structure de produit cartésien : association d'un nombre fixé d'éléments de types différents, non ordonnés, composés en champs nommés. L'accès aux éléments se fait directement par le nom de leur champ.

Nous présentons les transformations suivant le schéma du paragraphe III.1.1.2. Cependant pour ne pas surcharger, nous ne détaillons que la définition de l'image du constructeur du type source. Nous ne donnons la définition de toutes les opérations définies du type cible que pour les transformations de structure tse, tes, tsp et tps.

Rappelons qu'un type dont on ne précise pas d'opérations

. importées, possède celles du type de son expression,

. définies , possède celles du type de son expression ; si celui-ci est un constructeur de types instancié, alors le type n'hérite d'aucune opération définie.

Rappelons d'autre part que le morphisme de signatures associe à chaque opération du type source, l'opération du type cible de même nom.

1.2.1.- Les transformations de structure

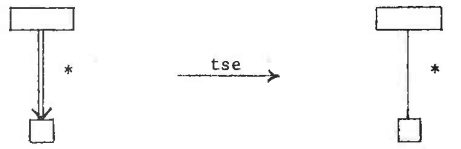
Les transformations de structure transforment un type d'une structure donnée en un type d'une structure différente. Ces transformations ont pour but de réduire la complexité des définitions des opérations d'un type. Disposant de quatre structures, nous pouvons définir 12 transformations de structure.

Tableau des transformations de structure :

	SUITE	ENSEMBLE	TABLE	PRODUIT CARTESIEN
SUITE		S[P] ↓ tse E[P]	S[PC[P,Q]] ↓ tst T[P] de S[Q]	S[PC[P,Q]] ↓ tsp PC[S[P],S[Q]]
ENSEMBLE	E[P] ↓ tes S[P]		E[PC[P,Q]] ↓ tet T[P] de E[Q]	E[PC[P,Q]] ↓ tep PC[E[P],E[Q]]
TABLE	T[P] de Q ↓ tts S[PC[P,Q]]	T[P] de Q ↓ tte E[PC[P,Q]]		T[P] de PC[Q,R] ↓ ttp PC[T[P] de Q, T[P] de R]
PRODUIT CARTESIEN	PC[S[P],S[Q]] ↓ tps S[PC[P,Q]]	PC[E[P],E[Q]] ↓ tpe E[PC[P,Q]]	PC[T[P] de Q T[P] de R] ↓ tpt T[P] de PC[P,Q]	

Transformation tse

d'une suite en un ensemble



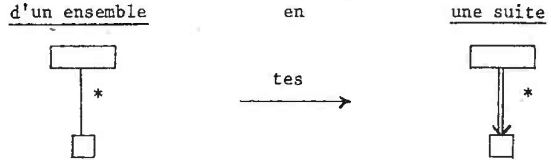
La transformation d'une suite en un ensemble se fait en oubliant l'ordre des éléments de la suite et en ne gardant qu'une occurrence de chaque élément.

Critère d'application : utilisation du prédicat d'appartenance app

TYPE SOURCE-TSE = S[P]
opérations utilisables importées : constructeurs : svide,ajout modificateurs : singl,supelt,conc observateurs : app,svide?,tête∘singl

TYPE CIBLE-TSE = E[P]	
opérations utilisables définies : modificateurs : svide, ajout,singl,supelt,conc observateurs : app,svide?,tête∘singl	variables ep,ep':CIBLE-TSE p : P
définitions des modificateurs svide : CIBLE-TSE ← svide = evide ajout : CIBLE-TSE ← CIBLE-TSE,P ajout(ep,p) = adj(ep,p) singl : CIBLE-TSE ← P singl(p) = single(p) supelt : CIBLE-TSE ← CIBLE-TSE,P supelt(ep,p) = supe(ep,p) conc : CIBLE-TSE ← CIBLE-TSE,CIBLE-TSE conc(ep,ep') = union(ep,ep')	
définition des observateurs app : BOOL ← CIBLE-TSE,P app(ep,p) = appe(ep,p) svide? : BOOL ← CIBLE-TSE svide?(ep) = evide?(ep) tête∘singl : P ← CIBLE-TSE tête∘singl(ep) = elemsingl(ep)	

Transformation tes



La transformation d'un ensemble en une suite se fait en imposant l'ordre de parcours de l'ensemble (arbitraire) comme ordre de la suite.

Critères d'application : non utilisation des opérations ensemblistes union, inclu et surtout intersec.

TYPE SOURCE-TES = E[P]

TYPE CIBLE-TES = S[P]	
opérations utilisables définies : modificateurs : evide,adj,single,supe, union,intersec observateurs : taillee,appe,evide?, inclu,eqe(tes(eq)=eqe), elemsingl opérateurs : $itere_{fct,prop}$,$sommee_{fcte}$	Variables sp,sp' : CIBLE-TES p,p' : P
définition des modificateurs evide : CIBLE-TES ← evide = svide adj : CIBLE-TES ← CIBLE-TES,P adj(sp,p) = si app(sp,p) alors sp sinon ajout(sp,p) single : CIBLE-TES ← P single(p) = singl(p) supe : CIBLE-TES ← CIBLE-TES,P supe(sp,p) = supelt(sp,p) union : CIBLE-TES ← CIBLE-TES,CIBLE-TES union(sp,sp') = conc(sp,sp') intersec : CIBLE-TES ← CIBLE-TES,CIBLE-TES intersec(sp,sp') = $iter_{id,app(sp',.)}(sp)$	

définition des observateurs

elemsingl : P ← CIBLE-TES

elemsingl(sp) = si eq(un,taille(sp)) alors tête(sp)
sinon indef

taillee : ENT ← CIBLE-TES

taillee(svide) = zéro

taillee(ajoutfn(sp,p)) = si app(sp,p) alors taillee(sp)
sinon suc(taillee(sp))

taillee(indef) = indef

appe : BOOL ← CIBLE-TES,P

appe(sp,p) = app(sp,p)

evide? : BOOL ← CIBLE-TES

evide?(sp) = svide?(sp)

eqe : BOOL ← CIBLE-TES,CIBLE-TES

eqe(evide,sp) = svide?(sp)

eqe(ajoutfn(sp,p),sp') =
appe(sp',p) et eqe(supelt(sp,p),supelt(sp',p))

eqe(indef,sp) = eq(sp,indef)

inclu : BOOL ← CIBLE-TES,CIBLE-TES

inclu(sp,svide) = non(eq(sp,indef))

inclu(sp,ajoutfn(sp',p')) = appe(sp,p') et inclu(sp,sp')

inclu(sp,indef) = eq(sp,indef)

définition des opérateurs pour fct:R←P, prop:BOOL←P,fcte:ENT←P

$itere_{fct,prop}$: E[R] ← CIBLE-TES

$itere_{fct,prop}$(svide) = evide

$itere_{fct,prop}$(ajoutfn(sp,p)) =
si prop(p) et non(app(sp,p))
alors adj($itere_{fct,prop}$(sp),fct(p))
sinon $itere_{fct,prop}$(sp)

$itere_{fct,prop}$(indef) = indef

$sommee_{fcte}$: ENT ← CIBLE-TES

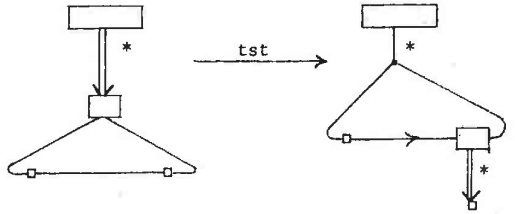
$sommee_{fcte}$(svide) = zéro

$sommee_{fcte}$(ajoutfn(sp,p)) =
si app(sp,p) alors $sommee_{fcte}$(sp)
sinon plus($sommee_{fcte}$(sp),fcte(p))

$sommee_{fcte}$(indef) = indef

Transformation tst

d'une suite de couples en une table de suites



La transformation d'une suite de couples en une table associant à un indice une suite, se fait en regroupant en sous-suites les éléments de la suite de même composante clé. Pour cela il faut choisir une des composantes du produit cartésien comme indice de la table (composante clé). L'accès aux sous-suites se fera directement grâce à cette composante. L'ordre général de la suite source ainsi que les noms des composantes du produit cartésien seront oubliés.

Critères d'application : utilisation de $iter_{fct,eq(clé(.),p)}$

non utilisation de conc

TYPE SOURCE-TST = S[PC[un:P, deux:Q]]

opérations utilisables

importées : constructeurs : svide,ajout
modificateurs : singl, supelt, conc
observateurs : app, svide?, taille
opérateurs : $somme_{fcte}$

TYPE CIBLE-TST = T[P] de S[Q]

opérations utilisables

définies : modificateurs : svide,ajout,singl,
supelt,conc
observateurs : app,svide?,taille
opérateurs : $somme_{fcte}$

variables

tpsq:CIBLE-TST
pq:PC[un:P
deux:Q]

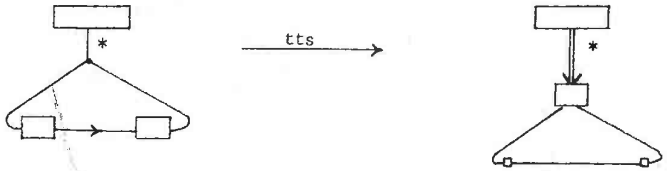
définition du modificateur ajout :

ajout : CIBLE-TST ← CIBLE-TST,PC [un:P,deux:Q]

ajout(tpsq,pq) = si appt(tpsq,un(pq))
alors mod(tpsq,un(pq),ajout(acct(tpsq,un(pq)),deux(pq)))
sinon insert(tpsq,un(pq),singl(deux(pq)))

Transformation tts

d'une table en une suite de couples



La transformation d'une table en une suite de couples se fait en imposant l'ordre d'insertion des éléments dans la table comme ordre de la suite. Les couples sont composés d'un indice et de l'élément qui lui était associé dans la table. Il est nécessaire de choisir les noms des composantes du produit cartésien. L'accès aux éléments (direct dans la table) se fera séquentiellement. Les éléments des suites obtenues par transformation ont tous une composante clé (ou indice) différente.

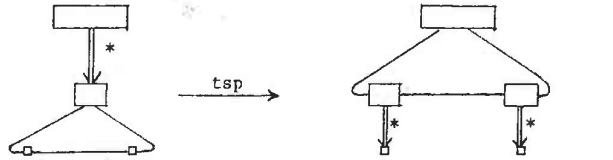
Critères d'application : non utilisation de l'accès direct acct
signification de l'ordre d'insertion des éléments

TYPE SOURCE-TTS = T[P] de Q

TYPE CIBLE-TTS = S[PC[un:P, deux:Q]]	
opérations utilisables définies : modificateurs : tvide,insert,supt,mod observateurs : acct,taillet,appt,tvide? eqt(tts(eq)=eqt) opérateurs : $iter_{fct,prop}^{sommet_{fcte}}$	variables spq:CIBLE-TTS pq:PC[un:P,deux:Q] p:P q:Q
définition du modificateur insert insert : CIBLE-TTS ← CIBLE-TTS,P,Q insert(svide,p,q) = ajout(svide,c(p,q)) insert(ajout(spq,pq),p,q) = si eq(un(pq),p) alors insert(spq,p,q) sinon ajout(insert(spq,p,q),pq) insert(indef,p,q) = indef	

Transformation tsp

d'une suite de couples en un produit cartésien de suites



La transformation d'une suite de couples en un produit cartésien de suites, se fait en oubliant les relations entre les éléments formant les produits cartésiens du type source. Seules resteront deux suites formées l'une par les premières composantes des couples et l'autre par les deuxièmes. Si les noms des composantes des couples du type source seront oubliés, il faut cependant choisir les noms des suites formant le produit cartésien du type cible.

Critères d'application :

utilisation d'opérations comportant dans leur profil les types des suites du type cible.

non utilisation des relations existant entre les éléments des couples du type source (par exemple la recherche de la suite formée des deuxièmes composantes des éléments ayant comme première composante une valeur donnée : $iter_{deux,éq(un(.),p)}$).

TYPE SOURCE-TSP = S[PC[un:P,deux:Q]]
opérations utilisables importées : constructeurs : svide,ajout modificateurs : singl,sup,supelt,ajoutrg,modif,conc observateurs : acc,tête,rang,taille,app,svide?,eq opérateurs : $iter_{fct,prop,somme_{fcte}}$ pour fct : R ← PC[P,Q] fct(pq) = fct'(f1(un(pq)),f2(deux(pq))) prop : BOOL ← PC[P,Q] prop(pq) = prop'(f1(un(pq)), f2(deux(pq))) fcte : ENT ← PC[P,Q],fcte(pq) = fcte'(f1(un(pq)), f2(deux(pq)) où fct' : R ← P',Q' prop' : BOOL ← P',Q' fcte' : ENT ← P',Q' f1 : P' ← P f2 : Q' ← Q

TYPE CIBLE-TST = PC[sun:S[P],sdeux:S[Q]]

opérations utilisables

définies : modificateurs : svide,ajout,singl
sup,supelt,ajoutrg,modif,conc
observateurs : acc,tête,rang,taille
app,svide?,eqs(tsp(eq)=eqs)
opérateurs : $iter_{fct,prop}$,$somme_{fcte}$

variables

spsq, spsq' : CIBLE-TSP
pq : PC[un:P,deux:Q]
e : ENT

définition des modificateurs

svide : CIBLE-TSP ←
svide = c(svide,svide)

ajout : CIBLE-TSP ← CIBLE-TSP,PC[un:P,deux:Q]
ajout(spsq,pq) = c(ajout(sun(spsq),un(pq)),
ajout(sdeux(spsq),deux(pq)))

singl : CIBLE-TSP ← PC[un:P,deux:Q]
singl(pq) = c(singl(un(pq)),singl(deux(pq)))

sup : CIBLE-TSP ← CIBLE-TSP,ENT
sup(spsq,e) = c(sup(sun(spsq),e),sup(sdeux(spsq),e))

supelt : CIBLE-TSP ← CIBLE-TSP,PC[un:P,deux:Q]
supelt(spsq,pq) = c(supelt(sun(spsq),un(pq)),
supelt(sdeux(spsq),deux(pq)))

ajoutrg : CIBLE-TSP ← CIBLE-TSP,ENT,PC[un:P,deux:Q]
ajoutrg(spsq,e,pq) = c(ajoutrg(sun(spsq),e,un(pq)),
ajoutrg(sdeux(spsq),e,deux(pq)))

modif : CIBLE-TSP ← CIBLE-TSP,ENT,PC[un:P,deux:Q]
modif(spsq,e,pq) = c(modif(sun(spsq),e,un(pq)),
modif(sdeux(spsq),e,deux(pq)))

conc : CIBLE-TSP ← CIBLE-TSP,CIBLE-TSP
conc(spsq,spsq') = c(conc(sun(spsq),sun(spsq')),
conc(sdeux(spsq),sdeux(spsq')))

définition des observateurs

acc : PC[un:P,deux:Q] ← CIBLE-TSP,ENT
acc(spsq,e) = c(acc(sun(spsq),e),acc(sdeux(spsq),e))

tête : PC[un:P,deux:Q] ← CIBLE-TSP
tête(spsq) = c(tête(sun(spsq)),tête(sdeux(spsq)))

rang : ENT ← CIBLE-TSP,PC[un:P,deux:Q]
rang(spsq,pq) =
si eq(rang(sun(spsq),un(pq)),rang(sdeux(spsq),deux(pq)))
alors rang(sun(spsq),un(pq)) sinon indef

taille : ENT ← CIBLE-TSP
taille(spsq) = si eq(taille(sun(spsq)),taille(sdeux(spsq)))
alors taille(sun(spsq))sinon indef

app : BOOL ← CIBLE-TSP,PC[un:P,deux:Q]
app(spsq,pq) = app(sun(spsq),un(pq)) et
app(sdeux(spsq),deux(pq))

svide? : BOOL ← CIBLE-TSP
svide?(spsq) = svide?(sun(spsq)) et svide?(sdeux(spsq))

eqs : BOOL ← CIBLE-TSP,CIBLE-TSP
eqs(spsq,spsq') = eq(spsq,spsq')

définition des opérateurs pour :

fct : R ← PC[un:P,deux:Q] fct(pq) = fct'(f1(un(pq)),f2(deux(qp)))

prop : BOOL ← PC[un:P,deux:Q] prop(pq) = prop'(f1(un(pq)),f2(deux(pq)))

fcte : ENT ← PC[un:P,deux:Q] fcte(pq) = fcte'(f1(un(pq)),f2(deux(pa)))

où fct' : R ← P',Q' prop' : BOOL ← P',Q' fcte : ENT ← P',Q'
f1 : P' ← P et f2 : Q' ← Q

$iter_{fct,prop}$: S[R] ← CIBLE-TSP

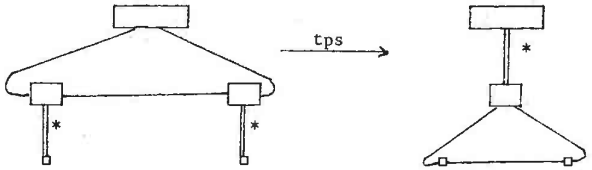
$iter_{fct,prop}(spsq) = iter2_{fct',prop'}(iter_{f1,vrai}(sun(spsq)), iter_{f2,vrai}(sdeux(spsq)))$

$somme_{fcte}$: ENT ← CIBLE-TSP

$somme_{fcte}(spsq) = somme2_{fcte'}(iter_{f1,vrai}(sun(spsq)), iter_{f2,vrai}(sdeux(spsq)))$

Transformation tps

d'un produit cartésien de suites en une suite de couples



La transformation d'un produit cartésien de suites en une suite de couples se fait en associant à chaque élément d'une des suites appelée suite guide, un élément de la deuxième vérifiant une relation donnée "rel". Si aucun élément de la deuxième suite ne peut être associé à un élément de la suite guide alors c'est l'élément indéfini "indef" qui lui sera associé. Ainsi chaque élément de la deuxième suite est associé à un (et un seul) élément de la suite guide. Celle-ci comportera donc au moins le même nombre d'éléments que la deuxième suite. L'ordre des éléments de la deuxième suite sera perdu. Il est nécessaire de donner des noms aux composantes des produits cartésiens du type cible tandis que ceux du type source seront perdus.

Critères d'application :

utilisation d'une relation entre chaque élément de la deuxième suite et un élément de la suite guide

non utilisation de la structure par suites.

TYPE SOURCE-TPS = PC[sun:SP,sdeux:SQ]	
invariant : $eq(svide,iter_{id,eq(indef,assqp(sun(spsq),.))}(sdeux(spsq)))$	
opérations utilisables importées constructeurs : c modificateurs : modifun,modifdeux observateurs : un,deux	variables spsq : SOURCE-TPS
lexique : l'invariant signifie que tout élément de la suite sdeux peut être associé à un élément de la suite guide sun.	

TYPE SP = S[P]	
opérations utilisables définie : assqp	variables sp : SP q : Q
définition assqp : P ← SP,Q $assqp(sp,q) = tête(iter_{id,rel(.,q)}(sp))$ où rel : BOOL ← P,Q	
lexique : assqp associe à un objet de type Q donné, un élément de la suite vérifiant la relation "rel".	

TYPE SQ = S[Q]	
opérations utilisables importées constructeurs : svide,ajout modificateurs : singl,supelt,conc observateurs : app,svide?,taille définie : asspq	variables sq : SQ p : P
définition asspq : Q ← SQ,P $asspq(sq,p) = tête(iter_{id,rel(p,.)}(sq))$ où rel : BOOL ← P,Q	
lexique : asspq associe à un objet de type P donné, un élément de la suite vérifiant la relation "rel".	

TYPE CIBLE-TPS = S[PC[un:P,deux:Q]]

opérations utilisables

définies : modificateurs : c,modifsun,modifsdeux
observateurs : sun,sdeux

variables

spq : CIBLE-TPS
sp : SP sq : SQ
p : P

définition des modificateurs

c : CIBLE-TPS ← SP,SQ
c(svide,svide) = svide
c(ajoutfn(sp,p),sq) =
= ajoutfn(c(sp,sup(sq,rang(sq,asspq(sq,p)))),
c(p,asspq(sq,p)))
c(indef,sq) = indef

modifsun : CIBLE-TPS ← CIBLE-TPS,SP
modifsun(spq,sp) = $c(sp,iter_{deux(.),vrai}(spq))$

modifsdeux : CIBLE-TPS ← CIBLE-TPS,SQ
modifsdeux(spq,sq) = $c(iter_{un(.),vrai}(spq),sq)$

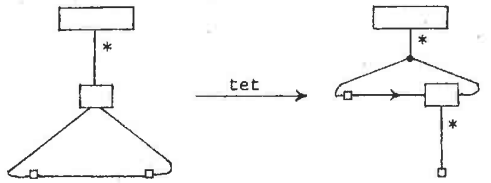
définition des observateurs

sun : SP ← CIBLE-TPS
sun(spq) = $iter_{un(.),vrai}(spq)$

sdeux : SQ ← CIBLE-TPS
sdeux(spq) = $iter_{deux(.),vrai}(spq)$

Transformation tet

d'un ensemble de couples en une table d'ensembles



La transformation d'un ensemble de couples en une table associant à un indice un ensemble, se fait en regroupant en sous-ensembles les éléments de l'ensemble de même composante clé. Pour cela il faut choisir une des composantes du produit cartésien comme indice de la table (composante clé). L'accès aux sous-ensembles se fera directement grâce à cette composante. Les noms des composantes du produit cartésien seront oubliés.

Critères d'application :

utilisation de $itere_{fct,eq(clé(.),p)}$

non utilisation des opérations ensemblistes (union,intersec et inclu).

TYPE SOURCE-TET = E[PC[un:P,deux:Q]]

TYPE CIBLE-TET = T[P] de E[Q]

opérations utilisables

définies : modificateurs : evide,adj,single,supe
union,intersec
observateurs : taillee,inclu,appe,
evide?,elemsingl,eqe
(tet(eq) = eqe)
observateurs : $itere_{fct,prop}$,$sommee_{fcte}$

variables

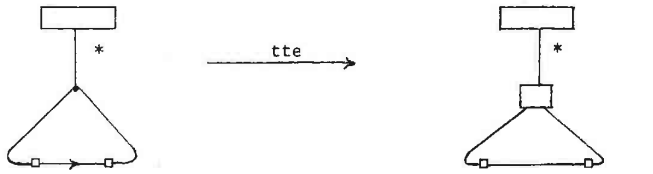
tpeq : CIBLE-TET
pq : PC[un:P,
deux:Q]

définition du modificateur adj :

adj : CIBLE-TET ← CIBLE-TET,PC[un:P,deux:Q]
adj(tpeq,pq) = si appt(tpeq,un(pq))
alors mod(tpeq,un(pq),adj(acct(tpeq,un(pq)),
deux(pq)))
sinon insert(tpeq,un(pq),single(deux(pq)))

Transformation tte

d'une table en un ensemble de couples



La transformation d'une table en un ensemble de couples se fait en composant les produits cartésiens d'un indice et de l'élément qui lui était associé dans la table. Il est nécessaire de choisir les noms des composantes du produit cartésien. L'accès aux éléments direct dans la table grâce à l'indice, se fera uniquement grâce au prédicat "appe". Les éléments des ensembles obtenus par transformation ont tous une composante clé (ou indice) différente.

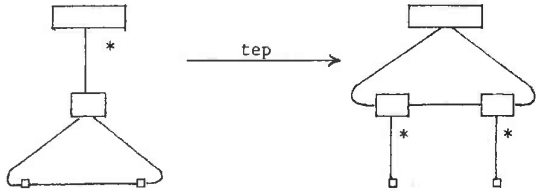
Critères d'application : nécessité d'utilisation d'opérations ensemblistes (union ou intersec)
non utilisation de l'accès direct acct

TYPE SOURCE-TTE = T[P] de Q

TYPE CIBLE-TTE = E[PC[un:P,deux:Q]]	
opérations utilisables définies : modificateurs : tvide,insert,supt,mod observateurs : acct,taillet,appt, tvide?,eqt (tte(eq) = eqt) opérateurs : $itert_{fct,prop'}$ $sommet_{fcte}$	variables epq : CIBLE-TTE pq : PC[un:P, deux:Q] p : P q : Q
définition du modificateur insert insert : CIBLE-TTE ← CIBLE-TTE,P,Q insert(evide,p,q) = adj(evide,c(p,q)) insert(adj(epq,pq),p,q) = si eq(un(pq),p) alors insert(epq,p,q) sinon adj(insert(epq,p,q),pq) insert(indef,p,q) = indef	

Transformation tep

d'un ensemble de couples en un produit cartésien d'ensembles



La transformation d'un ensemble de couples en un produit cartésien d'ensembles, se fait en oubliant les relations entre les éléments formant les couples du type source. Seuls resteront deux ensembles formés l'un par les premières composantes des produits cartésiens, l'autre par les deuxièmes. Si les noms des composantes du produit cartésien du type source seront oubliés, il faut cependant choisir les noms des ensembles formant le produit cartésien du type cible.

Critères d'application :

utilisation d'opérations comportant dans leur profil les types des ensembles du type cible.

non utilisation des relations existant entre les éléments des produits cartésiens du type source (par exemple la recherche de l'ensemble des deuxièmes composantes des éléments ayant comme première composante une valeur donnée : $itere_{deux,eq(un(.),p)}$)

TYPE SOURCE-TEP = E[PC[un:P,deux:Q]]

opérations utilisables

importées : constructeurs : evide,adj
modificateurs : single,supe,union,intersec
observateurs : taillee,appe,evide?,eq,inclu,elemsingl
opérateurs : $itere_{fct,prop}$,$sommee_{fcte}$ pour

fct : R ← PC[P,Q] fct(pq) = fct'(f1(un(pq)),f2(deux(pq)))
prop : BOOL ← PC[P,Q] prop(pq) = prop'(f1(un(pq)), f2(deux(pq)))
fcte : ENT ← PC[P,Q] fcte(pq) = fcte'(f1(un(pq)), f2(deux(pq)))
où fct' : R ← P',Q' prop' : BOOL ← P',Q' fcte : ENT←P',Q'
f1 : P' ← P f2 : Q' ← Q
et pour les variables p : P, q : Q, pq : PC[un:P,deux:Q]

TYPE CIBLE-TEP = PC[eun:E[P],edeux:E[Q]]

opérations utilisables

définies : modificateurs : evide,adj,single, supe,union,intersec
observateurs : taillee,appe,evide?, inclu,eqe(tep(eq)=eqe), elemsingl
opérateurs : $itere_{fct,prop}$ $sommee_{fcte}$

variables

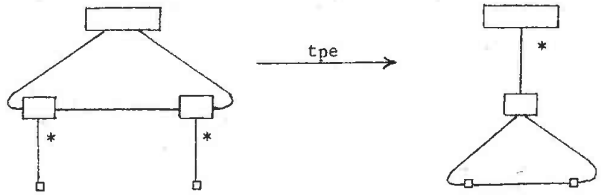
epeq : CIBLE-TEP
pq : PC[un:P deux:Q]

définition du modificateur adj

adj : CIBLE-TEP ← CIBLE-TEP,PC[un:P,deux:Q]
adj(epeq,pq) = c(adj(eun(epeq),un(pq)), adj(edeux(epeq),deux(pq)))

Transformation tpe

d'un produit cartésien d'ensembles en un ensemble de couples



La transformation d'un produit cartésien d'ensembles en un ensemble de couples se fait en associant à chaque élément d'un des ensembles appelé ensemble guide, un élément du deuxième ensemble vérifiant une relation donnée "rel". Si aucun élément du deuxième ensemble ne peut être associé à un élément de l'ensemble guide, alors c'est l'élément indéfini "indef" qui lui sera associé. Ainsi chaque élément du deuxième ensemble est associé à un (et un seul) élément de l'ensemble guide. Celui-ci comportera donc au moins autant d'éléments que le deuxième ensemble. Il est nécessaire de donner des noms aux composantes du produit cartésien du type cible tandis que ceux du type source seront oubliés.

Critères d'application :

utilisation d'une relation entre chaque élément du deuxième ensemble et un élément de l'ensemble guide.

non utilisation de la structure par ensembles.

TYPE SOURCE-TPE = PC[eun : EP, edeux : EQ]

invariant

$eq(evide,itere_{id,eq(indef,assqp(eun(epeq,.))}(edeux(epeq)))$

variables

epeq : SOURCE-TPE

lexique

l'invariant signifie que tout élément de l'ensemble edeux peut être associé à un élément de l'ensemble guide eun.

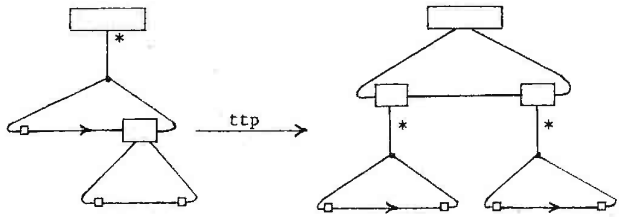
TYPE EP = E[P]	
opérations utilisables définies : assqp	variables ep : EP q : Q
définition assqp : P ← EP,Q assqp(ep,q) = tête(itere$_{id,rel(.,q)}$(ep)) où rel : BOOL ← P,Q	
lexique assqp associe à un objet de type Q donné, un élément de l'ensemble vérifiant la relation "rel"	

TYPE EQ = E[Q]	
opérations utilisables définies : asspq	variables eq : EQ p : P
définition asspq : Q ← EQ,P asspq(eq,p) = tête(itere$_{id,rel(p,.)}$(eq)) où rel : BOOL ← P,Q	
lexique asspq associe à un objet de type P donné un élément de l'ensemble vérifiant la relation "rel"	

TYPE CIBLE-TPE = E[PC[un : P, deux : Q]]	
opérations utilisables définies : modificateurs : c,modifeun, modifedeux observateurs : eun,edeux	variables ep : EP eq : EQ p : P
définition du modificateur c c : CIBLE-TPE ← EP,EQ c(evide,evide) = evide c(adj(ep,p),eq) = adj(c(ep,supe(eq,asspq(eq,p)), c(p,asspq(eq,p))) c(indef,eq) = indef	

Transformation ttp

d'une table de couples en un produit cartésien de tables



La transformation d'une table de couples en un produit cartésien de tables, se fait en dissociant les couples de la table source pour créer des tables différentes. Cependant deux éléments d'un même couple seront accessibles par le même indice chacun dans leur table respective. Il n'y a pas oubli de la relation. Les tables du type cible ont toutes le même ensemble d'indices. Si les noms des composantes du produit cartésien du type source sont oubliés, il faut cependant choisir les noms des tables formant le produit cartésien du type cible.

Critères d'application :

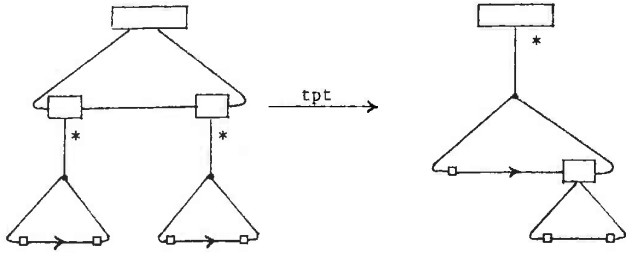
utilisations séparées des composantes du produit cartésien du type source.

TYPE SOURCE-TTP = T[P] de PC[un : Q, deux : R]

TYPE CIBLE-TTP = PC[tun : T[P] de Q, tdeux : T[P] de R]	
opérations utilisables définies : modificateurs : tvide,insert,supt,mod observateurs : acct,taillet,appt,tvide? eqt(ttp(eq) = eqt) opérateurs : itert$_{fct,prop}$,sommet$_{fcte}$	variables tpqtpr : CIBLE-TTP p : P qr : PC[un:Q deux:R]
définition du modificateur insert insert : CIBLE-TTP ← CIBLE-TTP,P,PC[un:Q,deux:R] insert(tpqtpr,p,qr) = c(insert(tun(tpq tpr),p,un(qr)), insert(tdeux(tpq tpr,p,deux(qr)))	

Transformation tpt

d'un produit cartésien de tables en une table de couples



La transformation d'un produit cartésien de tables en une table de couples se fait en associant les éléments des tables du type source de même indice. Si un indice d'une des tables source n'existe pas dans la deuxième table, le couple qui sera associé à cet indice dans la table du type cible comportera une composante indéfinie. Si les noms des tables formant le produit cartésien du type source sont oubliés, il faut cependant choisir les noms des composantes du produit cartésien du type cible.

Critères d'application

utilisation en même temps des éléments associés aux mêmes indices dans les tables du type source : c(acct(tun,p),acct(tdeux,p))

TYPE SOURCE-TPT = PC[tun : T[P] de Q, tdeux : T[P] de R]]

TYPE CIBLE-TPT = T[P] de PC[un : Q, deux : R]	
opérations utilisables définies : modificateurs : c,modiftun,modiftdeux observateurs : tun,tdeux,eqp (tpt(eq) = eqp)	variables tpq : T[P] de Q tpr : T[P] de R p : P, q : Q
définition du modificateur c c : CIBLE-TPR ← T[P] de Q, T[P] de R c(tvide,tpr) = $itert_{c(indef,r(.)),vrai}$(tpr) c(tpq,tvide) = $itert_{c(q(.),indef),vrai}$(tpq) c(insert(tpq,p,q),tpr) = insert(c(tpq,supt(tpr,p)),p,c(q,acct(tpr,p)))	

1.2.2.- Transformations de généralisation

Un produit cartésien est une association d'un nombre fixé d'éléments de types différents.

Nous appelons généralisation les transformations

. g donnant à un type quelconque, la structure de produit cartésien et

. gp augmentant le nombre de composantes d'un produit cartésien.

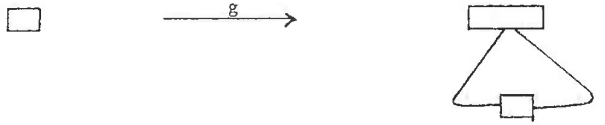
$$P \xrightarrow{g} PC[P] \quad \text{et} \quad PC[P,Q] \xrightarrow{gp} PC[P,Q,R]$$

La transformation gp se généralise à un nombre quelconque de composantes.

Ces transformations permettent à un type, d'évoluer.

Transformation g

d'un type en un produit cartésien



La transformation d'un type en un produit cartésien se fait en considérant un objet du type source comme étant la seule composante d'un produit cartésien.

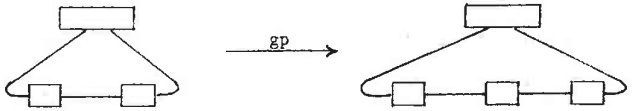
Critères d'application : - utilisation conjointe de deux objets
- unification de n-uplets

TYPE SOURCE-G = P

TYPE CIBLE-G = PC[un : P]
opérations utilisables définies : {g(f)/f ∈ oputil(P)}
définitions de g(f) g(f) = f∘un remarque : les opérations g(f) seront encore appelées f

Transformation gp

d'un produit cartésien à deux composantes en un produit cartésien à trois composantes



La transformation d'un produit cartésien à deux composantes en un produit cartésien à trois composantes, se fait en associant aux objets du type source une composante indéfinie supplémentaire dont il faut donner le nom et le type. Cette transformation se généralise à un nombre quelconque de composantes.

Critères d'application : - utilisation conjointe d'un produit cartésien et d'un autre objet
- unification de n-uplets.

TYPE SOURCE-GP = PC[un : P , deux : Q]

TYPE CIBLE-GP = PC[un : P, deux : Q, trois : R]

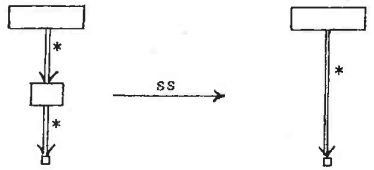
1.2.3.- Transformations de simplification

Les transformations de simplification permettent à partir d'un type hiérarchique, de le rendre peut être plus compréhensible ou plus facile à manipuler. Pour cela elles réduisent la hiérarchie ou plus intuitivement les "mettent à plat". A chaque structure étudiée correspond une transformation de simplification.

SUITE	ENSEMBLE	TABLE	PRODUIT CARTESIEN
S[S[P]] ↓ss S[P]	E[E[P]] ↓se E[P]	T[P] de T[Q] de R ↓st T[PC[P,Q]] de R	PC[PC[P,Q],R] ↓sp PC[P,Q,R]

Transformation ss

d'une suite de suites en une suite



La transformation d'une suite de suites en une suite se fait en concaténant les différentes sous suites du type source. Le rang des éléments dans leurs sous suites sera donc oublié.

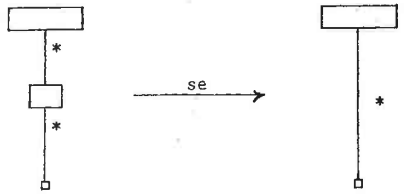
Critères d'application :
- utilisation des éléments des sous suites du type source individuellement.
- non utilisation de la structure en sous suites.

TYPE SOURCE-SS = S[S[P]]
opérations utilisables
importées : constructeurs : svide,ajout
modificateurs : singl,conc
observateurs : app
opérateurs : $iter_{iter_{fct,prop},vrai}$, $somme_{somme_{fcte}}$

TYPE CIBLE-SS = S[P]	
opérations utilisables définies : modificateurs : svide,sajout,ssingl, ssupelt,sconc observateurs : sapp remarque : $ss(iter_{iter_{fct,prop},vrai}) = iter_{fct,prop}$ et $ss(somme_{somme_{fcte}}) = somme_{fcte}$	variables csp : CIBLE-SS sp : S[P]
définition du modificateur sajout sajout : CIBLE-SS * CIBLE-SS,S[P] sajout(csp,sp) = conc(csp,sp)	

Transformation se

d'un ensemble d'ensembles en un ensemble



La transformation d'un ensemble d'ensembles en un ensemble, se fait par union des sous ensembles du type source. Si les sous-ensembles n'avaient pas des intersections vides, ne sera gardée qu'une occurrence de chaque élément des intersections. De ce fait la taille de l'ensemble résultant de la transformation n'est pas égale à la somme des tailles des sous ensembles du type source.

Critères d'application :

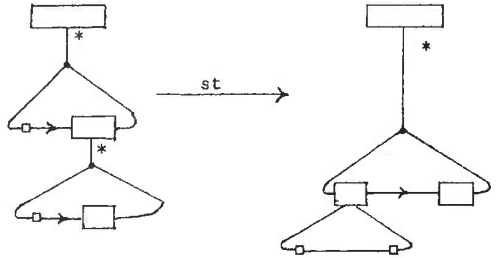
- utilisation des éléments des sous ensembles du type source individuellement.
- non utilisation de la structure en sous ensembles.

TYPE SOURCE-SE = E[E[P]]
opérations utilisables importées : constructeurs : evide,adj modificateurs : single,union observateurs : appe, elemsingl opérateurs : $itere_{itere_{fct,prop},vrai}$, $sommee_{sommee_{fcte}}$

TYPE CIBLE-SE = E[P]	
opérations utilisables définies : modificateurs : svide,sadj,ssingle,sunion observateurs : sappe,(se(elemsingl)=id) remarque : $se(itere_{itere,vrai}) = itere$, $se(sommee_{sommee_{fcte}}) = sommee_{fcte}$	variables cep : CIBLE-SE ep : E[P]
Définition du modificateur sadj sadj : CIBLE-SE ← CIBLE-SE,E[P] sadj(cep,ep) = union(cep,ep)	

Transformation st

d'une table de tables en une table



La transformation d'une table de tables en une table se fait en associant chaque élément des sous-tables au couple formé de l'indice de la sous-table à laquelle ils appartenaient et de leur indice dans cette sous-table.

Critères d'application :

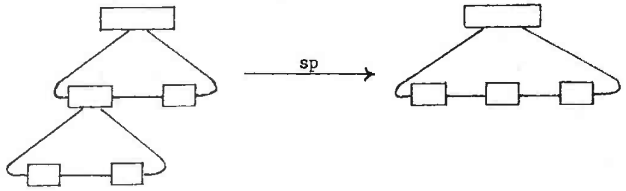
- utilisation des éléments des sous-tables du type source individuellement.
- non utilisation de la structure en sous-tables.

TYPE SOURCE-ST = T[P] de T[Q] de R

TYPE CIBLE-ST = T[PC[P,Q]] de R	
opérations utilisables définies : modificateurs : stvide,sinsert,ssupt, smod observateurs : sacct,staillet,sappt opérateurs : $sitert_{fct,prop}$, $sommet_{fcte}$	variables ctpqr : CIBLE-ST tqr : T[Q] de R p : P
définition du modificateur sintert sinsert : CIBLE-ST ← CIBLE-ST,P,T[Q] de R sinsert(ctpqr,p,tqr) = $itert_{insert(ctpqr,c(p,un(.)),deux(.)),vrai}$ (tqr)	

Transformation sp

d'un produit cartésien de couples en un produit cartésien



La transformation d'un produit cartésien de couples en un produit cartésien se fait en "oubliant" la hiérarchie des produits cartésiens du type source. Cette transformation se généralise à un nombre quelconque de composantes.

Critères d'application :

- utilisation des composantes des sous-produits cartésiens du type source individuellement.
- non utilisation de la structure en sous-produits cartésiens.

TYPE SOURCE-SP = PC[a : PC[un : P, deux : Q], b : R]

TYPE CIBLE-SP = PC[aun : P, adeux : Q,b : R]	
opérations utilisables définies : modificateurs : sc,smodifa observateur : a	variables pq : PC[P,Q] r : R
définition du modificateur sc sc : CIBLE-SP ← PC[P,Q],R sc(pq,r) = c(un(pq),deux(pq),r)	

1.2.4.- Transformations de développement

Les transformations de développement permettent de rendre un type hiérarchique,plus apte à évoluer.

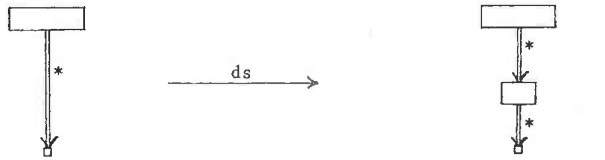
Pour cela, elles introduisent de nouvelles sous structures selon un critère donné : une relation d'équivalence ou un indice commun par exemple.

A chaque structure étudiée correspond une transformation de développement :

SUITE	ENSEMBLE	TABLE	PRODUIT CARTESIEN
S[P] ↓ ds S[S[P]]	E[P] ↓ de E[E[P]]	T[PC[P,Q]] de R ↓ dt T[P] de T[Q] de R	PC[P,Q,R] ↓ dp PC[PC[P,Q],R]

Transformation ds

d'une suite en une suite de suites



La transformation d'une suite en une suite de suites se fait en formant des sous suites de la suite source, selon un critère donné (une relation d'équivalence).

Critères d'application :

- utilisation des objets de la suite source par classes distinctes. Ces classes sont formées à l'aide d'un opérateur de relation.
- non utilisation des objets de la suite source individuellement.

TYPE SOURCE-DS = S[PR]

TYPE PR = P	
opérations utilisables définie : rel	variables pr,pr',pr" : PR

définition de rel

```
rel : BOOL ← PR,PR
      rel(pr,pr) = vrai
      rel(pr,pr') = rel(pr',pr)
(rel(pr,pr') et rel(pr',pr")) impl(rel(pr,pr")) = vrai
                    rel(indef,pr) = faux
```

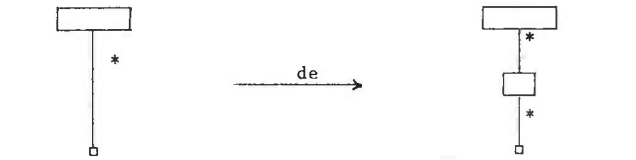
TYPE CIBLE-DS = S[S[PR]]	
opérations utilisables définies : modificateurs : dsvide,dajout,dsingl, dsup,dajoutrg,dmodif, dconc observateurs : dacc,dtête,drang,dtaille, dapp,dsvide?,deq opérateurs : $\text{diter}_{fct,prop}$,dsomme_{fcte}	variables sspr : CIBLE-DS spr : S[PR] pr : PR

définition du modificateur dajout

```
dajout : CIBLE-DS ← CIBLE-DS,PR
dajout(svide,pr) = singl(singl(pr))
dajout(ajout(sspr,spr),pr)
      = si rel(tête(spr),pr)
          alors ajout(sspr,ajout(spr,pr))
          sinon ajout(dajout(sspr,pr),spr)
```


Transformation de

d'un ensemble en un ensemble d'ensembles



La transformation d'un ensemble en un ensemble de sous ensembles se fait en regroupant les éléments de l'ensemble de type source selon un critère donné (une relation d'équivalence).

Critères d'application :

- utilisation des éléments de l'ensemble source par classes distinctes. Les classes sont formées à l'aide d'un opérateur de relation.
- non utilisation des éléments de l'ensemble source individuellement.

TYPE SOURCE-DE = E[PR]

TYPE PR = P	
opérations utilisables définies : rel	variables pr,pr',pr" : PR

définition de rel

```
rel : BOOL ← PR,PR
      rel(pr,pr') = vrai
      rel(pr,pr') = rel(pr',pr)
      ((rel(pr,pr')
            et rel(pr',pr"))impl rel(pr,pr")) = vrai
      rel(indef,pr) = faux
```

TYPE CIBLE-DE = E[E[PR]]

opérations utilisables

définies : modificateurs : devide,dadj,dsingle
dunion,dsupe,
dintersec
observateurs : dtaillee,dappe,devide?
deq,delemsingl
opérateurs : $ditere_{fct,prop'}$
$dsommee_{fcte}$

variables

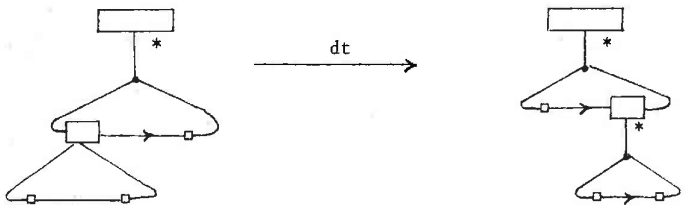
eepr : CIBLE-DS
epr : E[PR]
pr,pr' : PR

définition du modificateur dadj

```
dadj : CIBLE-DE ← CIBLE-DE,PR
       dadj(evide,pr) = single(single(pr))
       dadj(adj(eepr,evide),pr) =
           = adj(dadj(eepr,pr),evide)
       dadj(adj(eepr,adj(epr,pr)),pr')
           = si rel(pr,pr')
                alors adj(eepr,adj(adj(epr,pr),pr'))
                sinon adj(dadj(eepr,pr'),adj(epr,pr))
```


Transformation dt

d'une table en une table de tables



La transformation d'une table à deux dimensions en une table de tables se fait en associant à un des indices la table qui associe aux seconds indices les éléments correspondants.

Critères d'application :

- utilisation d'éléments groupés par la propriété "possédent le même $2^{\text{ème}}$ indice".
- non utilisation des éléments de la table source individuellement.

TYPE SOURCE-DT = T[PC[un : P, deux : Q]] de R

TYPE CIBLE-DT = T[P] de T[Q] de R

opérations utilisables

définies : modificateurs : dtvide,dinsert,dsupt,
dmod
observateurs : dacct,dtaillet,dappt,
dtvide?,deq
opérateurs : $ditert_{fct,prop'}$
$dsommet_{fcte}$

variables

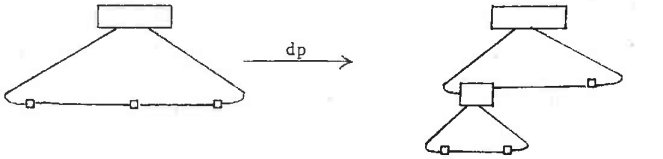
ttpqr : CIBLE-DT
pq : PC[un:P,
deux:Q]
r : R

définition du modificateur dinsert

```
dinsert : CIBLE-DT ← CIBLE-DT, PC[P,Q],R
dinsert(ttpqr,r) =
    = si appt(ttpqr,un(pq))
      alors mod(ttpqr,un(pq),insert(acct(ttpqr,un(pq)),deux(pq),r))
      sinon insert(ttpqr,un(pq),insert(tvide,deux(pq),r))
```

Transformation dp

d'un produit cartésien en un produit cartésien de couples



La transformation d'un produit cartésien en un produit cartésien de couples se fait en regroupant certaines composantes du produit cartésien source pour en faire une composante du produit cartésien cible. Cette transformation se généralise à un nombre quelconque de composantes.

Critères d'application

- utilisation conjointe de plusieurs composantes d'un produit cartésien.
- non utilisation de ces composantes individuellement.

TYPE SOURCE-DP = PC[un : P, deux : Q, trois : R]

TYPE CIBLE-DP = PC[un : PC[unun : P, undeux : Q],deux : R]	
opérations utilisables définies : modificateurs : dc,dmodifun,dmodifdeux observateurs : un,deux	variables p : P, q : Q r : R
définition du modificateur dc dc : CIBLE-DP ← P,Q,R dc(p,q,r) = c(c(p,q),r)	

2.- ANALYSE D'UN UNIVERS

Nous avons défini un ensemble de transformations. Nous allons à présent donner des critères de choix de transformations à appliquer. En effet suivant le but recherché, il faudra appliquer telle ou telle transformation. Ces buts peuvent être

- l'optimisation des opérations définies d'un type,
- la suppression de la redondance,
- la simplification ou
- le développement.

Remarquons que la suppression de la redondance nécessite une étude globale de l'univers et que la simplification ou le développement ne peuvent se justifier que s'ils sont appliqués à l'ensemble de l'univers. Aussi est-il nécessaire de l'étudier globalement avant de choisir les transformations à appliquer.

Nous définissons la propriété de dérivabilité et la relation de ressemblance permettant de comparer, via des transformations, les types d'un univers. Puis pour chacun des buts exposés ci-dessus, nous donnons des critères de choix de transformations. Enfin nous décrivons un schéma de transformation d'un univers.

2.1.- Dérivation et ressemblance

Pour analyser un univers, il faut disposer de critères de comparaison des types le composant. Les relations "être issu" et de parenté relient des types de même structure. La ressemblance relie des types de structures différentes.

Deux types sont ressemblants s'il est possible par transformation d'obtenir deux types parents.

Rappelons que deux types parents peuvent être unis. La propriété de dérivabilité d'un type par une transformation indique si une transformation peut être appliquée sur un type donné.

Définition : Un type T est dérivable par une transformation r si et seulement si r est applicable sur T.

Propriété : Une transformation r est applicable sur un type T si et seulement si T est issu du type source de r.

Exemple de type dérivable

Reprenons l'exemple du loueur de bateaux.

Etudions le type DOUBLE-CAH.

Il a la structure d'un produit cartésien de suites.

Regardons s'il est dérivable par la transformation tps qui permet d'obtenir (à partir d'un produit cartésien de suites) une suite de couples.

La transformation se fait en associant à chaque élément d'une des suites, appelée suite guide, un élément de la deuxième suite vérifiant une relation donnée rel, ou à défaut la constante indéfinie. Ainsi chaque élément de la deuxième suite doit être associé à un élément de la suite guide (celle-ci comportera au moins le même nombre d'éléments que la deuxième).

Il faut donc choisir une suite guide et une relation permettant d'associer tout élément de la deuxième suite à un et un seul élément de la suite guide.

Le type DOUBLE-CAH est le type des doubles cahiers. Ceux-ci sont composés d'un cahier des départs de type CAHD et d'un cahier des retours de type CAHR. Les deux cahiers ont une structure de suite de lignes où une ligne est un produit cartésien d'un numéro de bateau et d'une heure :

expr(DOUBLE-CAH) = PC[cahdep : CAHD, cahret : CAHR]
expr(CAHD) = S[LIGNE]
expr(CAHR) = S[LIGNE]
expr(LIGNE) = PC[numbat : NUMBAT, heure : HEURE]

A chaque départ de bateau (correspondant à une ligne du cahier des départs) on peut associer le retour correspondant à la même location (inscrit dans le cahier des retours). A un instant donné tous les bateaux loués n'étant pas rentrés, le cahier des départs contient toujours au moins autant d'éléments que le cahier des retours. C'est donc le cahier des départs qui sera la suite guide. La relation associant à un départ, le retour correspondant a déjà été défini : il s'agit de la relation 'reldepret', définie dans le type LIGNE.

Etudions maintenant si le type DOUBLE-CAH est issu d'une instance du type source de la transformation tps :

σ(SOURCE-TPS) $\overset{?}{\rightsquigarrow}$ DOUBLE-CAH

- structure(DOUBLE-CAH) $\overset{?}{=}$ σ(structure (SOURCE-TPS))

  structure(DOUBLE-CAH) = PC[cahdep:CAHD,cahret:CAHR]
  = PC[cahdep:SUITE[LIGNE],cahret:SUITE[LIGNE]]
  = σ(expression(SOURCE-TPS))

  où $\sigma(P) = \sigma(Q)$ = LIGNE

- opimputil(DOUBLE-CAH) $\subseteq$ σ(opimputil(SOURCE-TPS))

  car . opimputil(DOUBLE-CAH) = {c(svide,svide)}
  $\subseteq$ σ(opimputil(SOURCE-TPS))

  . opimputil(CAHD) = opimputil(CAHR) = {svide,ajout}
  $\subset$ σ(opimputil(SP))
  $\subset$ σ(opimputil(SQ))

- opdefutil(DOUBLE-CAH) $\supseteq$ σ(opdefutil(SOURCE-TPS))

  car . opdefutil(SOURCE-TPS) = $\emptyset$
  . opdefutil(SP) ={assqp}
  l'opération définie derdep du type CAHD a la même définition que l'opération assqp pour la relation eq(numbat(.),b)
  . opdefutil(SQ) ={asspq}
  l'opération définie derret du type CAHR a la même définition que l'opération asspq pour la relation eq(numbat(.),b)

- le type DOUBLE-CAH vérifie l'invariant du type SOURCE-TPS :
  tout retour enregistré peut être associé à un et un seul départ.

Le type DOUBLE-CAH est donc dérivable par la transformation tps.

Définition :

Le type r(T) obtenu par transformation de T par r est appelé le dérivé de T par r.

- son expression est celle du type cible de r, instancié de la même manière que le type source.

- ses opérations sont les transformées des opérations de T,
- ses constructeurs sont la transformée des constructeurs de T,
- son invariant est la conjonction de la transformée de l'invariant de T et de l'invariant du type cible de r.

ou plus formellement : si TC est le type cible de r

. expr(r(T)) = σ(expr(TC))

. oper(r(T)) = r(oper(T))

mais la transformée d'une opération importée pouvant être une opération définie et la transformée d'une opération définie, une opération importée, la seule partition qui reste stable est celle associée aux concepts d'opérations utilisables et d'opérations cachées :

oputil(r(T)) = r(oputil(T))

construct(r(T)) = r(construct(T))

. invar(r(T)) = r(invar(T)) et invar(TC)

Remarque : r(T) est parent de TC. Mais r(T) n'est pas issu de TC car il ne comporte pas forcément toutes les opérations définies de TC.

Exemple de dérivation :

Dérivation du type DOUBLE-CAH par la transformation tps.

Appelons CAHDR le type dérivé.

- l'expression de CAHDR est celle du type cible de tps, instancié de la même façon que le type source :

  expr(CAHDR) = S[PC[dep : LIGNE, ret : LIGNE]]

  où la composante dep correspond à une ligne du cahier des départs et la composante ret à une ligne du cahier des retours. (Nous avons dû choisir les noms des composantes dep et ret).

- les opérations de CAHDR sont les transformées par tps des opérations de DOUBLE-CAHIER

  opérations importées utilisables : tps(c(svide,svide)) = svide

  opérations définies utilisables : tps(departb) = departbt

  tps(retourb) = retourbt

  tps(calculloc) = calculloct

Définition des opérations par la transformée des définitions :

Posons cdr = tps(dcah)

où dcah est une variable de type DOUBLE-CAH et

cdr une variable de type CAHDR

départb(cdr,ℓ) = si rentré (cdr,numbat(ℓ))
alors ajout(cdr,c(ℓ,indef))
sinon cdr

retourb(cdr,ℓ) = si rentré(cdr,numbat(ℓ))
alors cdr
sinon modif(cdr,rang(cdr,ℓdr),modifret(ℓdr,ℓ))

où ℓdr : PC[dep : LIGNE, ret : LIGNE] est défini par

ℓdr = tête($iter_{id,reldepret(dep(.),ℓ)}$(cdr))

calculloc(cdr) = $iter_{consloc,vrai}$(cdr)

rentré(cdr,b) = pasparti(cdr,b) ou pasreparti(cdr,b)

pasparti doit aussi être modifié si nous ne voulons pas garder la structure de cahier devenue inutile, ainsi que derdep et derret (il y a identification entre le type DOUBLE-CAH transformé en un produit cartésien à une seule composante et sa composante).

parparti(cdr,b) = eq(indef,derdep(cdr,b))

derdep(cdr,b) = tête($iter_{id,eq(numbat(dep(.)),b)}$(cdr))

pasreparti(cdr,b) = reldepret(derdep(cdr,b),derret(cdr,b))

derret(cdr,b) = tête($iter_{id,eq(numbat(ret(.)),b)}$(cdr))

L'opération assdepret est devenue inutile car :

assdepret(cdr,dep(acc(cdr,e))) = ret(acc(cdr,e))

Le type CAHDR a un invariant toujours égal à vrai.

La notion de dérivabilité d'un type par une transformation induit une relation appelée de ressemblance entre les types d'un même univers. Cette relation dépend d'un ensemble de transformations noté $\mathcal{R}$. Celui-ci est construit à partir d'un ensemble de transformations dites de base en y ajoutant la transformation identité et toutes les compositions possibles des transformations entre elles.

Soient $\mathcal{R}$ un ensemble de transformations, T1 et T2 deux types présentés en SPES différents du type VIDE.

Définition : T1 et T2 sont ressemblants si et seulement si il existe r1 et r2 deux transformations de $\mathcal{R}$ telles que

T1 soit dérivable par r1

T2 soit dérivable par r2

et que leurs dérivés soient parents :

$$T1 \xrightarrow{r1} r1(T1) \rightsquigarrow r2(T2) \xleftarrow{r2} T2$$

La relation de ressemblance est une "généralisation" de la relation de parenté. En particulier deux types parents sont ressemblants :

$$T1 \rightsquigarrow T2 \Rightarrow T1 \text{ et } T2 \text{ ressemblants.}$$

Nous avons vu que deux types parents ont des propriétés en commun, celles-ci pouvant être réunies dans un même type appelé leur fusion.

Il en va de même avec deux types ressemblants. Pour réunir leurs propriétés il faut au préalable les dériver. C'est la fusion par dérivation.

Définition : Soient T1 et T2 deux types ressemblants. On appelle fusion par dérivation de T1 par r1 et T2 par r2 la fusion des dérivés de T1 par r1 et de T2 par r2 fusiondériv $(T1|_{r1}, T2|_{r2}) = \text{fusion}(r1(T1), r2(T2))$.

Remarques :

- la relation de ressemblance de deux types est réflexive et symétrique mais pas transitive car la relation de parenté ne l'est pas.
- Il y a en général plusieurs manières de fusionner par dérivation deux types selon le choix des transformations.

Un type et son dérivé par une transformation sont ressemblants et leur fusion par dérivation est égale au dérivé :

$$\text{fusiondériv}(T|_r, r(T)|_{id}) = r(T).$$

Deux types sont ressemblants si et seulement si ils peuvent être fusionnés par dérivation. Cette fusion permet de réunir leurs propriétés.

Exemple de types ressemblants :

Le type DOUBLE-CAH de structure

PC[cahdep:S[LIGNE], cahret:S[LIGNE]]

est dérivable par tps et son dérivé appelé CAHDR a la structure suivante :

S[PC[dep : LIGNE, ret : LIGNE]].

La structure du type ENS-LOC est :

E[PC[bat : NUMBAT, dep : HEURE, ret : HEURE]]

Les types DCAH, CAHDR et ENS-LOC sont ressemblants.

Un type fusion par dérivation de ces trois types pourrait avoir la structure suivante :

S[PC[bat : NUMBAT, dep : HEURE, ret : HEURE]]

Appelons CAHU ce type : Il est obtenu par fusion des types dérivés de CAHDR par sp et de ENS-LOC par tes. (Rappelons que

fusiondériv(DOUBLE-CAH,CAHDR) = CAHDR).

Le type CAHDR est modifié par la transformation par sp de sa sous-structure PC[dep : LIGNE, ret : LIGNE].

Celle-ci est dérivable par sp, car le type de son expression est égal à une instanciation du type SOURCE-SP.

Le type ENS-LOC est dérivable par la transformation tes en un type de structure de suite : le type de son expression est égal à une instanciation du type SOURCE-TES. L'ordre arbitrairement introduit pour transformer un ensemble en une suite est le même que celui du type CAHDR à savoir l'ordre d'enregistrement des départs (ou d'ordre d'insertion des éléments dans la suite).

2.2.- Choix d'une transformation

Les critères de comparaison de types que nous venons de voir permettent d'analyser un univers pour choisir les transformations à appliquer dans le but soit

- d'optimiser les opérations définies d'un type,
- d'éliminer la redondance entre deux types,
- de simplifier une structure,
- de développer une structure.

Nous allons étudier selon les buts recherchés, comment analyser un univers.

2.2.1.- Optimisation de la définition des opérations d'un type

La présentation des différentes transformations comporte les critères méthodologiques d'application suivants :

- utilisation de certaines opérations qui bénéficient de la transformation en ce sens que leur définition est optimisée.
- non utilisation de certaines opérations qui pâtissent au contraire de la transformation.

Mais le critère qui souvent guidera le plus est d'être dérivable par la transformation.

> Si un type T vérifie ces critères d'application d'une transformation r
> Alors les définitions des opérations de T bénéficieront de la transformation.
> On dit alors que T est optimisé par r.

Remarquons que généralement ce seront les transformations de structure qui permettront d'optimiser les opérations d'un type.

Exemple : Le type CAHDR est une optimisation du type DOUBLE-CAH dont il est dérivé par la transformation tps. En effet, le type DOUBLE-CAH comporte les opérations suivantes bénéficiant de la transformation :

- assdepret devient triviale (les composantes d'un couple (ligne-départ, ligne-retour) vérifient le prédicat assdepret)
- calculloc, l'opération de construction de l'ensemble des locations n'est plus qu'une itération simple (au lieu de deux imbriquées).

2.2.2.- Recherche et Elimination de la redondance

Il y a redondance dès qu'il y a duplication d'une information : deux structures de données sont redondantes si elles possèdent des informations identiques, éventuellement sous une forme différente ou conjointement avec d'autres informations.

Mais la redondance n'est un défaut que si le volume des informations dupliquées est important.

Pour détecter de la redondance dans une spécification, nous étudions les structures de données : si deux objets de types différents ont des informations en commun, alors les types de ces informations sont ressemblants.

Partant de cette constatation nous étudions les types des objets : si deux objets ont des composantes ou des sous structures de types ressemblants, alors il y a redondance de types. C'est-à-dire il y a peut-être redondance de l'information, mais pas toujours. Il y a sûrement redondance des opérations (ou des fonctionnalités) définies. Dans tous les cas l'élimination de la redondance se fait en fusionnant par dérivation les types redondants. S'il y a redondance des informations alors son élimination fait apparaître une équation de la forme $r_1(d_1) = r_2(d_2)$ identifiant les données redondantes. Dans le cas contraire, le nombre d'opérations (ou de fonctionnalités) d'un système est réduit.

Définition : Soient T1 et T2 deux types redondants.
Il est possible d'éliminer la redondance par fusion par dérivation si et seulement si T1 et T2 sont ressemblants.

Nous étudions plusieurs cas d'introduction de redondance dans le but de formuler des conditions suffisantes d'existence de redondance entre deux types de données.

Nous distinguons deux cas de redondance de structures entraînant une duplication de l'information :

1.- entre deux structures distinctes, ou

2.- dans une même structure.

1er cas : Redondance entre deux structures distinctes

L'analyse descendante d'un problème, propose de spécifier le résultat recherché à l'aide de résultats intermédiaires introduits. Ceux-ci devront être spécifiés à leur tour. Le processus s'arrêtera lorsque tous les résultats introduits seront reconnus être des données du problème :

$res = f(...res_i...)$ où res est le résultat recherché

$res_i = f_i(...res_j...)$ res_i, res_j, res_k sont des résultats intermédiaires introduits

...

$res_k = f_k(...d_\ell...)$ d_ℓ est une donnée du problème.

Les résultats intermédiaires res_i, res_j, ...res_k ne contiennent pas de nouvelles informations par rapport aux données ...d_ℓ..., puisqu'ils ont été calculés à partir d'elles. Il y a donc redondance de l'information. Ceci est gênant si les résultats intermédiaires sont de types structurés et que leur réalisation occupe beaucoup de place mémoire. C'est-à-dire s'il n'y a pas beaucoup de perte d'information entre les données et les résultats intermédiaires. Ce sera le cas si les types des résultats intermédiaires et des données sont ressemblants.

Caractérisation du 1er cas de redondance :

Il y a redondance d'information entre deux types d'un univers si les objets de l'un peuvent être construits à partir des objets de l'autre, par application d'une opération :

Soient T1 et T2 deux types d'un univers.
Si - T1 et T2 sont ressemblants et si
- l'un des deux est muni d'une opération f : T1 ← T2 telle que
  • soit f est un des constructeurs des objets de T1,
  • soit les constructeurs des objets de T1 sont des opérations importées.
Alors T1 et T2 sont redondants.

Exemple : Dans l'univers u_1 du loueur de bateaux, les types DOUBLE-CAH et ENS-LOC sont ressemblants.

Le type DCAH est muni de l'opération calculloc : ENS-LOC ← DCAH permettant de construire l'ensemble des locations à partir d'un double cahier. Les constructeurs du type ENS-LOC sont ceux importés de l'expression.

Donc les types DCAH et ENS-LOC sont redondants.

La fusion par dérivation de DOUBLE-CAH et de ENS-LOC identifiera les doubles cahiers et les ensembles de locations calculés par l'opération calculloc à partir de ceux-ci.

2ème cas : Redondance d'une information dans une structure

Lors de l'analyse descendante d'un problème, les données sont introduites au moment de leur utilisation. Si pour résoudre un problème une même donnée est utilisée à plusieurs reprises, elle apparaîtra plusieurs fois dans les structures de données et risque ainsi d'être implantée à plusieurs reprises. C'est ce qui se passe dans l'exemple du loueur de bateaux :

Le type du cahier des locations CAHLOC, dérivé du type DCAH du double cahier a la structure d'une suite de couples de deux lignes, l'une correspondant au départ, l'autre au retour. Chaque ligne précise le numéro du bateau. Mais les couples du produit cartésien ont été formés de telle manière qu'il s'agisse du même numéro de bateau :

```
expr(CAHLOC) = S[PC[dep : LIGNE, ret : LIGNE]]
avec expr(LIGNE) = PC[numbat : NUMBAT, heure : HEURE]
invar(CAHLOC) = eq(numbat(dep(acc(cahℓ,e))),
                   numbat(ret(acc(cahℓ,e))))
```

Il y a bien redondance de l'information.

L'élimination de la redondance se fait ici aussi par fusion par dérivation. Les dérivations sont les transformations caractéristiques permettant d'ajouter un champ à un produit cartésien : Les transformations de généralisation. Elles sont généralement utilisées composées avec la transformation de simplification du produit cartésien.

Caractérisation du 2ème cas de redondance :

Il y a redondance d'information dans une structure si elle est composée d'un produit cartésien de deux sous-structures redondantes.

La fusion par dérivation de ces deux sous-structures identifiera les données redondantes.

Un cas courant est la redondance dans un type ayant la structure d'un produit cartésien de n-uplets.

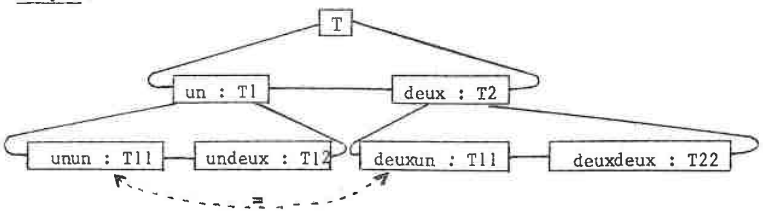
Etudions plusieurs cas de transformations possibles permettant de l'éliminer.

1.- Elimination par "mise à plat"

TYPE T1 = PC[unun : T11, undeux : T12]
TYPE T2 = PC[deuxun : T11, deuxdeux : T22]
TYPE T = PC[un : T1, deux : T2]
invariant eq(unun (un(t)), deuxun (deux(t))
variable t : T

Les composantes unun et deuxun sont égales :

Graphe :



Les types T1 et T2 sont redondants. Ils peuvent être fusionnés par dérivation par la transformation gp de généralisation :

$$T1 \xrightarrow{gp} TT \xleftarrow{gp} T2$$

TYPE TT = PC[unun : T11, undeux : T12, deuxdeux : T22]

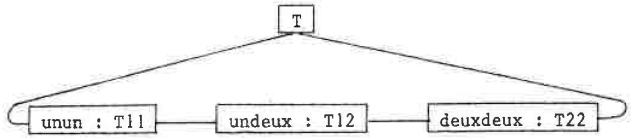
Les composantes unun et deuxun sont identifiées.

Le type T a la nouvelle structure suivante :

TYPE T = PC[un : TT]

simplifiée en TYPE T = TT par la transformation de simplification du produit cartésien à une seule composante, à sa composante.

Nouveau graphe :

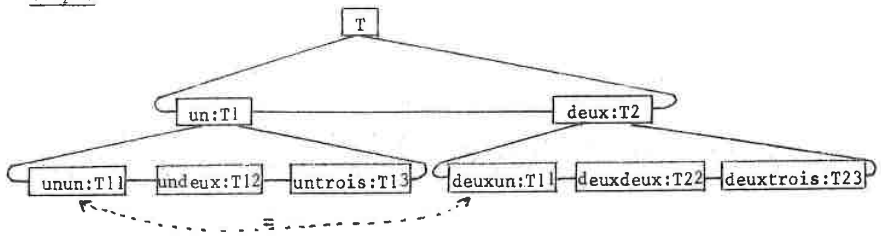


2.- Elimination par "remontée"

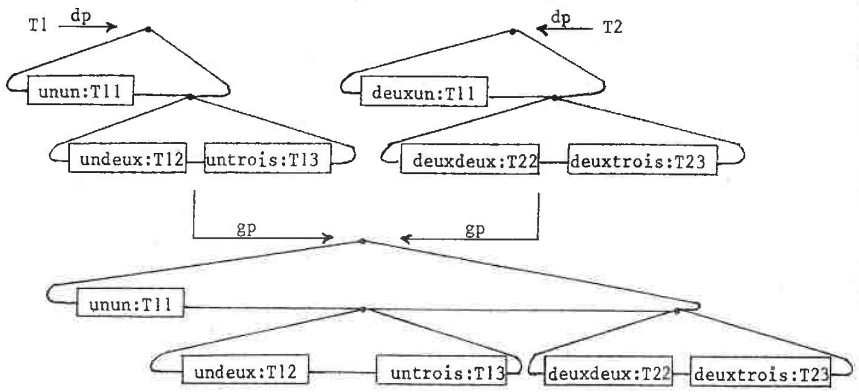
TYPE T1 = PC[unun : T11, undeux : T12, untrois : T13]
TYPE T2 = PC[deuxun : T11, deuxdeux : T22, deuxtrois : T23]
TYPE T = PC[un : T1, deux : T2]
invariant eq(unun(un(t)),deuxun(deux(t))
variable t : T

Les composantes unun et deuxun sont égales.

Graphe :



Les types T1 et T2 sont redondants. Ils peuvent être fusionnés par dérivation par la transformation composée gp.dp.



Les composantes unun et deuxun sont identifiées.

2.2.3.- Simplification d'une structure

Une structure trop hiérarchique est généralement moins "parlante" qu'une structure "a plat". Aussi il peut être agréable de réduire les "sous-structures" introduites.

> Si un type T est dérivable par une transformation de simplification, alors le type dérivé sera "plus simple".

La simplification des types d'un univers constitue la première étape de sa transformation.

2.2.4.- Développement d'une structure

Il est généralement admis qu'une structure composée de sous-structures est plus apte à évoluer qu'une structure "en un bloc".

> Si un type T est dérivable par une transformation de développement alors, le type dérivé sera plus apte à évoluer.

Le développement des types d'un univers constitue la dernière étape de sa transformation.

2.3.- Transformation d'un univers

Nous décrivons un schéma de transformation d'un univers.

Puis nous l'appliquons au petit exemple du loueur de bateaux. Un exemple plus complet est étudié au paragraphe suivant.

2.3.1.- Schéma de transformation

Le schéma de transformation d'un univers doit être considéré non comme un guide méthodologique, mais comme un fil conducteur. La transformation d'un univers passe par plusieurs étapes et l'univers obtenu à la fin de chacune est une spécification du problème à résoudre. Ainsi il peut être considéré à son tour comme point de départ ou comme point d'arrivée du processus de transformation.

Le schéma de transformation est composé de quatre étapes :

1.- Simplification de l'Univers

Pour faire apparaître plus clairement les composantes redondantes d'une structure, il faut la simplifier. Dans une première étape nous appliquons toutes les transformations de simplification applicables.

2.- Elimination de la redondance

a - Dans une structure

Un univers est composé de plusieurs structures indépendantes (bien qu'elles puissent avoir des types et des informations en commun). Pour chacune nous éliminons la redondance en la parcourant du plus grand type pour la relation $\geqslant$ vers les plus petits.

b - Entre deux structures

Par comparaison des types intervenant dans deux structures différentes, nous recherchons ceux qui sont ressemblants : il peut alors y avoir redondance de l'information. L'élimination de la redondance dans ce cas peut transformer l'univers en profondeur puisqu'elle permet d'identifier des données de structures distinctes.

3.- Optimisation des opérations d'un type

Cette étape transforme les structures des types de l'univers, de façon indépendante, dans le but d'optimiser la définition des opérations. Cette étape doit être effectuée après toutes les transformations profondes d'un univers. L'univers est parcouru des plus grands types pour la relation $\geq$ vers les plus petits.

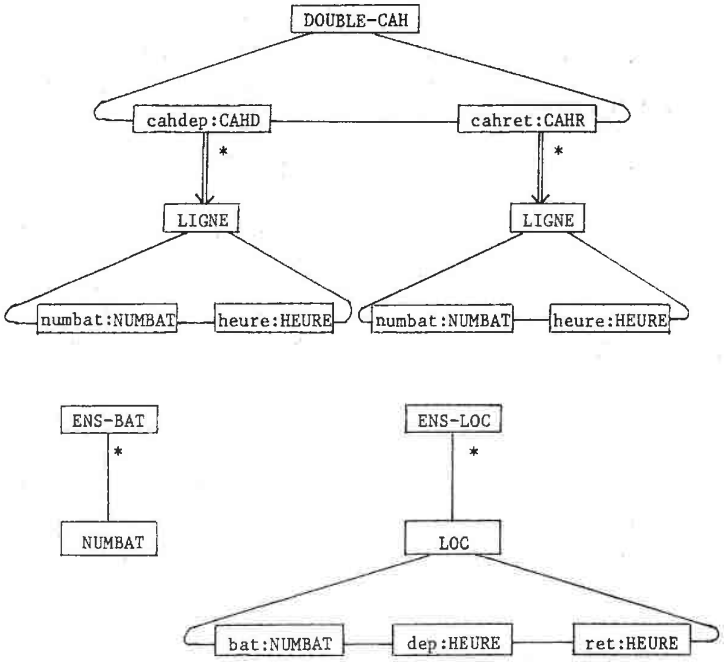
4.- Développement de l'univers

Pour rendre un univers transformé plus apte à évoluer, il faut le rendre plus modulaire : nous appliquons enfin toutes les transformations de développement applicables.

Dans le cas général, les transformations ne modifient pas l'aspect externe d'un type : les noms des types et des opérations ainsi que leur comportement restent les mêmes. Il n'est donc pas nécessaire de répercuter sur l'univers les transformations. Cependant lors de la fusion par dérivation il y a unification des noms des types et des objets : dans ce cas une simple modification purement syntaxique est nécessaire dans les types utilisant les types fusionnés.

2.3.2.- Exemple de transformation d'un univers : le loueur de bateaux

Reprenons l'univers u_1 du loueur de bateaux composé des structures suivantes :



1ère Etape : Simplification de l'univers

Aucune transformation de simplification ne peut être appliquée.

2ème Etape : Elimination de la redondance

a - Dans une structure

Le type DOUBLE-CAH est composé d'un produit cartésien de deux sous-structures de types CAHD et CAHR.

CAHD et CAHR sont parents puisqu'ils ont la même structure (donc des expressions équivalentes) et des invariants égaux à "vrai". Ils sont donc ressemblants. Mais CAHD et CAHR ne sont pas des informations redondantes car ni l'un ni l'autre n'est muni d'une opération de profil CAHR ← CAHD ou CAHD ← CAHR.

Nous allons tout de même les fusionner pour diminuer le nombre de types : Appelons CAH le type fusion de CAHD et CAHR.

Il est muni des opérations de CAHD (pasparti,derdep) et de celles de CAHR (derret, assdepret). Remarquons que les opérations derdep et derret ont la même définition : dernière ligne enregistrée d'un cahier cah se référant à un bateau b donné ($tête(iter_{id,eq(numbat(.),b)}(cah))$.

Dans le type CAH nous appelons derenr cette opération.

TYPE CAH = SUITE[LIGNE]	
opérations utilisables importées : svide,ajout définies : pasparti,derenr,assdepret	déclarations de variables cah : CAH b : NUMBAT, ℓ : LIGNE

définitions

pasparti : BOOL ← CAH,NUMBAT

pasparti(cah,b) = eq(indef,derenr(cah,b))

derenr : LIGNE ← CAH,NUMBAT

derenr(cah,b) = $tête(iter_{id,eq(numbat(.),b)}(cah))$

assdepret : LIGNE ← CAH,LIGNE

assdepret(cah,ℓ) = $tête(iter_{id,reldepret(ℓ,.)}(cah))$

Lexique

TYPE CAH : type des cahiers des départs et des retours

pasparti : prédicat indiquant qu'un bateau n'est pas enregistré dans un cahier (s'applique sur un cahier des départs)

derenr : accès au dernier enregistrement d'un bateau

assdepret : accès au dernier enregistrement pouvant correspondre à la même location que la ligne (correspondant à un départ) donnée (s'applique sur un cahier des retours)

D'un point de vue externe, les types CAHD et CAHR ont été modifiés :

- ils ont changé de nom et s'appellent CAH,
- les opérations derdep et derret aussi et s'appellent derenr.

Il faut repercuter ces modifications syntaxiques sur les types utilisant les types CAHD et CAHR : il n'y a que le type DOUBLE-CAH. Seuls son expression et la définition de pasreparti sont modifiés.

TYPE DOUBLE-CAH = PC[cahdep :CAH, cahret : CAH]	
opérations utilisables importées : c(svide,svide) définies : (c)departb,(c)retourb calculloc	déclaration de variables dcah : DOUBLE-CAH ℓ : LIGNE b : NUMBAT

définitions

(c) departb : DOUBLE-CAH ← DOUBLE-CAH,LIGNE

departb(dcah,ℓ)

= si rentré(dcah,numbat(ℓ))

alors modifcahdep(dcah,ajout(cahdep(dcah),ℓ))

sinon dcah

(c) retourb : DOUBLE-CAH ← DOUBLE-CAH,LIGNE

retourb(dcah,ℓ)

= si rentré(dcah,numbat(ℓ))

alors dcah

sinon modifcahret(dcah,ajout(cahret(dcah),ℓ))

rentré : BOOL ← DOUBLE-CAH,NUMBAT

rentré(dcah,b) = pasparti(cahdep(dcah),b)

ou pasreparti(dcah,b)

pasreparti : BOOL ← DOUBLE-CAH, NUMBAT

pasreparti(dcah,b)

= reldepret(derenr(cahdep(dcah),b),

derenr(cahret(dcah),b))

calculloc : ENS-LOC ← DOUBLE-CAH

calculloc(dcah)

= $iter_{consloc(.,assdepret(cahret(dcah),.)),vrai}(cahdep(dcah))$

lexique :

TYPE DOUBLE-CAH : type des doubles cahiers composés d'un cahier des départs (cahdep) et d'un cahier des retours (cahret)

departb : enregistrement d'un départ de bateau

retourb : enregistrement d'un retour de bateau

rentré : prédicat indiquant si un bateau n'est pas parti ou s'il est revenu après son dernier départ

pasreparti : prédicat indiquant si un bateau dont un retour a été enregistré, n'a pas de départ postérieur enregistré

calculloc : calcule la liste des locations à partir du double cahier.

b - Entre deux structures

Les structures que nous avons à comparer sont celles des types DOUBLE-CAH, ENS-BAT et ENS-LOC.

Le type ENS-BAT de structure d'ensemble de numéros de bateaux (E[NUMBAT]) ne ressemble ni au type DOUBLE-CAH ni au type ENS-LOC.

Les types DOUBLE-CAH et ENS-LOC sont ressemblants (cf. § III.2.1.2). Sont-ils redondants ?

Le type DOUBLE-CAH est muni de l'opération calculloc de profil: calculloc : ENS-LOC ← DOUBLE-CAH.

Les constructeurs du type ENS-LOC sont les constructeurs importés de l'expression (le type ENS-LOC n'est muni que d'une autre opération définie : batloués).

Donc par la caractérisation du 1^{er} cas de redondance, les types DOUBLE-CAH et ENS-LOC sont redondants. S'il y a redondance des informations, cela apparaîtra lors de la fusion par dérivation des deux types.

Les types DOUBLE-CAH et ENS-LOC peuvent être fusionnés par dérivation de DOUBLE-CAH par sp∘tps et de ENS-LOC par tes, ou un type que nous appelons CAHU de structure

S[PC[bat : NUMBAT, dep : HEURE, ret : HEURE]]

Nous avons décrit le type dérivé de DOUBLE-CAH par tps (§ III.2.1.1). Nous donnons le type CAHU. Il est muni des transformées des opérations des types DOUBLE-CAH et ENS-LOC. Nous nommons ces opérations fu pour toute opération f des types DOUBLE-CAH et ENS-LOC. Elles sont définies par la transformée des définitions. Nous avons vu que l'opération assdepret devenait triviale lors de la transformation tps.

L'opération calculloc devient l'identité. D'où identification des objets transformés des doubles-cachiers et des ensembles de location. En effet calculloc était défini dans CAHDR (type dérivé de DOUBLE-CAH par tps) par $iter_{consloc,vrai}$. Or la transformation du type CAHDR par sp est la transformation des couples de lignes en location. Cette transformation est réalisée par l'opération consloc associée au type LIGNE. Donc calculloc est transformé en l'opération $iter_{id,vrai}$, c'est-à-dire l'identité. Il y a bien identification des objets redondants.

TYPE CAHU = S[LOC]

opérations utilisables
importées : svide
définies : (c)departu , (c)retouru
batlouésu

déclarations de variables
cahu : CAHU
loc : LOC
b : NUMBAT
h : HEURE

définitions

(c) departu : CAHU ← CAHU,LIGNE
departu(cahu,ℓ) = si rentréu(cahu,numbat(ℓ))
alors ajout(cahu,consloc(ℓ,indef))
sinon cahu

(c) retouru : CAHU ← CAHU,LIGNE
retouru(cahu,ℓ) = si rentréu(cahu,numbat(ℓ))
alors cahu
sinon modif(cahu,rang(cahu,derenru(cahu,
numbat(ℓ))),
modifret(derenru(cahu,numbat(ℓ)),
heure(ℓ)))

rentréu : BOOL ← CAHU,NUMBAT
rentréu(cahu,b) = paspartiu(cahu,b)
ou pasrepartiu(cahu,b)

paspartiu : BOOL ← CAHU,NUMBAT
paspartiu(cahu,b) = eq(indef,derenru(cahu,b))

pasrepartiu : BOOL ← CAHU,NUMBAT
pasrepartiu(cahu,b) = non(eq(indef,ret(derenru(cahu,b))))

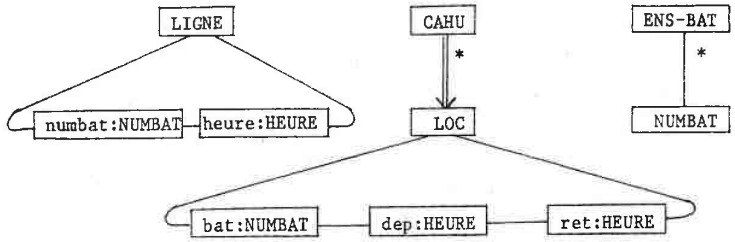
derenru : LOC ← CAHU,NUMBAT
derenru(cahu,b) = tête($iter_{id,eq(bat(.),b)}$(cahu))

batlouésu : ENS-BAT ← CAHU,HEURE
batlouésu(cahu,h) = $iter_{id,louéheure(.,h)}$(cahu)

lexique :

TYPE CAHU	: type des cahiers des locations
departu	: enregistrement du départ d'un bateau
retouru	: enregistrement du retour d'un bateau
rentréu	: prédicat indiquant si un bateau n'est pas parti ou bien s'il est rentré après son dernier départ
paspartiu	: prédicat indiquant si un bateau n'est pas enregistré
pasrepartiu	: prédicat indiquant si un bateau n'a pas de départ enregistré postérieur au dernier retour
derenru	: accès à la dernière location enregistrée d'un bateau
batlouésu	: ensemble des bateaux loués à une heure donnée

L'univers ainsi transformé est composé des types structurés de manière suivante :



où CAHU est muni des opérations définies utilisables suivantes :

departu, retouru, batloués

LOC de l'opération louéheure
LIGNE de consloc (reldepret est devenue inutile).

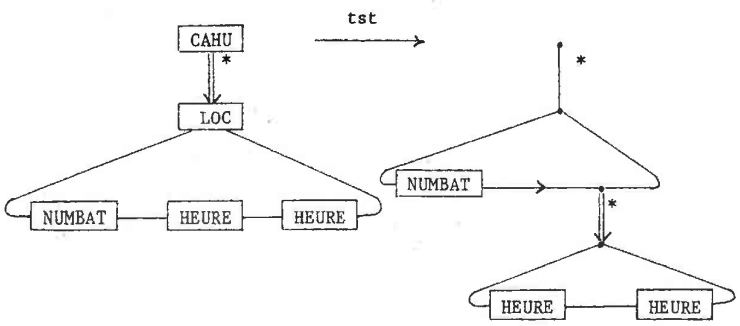
Les types LIGNE et LOC sont ressemblants :
Le type LIGNE est dérivable par la transformation gp en type de structure équivalente à celle du type LOC. Le type LIGNE est muni de l'opération consloc de profil LOC ← LIGNE,LIGNE permettant de construire une location à partir de deux lignes. Le type est utilisé comme paramètre des opérations départu et retouru. Il n'y a pas redondance de l'information. Nous ne fusionnons pas les types LIGNE et LOC.

3ème Etape : Optimisation des opérations

Pouvons-nous appliquer des transformations de structure sur les types de l'univers ?

- Pas sur les types LIGNE et LOC.
- Nous pouvons transformer ENS-BAT en une suite de numéros de bateaux par la transformation tes. Le type ENS-BAT n'étant muni d'aucune opération définie utilisable, le transformer ne l'optimiserait pas.

- CAHU ne peut pas être transformé en un ensemble de location par la transformation tse car l'ordre d'adjonction des éléments est utilisé : L'opération derenr fournit la dernière location enregistrée pour un bateau donné.
- CAHU a été obtenu par transformation du type DOUBLE-CAH par tps. Transformer à présent CAHU par tsp (en un produit cartésien de suites) nous ferait perdre le bénéfice de la première transformation.
- CAHU est dérivable par tst en un type de structure de table : T[NUMBAT] de S[PC[dep : HEURE, ret : HEURE]] associant à un numéro de bateau la suite de ses locations. L'opération derenr serait optimisée en un accès direct à la tête de la suite associée à un bateau donné.



4ème Etape : Développement de l'univers

Aucune transformation de développement n'est applicable.

3.- SPECIFICATION ET TRANSFORMATION DU SYSTEME SPES-TYPES

Nous spécifions le système SPES-TYPES d'aide à la construction d'univers de types. Ce système est basé sur les définitions que nous avons exposé tout au long de ce travail. Puis, suivant le schéma d'analyse exposé ci-dessus, nous transformons sa spécification.

3.1.- Spécification du système SPES-TYPES

3.1.1.- Spécification informelle

Le système doit permettre de construire de manière incrémentale un univers spécifiant un problème donné.

Structure des objets du système

Un univers est un ensemble de présentations de types ; Une présentation de type est composée par :

- un nom,
- une expression,
- un invariant,
- un ensemble de noms d'opérations importées utilisables,
- un ensemble de noms d'opérations définies utilisables,
- l'ensemble des définitions des opérations définies.

Fonctionnalités du système

Parmi les opérations, on distingue celles sur les univers et celles sur les présentations de types. Ces deux ensembles d'opérations sont composés de constructeurs (d'univers ou de types), de modificateurs et d'observateurs.

A - Opérations sur les univers

a - Constructeur

La construction d'un univers se fait par adjonction successive de types. A ce stade un type est défini uniquement par une expression et éventuellement

par un invariant. Ce type importe toutes les opérations utilisables du type de son expression et ne possède aucune opération définie propre.

b - Modificateurs

La seule modification d'un univers est la suppression d'un type.

c - Observateurs

L'observation d'un univers consiste

- à avoir accès à un type défini. L'accès se fait par le nom du type. Le résultat est sa présentation ;
- à comparer des types de l'univers deux à deux pour les relations
  - issu de ;
  - parents ;
  - équivalents ;
  - $\succcurlyeq$.

B - Opérations sur les présentations

a - Constructeurs

Tout type est défini lors de son adjonction dans l'univers, par un nom, une expression et éventuellement un invariant. Pour compléter sa définition, on dispose des opérations suivantes :

- adjonction d'un prédicat à l'invariant ;
- enrichissement par adjonction d'une opération définie ;
- restriction par suppression d'une opération importée.

b - Modificateurs

Pour modifier un type on dispose des opérations suivantes :

- suppression de l'invariant (celui-ci devient égal à "vrai") ;
- suppression d'une opération définie ;
- adjonction d'une opération importée.

c - Observateurs

On peut avoir accès à toutes les composantes d'un type à savoir :

- à son expression ;
- à son invariant ;
- à ses opérations définies (utilisables ou non) ;

De plus on désire pouvoir connaître

- sa structure ;
- le type défini par son expression.

3.2.- Spécification formelle du système SPES-TYPES

TYPE UNIVERS = E[PRES-TYPE]	
invariant infeg(taillee(itere$_{id,eq(ntyp,nom(.))}$(uni)),un)	
opérations utilisables importées : aucune définies : constructeurs : uvide,adjtyp modificateurs : suptyp observateurs : acctyp,struct, existe?,issu?, parent?, equiv?, $\succcurlyeq$?	variables uni : UNIVERS pres : PRES-TYPE ntyp,ntyp' : NOM-TYPE antyp,antyp' : PRES-TYPE
lexique : Univers de présentations de types invariant : un univers ne peut contenir deux types de même nom. opérations définies : uvide : univers vide adjtyp : adjonction d'un type dans un univers suptyp : suppression d'un type nommé acctyp : accès à la présentation d'un type nommé struct : structure d'un type nommé existe? : prédicat d'existence d'un type nommé dans l'univers issu? : prédicat:T est-il issu de T' ? parent? : prédicat:T et T' sont-ils parents ? equiv ? : prédicat:T et T' sont-ils équivalents ? $\succcurlyeq$? : prédicat:T est-il plus grand que T' pour la relation $\succcurlyeq$?	

définitions

constructeurs :

uvide : UNIVERS ←

uvide = evide

adjtyp : UNIVERS ← UNIVERS,PRES-TYPE

adjtyp(uni,pres) = adj(uni,pres)

Posons antyp = acctyp(uni,ntyp) et antyp' = acctyp(uni,ntyp')

modificateur :

suptyp : UNIVERS ← UNIVERS,NOM-TYPE

suptyp(uni,ntyp) = supe(uni,antyp)

observateurs :

acctyp : PRES-TYPE ← UNIVERS,NOM-TYPE

acctyp(uni,ntyp) = elemsingl(itere$_{id,eq(nom(.),ntyp)}$(uni))

struct : EXPR ← UNIVERS,NOM-TYPE

struct(uni,ntyp) =

si appe(type-base,nom(type-expr(antyp)))

alors expr(antyp)

sinon c(construct(expr(antyp)),

iter$_{struct(uni,.),vrai}$(sparam(expr(antyp))))

existe? : BOOL ← UNIVERS,NOM-TYPE

existe?(uni,ntyp) = non(eq(indef,antyp))

issu? : BOOL ← UNIVERS,NOM-TYPE,NOM-TYPE

issu?(uni,ntyp,ntyp') =

equiv?(uni,nom(type-expr(antyp)),nom(type-expr(antyp')))

et inclu?(opimputil(antyp'),opimputil(antyp))

et inclu?(opdefutil(antyp),opdefutil(antyp'))

et invar(antyp)impl invar(antyp')

parent? : BOOL ← UNIVERS,NOM-TYPE,NOM-TYPE

parent?(uni,ntyp,ntyp') =

equiv?(uni,nom(type-expr(antyp)),nom(type-expr(antyp')))

et non(eq(faux,invar(antyp) et invar(antyp')))

equiv? : BOOL ← UNIVERS,NOM-TYPE,NOM-TYPE

equiv?(uni,ntyp,ntyp') =

(eq(struct(uni,ntyp),struct(uni,ntyp'))

ou (eq(construct(struct(uni,ntyp)),construct(struct(uni,ntyp')))

et eq(taille(sparam(struct(uni,ntyp))),

taille(iter2$_{id,equiv?(uni,.,.)}$(sparam(struct(uni,ntyp)),

sparam(struct(uni,ntyp'))))))

et eq(oputil(antyp),oputil(antyp'))

et eq(invar(antyp),invar(antyp'))

$\geqslant$? : BOOL ← UNIVERS,NOM-TYPE,NOM-TYPE

$\geqslant$?(uni,ntyp,ntyp') =

eq(ntyp,ntyp')

ou app(sparam(expr(antyp)),ntyp')

ou infeg(un,taille(iter$_{id,\geqslant?(uni,.,ntyp')}$(sparam(expr(antyp)))))

TYPE PRES-TYPE =
PC[nom : NOM-TYPE, expr : EXPR , invar : PRED,
oputil : OP-UTIL, defopdef : E-DEF-OP]

invariant
non(eq(indef,nom(pres)))
et non(eq(indef,expr(pres))) ou appe(type-base,nom(pres))
et eq(evide,intersec(opimputil(pres),opdefutil(pres)))
et typapprof(profil(defnop(pres,nop)),nom(pres))
et inclu(opdefutil(pres),opdef(pres))

opérations utilisables
importées :
constructeur : c
modificateurs : modif-expr,modif-invar
observateurs : nom,expr,invar,oputil,
defopdef
définies :
constructeurs : deftyp,type-expr,type
modificateurs : adjinvar,supinvar
adjopimp,supopimp
adjopdef,supopdef
observateurs : opimputil,opdefutil
defnop,opdef,oputilisables

variables
pres : PRES-TYPE
ntyp : NOM-TYPE
ex : EXPR
defop: DEF-OP
nop : NOM-OP
pred : PRED

Lexique : Présentations de types

Une présentation est composée d'un nom, d'une expression, d'un invariant, d'opérations utilisables (opérations importées et opérations définies) et de l'ensemble des définitions des opérations définies (utilisables et cachées).

invariant :

Une présentation de type ne peut pas avoir

- un nom indéfini ;
- une expression indéfinie sauf s'il s'agit d'un type de base ;
- deux opérations de même nom.

D'autre part

- le nom de la présentation doit apparaître dans le profil des opérations définies ;
- les opérations définies nommées dans la composante def de la composante oputil doivent être définies dans la composante defopdef.

opérations définies utilisables :

type-expr	: présentation du type défini par une expression de type
deftyp	: définition d'un type à partir d'un nom, d'une expression et d'un invariant
type	: instanciation d'un constructeur de types
adjopdef	: adjonction d'une opération définie utilisable
adjopimp	: adjonction d'une opération importée utilisable
adjinvar	: adjonction d'un invariant
supopdef	: suppression d'une opération définie utilisable
supopimp	: suppression d'une opération importée utilisable
supinvar	: suppression de l'invariant (celui-ci devient vrai)
opdef	: ensemble des opérations définies
opimputil	: ensemble des opérations importées utilisables
opdefutil	: ensemble des opérations définies utilisables
defnop	: accès à la définition d'une opération définie nommée
oputilisables	: ensemble des opérations utilisables

opérations définies cachées :

adjnomopdef	: adjonction du nom d'une opération définie à l'ensemble des noms des opérations définies
adjdefopdef	: adjonction de la définition d'une opération définie à l'ensemble des définitions des opérations définies
supnomopdef	: suppression du nom d'une opération définie de l'ensemble des noms des opérations définies
supdefopdef	: suppression de la définition d'une opération définie de l'ensemble des définitions des opérations définies

définitions

constructeurs

deftyp : PRES-TYPE ← NOM-TYPE,EXPR,PRED

```
deftyp(ntyp,ex,pred) =
        c(ntyp,ex,pred,
           c(opututilisables(type(ex)),evide),evide)
```

type-expr : PRES-TYPE ← PRES-TYPE

```
type-expr(pres) = type(expr(pres))
```

type : PRES-TYPE ← EXPR

```
type(ex) = instanciation(construct(ex),sparam(ex))
```

modificateurs

adjopdef : PRES-TYPE ← PRES-TYPE,DEF-OP

```
adjopdef(pres,defop) = adjnomopdef(adjdefopdef
                                (pres,defop),nom-op(defop))
```

adjopimp : PRES-TYPE ← PRES-TYPE,NOM-OP

```
adjopimp(pres,nop) = modif-oputil(pres,
        modif-imp(opimputil(pres),adj(opimputil (pres),nop)))
```

adjinvar : PRES-TYPE ← PRES-TYPE,PRED

```
adjinvar(pres,pred) = modif-invar(pres,invar(pres)et pred)
```

supopdef : PRES-TYPE ← PRES-TYPE,NOM-OP

```
supopdef(pres,nop) = supnomopdef(supdefopdef(pres,nop),nop)
```

supopimp : PRES-TYPE ← PRES-TYPE,NOM-OP

```
supopimp(pres,nop) = modif-oputil(pres,
        modif-imp(opimputil(pres),supe(opimputil(pres),nop)))
```

supinvar : PRES-TYPE ← PRES-TYPE

```
supinvar(pres) = modif-invar(pres,vrai)
```


observateurs

opimputil : E-NOM-OP ← PRES-TYPE

```
opimputil(pres) = imp(oputil(pres))
```

opdefutil : E-NOM-OP ← PRES-TYPE

```
opdefutil(pres) = def(oputil(pres))
```

opdef : E-NOM-OP ← PRES-TYPE

$opdef(pres) = itere_{nom\text{-}op,vrai}(defopdef(pres))$

oputilisables : E-NOM-OP ← PRES-TYPE

```
oputilisables(pres) = union(opimputil(pres),opdefutil(pres))
```

defnop : DEF-OP ← PRES-TYPE,NOM-OP

```
defnop(pres,nop) = defenop(defopdef(pres),nop)
```

opérations cachées

adjnomopdef : PRES-TYPE ← PRES-TYPE,NOM-OP

```
adjnomopdef(pres,nop) = modif-oputil(pres,modif-def
        (opdefutil(pres),adj(opdefutil(pres),nop)))
```

adjdefopdef : PRES-TYPE ← PRES-TYPE,DEF-OP

```
adjdefopdef(pres,defop) = modif-defopdef(pres,
        adj(defopdef(pres),defop))
```

supnomopdef : PRES-TYPE ← PRES-TYPE,NOM-OP

```
supnomopdef(pres,nop) = modif-oputil(pres,modif-def
        (opdefutil(pres),supe(opdefutil(pres),nop)))
```

supdefopdef : PRES-TYPE ← PRES-TYPE,NOM-OP

```
supdefopdef(pres,nop) = modif-defopdef(pres,
        supe(defopdef(pres),defnop(pres,nop)))
```

TYPE EXPR = PC[construct : NOM-CONSTRUCT,
sparam : S-NOM-TYPE]

lexique : l'expression d'un type est composée d'un constructeur de types instancié par des types paramétres effectifs

TYPE S-NOM-TYPE = S[NOM-TYPE]

lexique : Suite de noms de types

TYPE OP-UTIL = PC[imp : E-NOM-OP,
def : E-NOM-OP]

lexique : Les opérations utilisables d'un type sont composées des opérations importées et des opérations définies

TYPE E-NOM-OP = E[NOM-OP]

lexique : Ensemble de noms d'opérations

TYPE E-DEF-OP = E[DEF-OP]

opérations utilisables	variables
définies : defenop	edefop : E-DEF-OP nop : NOM-OP

lexique : Ensemble de définitions d'opérations
defenop : accès à la définition d'une opération nommée

définition
defenop : DEF-OP ← E-DEF-OP,NOM-OP
defenop(edefop,nop) =
elemsingl(itere$_{id,eq(nom\text{-}op(.),nop)}$(edefop))

TYPE DEF-OP = PC[nom-op : NOM-OP,
profil : PROFIL,
def : E-EQUAT]

invariant	variable
non(eq(indef,nom-op(defop)))	defop : DEF-OP

lexique : Type de la définition d'une opération
invariant : le nom d'une opération ne doit pas être indéfini

TYPE PROFIL = PC[codomaine : NOM-TYPE,
domaine : S-NOM-TYPE]

invariant
non(eq(indef,codomaine(prof)))

opérations utilisables	variables
définie : typapprof	prof : PROFIL ntyp : NOM-TYPE

lexique : Type du profil d'une opération
invariant : le codomaine d'un profil ne doit pas être indéfini
typapprof : prédicat. Un type donné apparaît-il dans le profil ?

définition
typapprof : BOOL ← PROFIL,NOM-TYPE
typapprof(prof,ntyp) =
eq(ntyp,codomaine(prof)) ou app(domaine(prof),ntyp)

TYPE E-EQUAT = E[EQUAT]

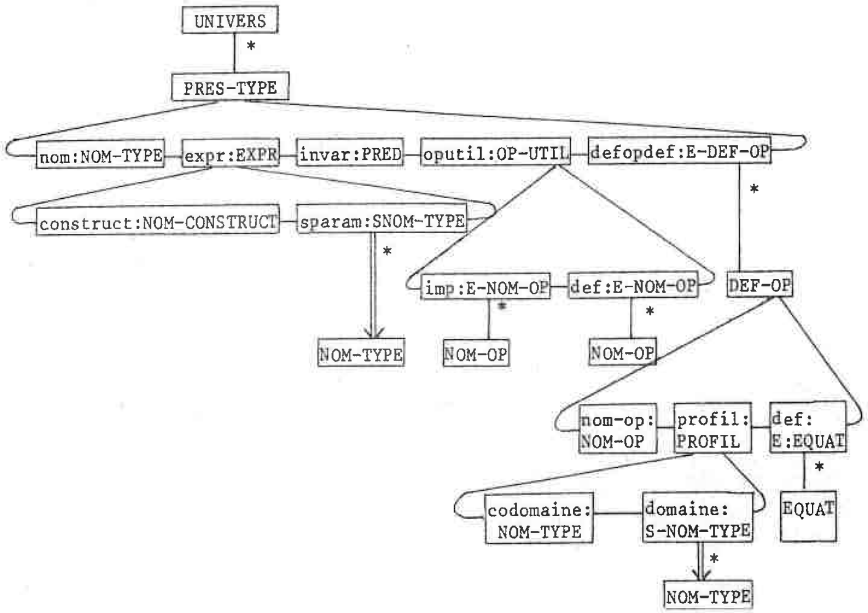
lexique : ensemble d'équations

Les types NOM-TYPE des noms de types
NOM-OP des noms d'opérations
NOM-CONSTRUCT des noms des constructeurs de types
EQUAT des équations
PRED des prédicats

ainsi que les ensembles "type-base" de type E[NOM-TYPE] et
"constructeurs" de type E[NOM-CONSTRUCT]
et l'opération d'instanciation d'un constructeur de types par des paramètres effectifs sont supposés prédéfinis.

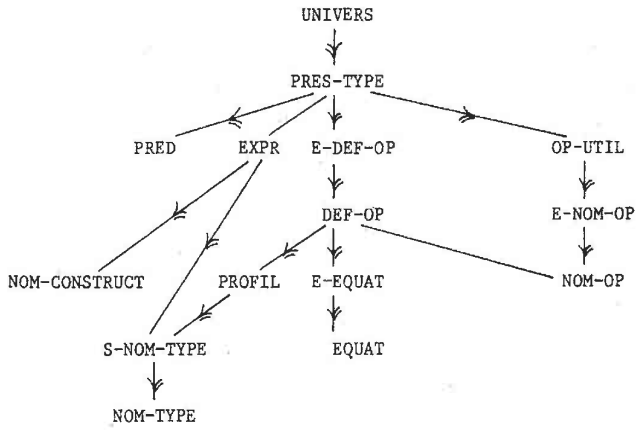
Les types NOM-TYPE, NOM-OP et NOM-CONSTRUCT ont une structure de suite (de caractères).

Graphe de la structure du type univers



3.2.- Transformation de l'Univers spécifiant le système SPES-TYPES

L'Univers est formé des types ordonnés par la relation $\geqslant$ de la manière suivante :



La spécification a été construite de manière descendante sans se préoccuper d'optimisation ni d'évolutivité ou de simplicité.

Une étape de transformation doit nous permettre de rechercher et éventuellement d'éliminer de la redondance : on peut dès maintenant remarquer qu'il y a peut-être redondance puisqu'on retrouve deux fois les mêmes types dans deux composantes distinctes du type PRES-TYPE : le type NOM-OP et le type S-NOM-TYPE.

D'autre part, nous avons donné intuitivement aux types leur structure. Il est nécessaire de vérifier maintenant s'il n'existe pas une structure plus adéquate permettant d'optimiser les définitions des opérations définies. Enfin nous nous préoccupons d'évolutivité : une spécification est d'autant plus apte à évoluer qu'elle est modulaire.

La transformation se fait en quatre grandes étapes :

1.- Simplification :

Dans le but de simplifier la spécification, nous appliquons toutes les

transformations de simplification possibles c'est-à-dire à trois reprises la transformation sp (d'un produit cartésien de n-uplets en un produit cartésien généralisé) sur les types PRES-TYPE et DEF-OP. Nous obtenons un univers de types moins hiérarchiques.

2.- Elimination de la redondance :

A - Dans une structure : Il y a redondance du nom des opérations définies utilisables citées dans une présentation du type

- dans la composante def de la composante oputil d'une part

et - dans la définition de l'opération d'autre part.

Après avoir généralisé le type E-NOM-OP de la composante opdefutil nous fusionnons les composantes opdefutil et defopdef. Remarquons qu'il nous faut ajouter une composante au type DEF-OP. Cette composante appelée utilisable ? est de type booléen. Elle permet de savoir d'une opération définie si elle est utilisable ou non.

B - Entre deux structures : l'univers n'est composé que d'une seule structure.

3.- Optimisation

Nous appliquons la transformation tet sur les types

a - UNIVERS

b - E-DEF-OP

pour leur donner une structure de table :

T[NOM-TYPE] DE PRESSN-TYPE pour le type UNIVERS, et

T[NOM-OP] DE DEFSN-OP pour le type E-DEF-OP.

Les opérations d'accès aux éléments sont optimisées par ces transformations.

4.- Développement

Pour terminer nous réintroduisons les structures simplifiées à l'étape 1 : la composante expr du type PRESSN-TYPE

la composante profil du type DEFSN-OP.

De plus, nous introduisons une sous table au type E-DEF-OP : il aura la structure d'une table d'indice booléen associant à vrai la table des définitions des opérations définies utilisables et à faux la table des définitions des opérations définies cachées.

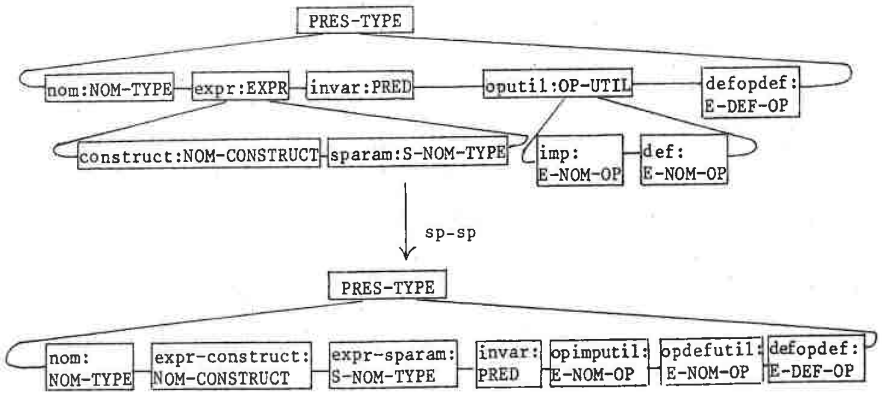
Nous allons étudier ces différentes transformations. Pour ne pas surcharger la lecture, nous ne donnons de la présentation des types transformés que l'expression, l'invariant, l'ensemble des opérations utilisables et la définition des opérations modifiées.

3.2.1.- Simplification

a - Simplification du type PRES-TYPE

Le type PRES-TYPE est un produit cartésien dont les composantes expr et oputil sont de type produit cartésien. On peut donc appliquer 2 fois la transformation sp (d'un produit cartésien de n-uplets en un produit cartésien).

Graphe des transformations :



Remarquons que les opérations définies opdefutil et opimputil du type PRES-TYPE sont des accès aux composantes du sous produit cartésien oputil. Ces deux opérations seront alors importées et non plus définies. La transformée de l'opération importée oputil est une opération définie.

Type PRES-TYPE après transformation :

```
TYPE PRES-TYPE = PC[nom : NOM-TYPE, expr-construct : NOM-CONSTRUCT,
                    expr-sparam : S-NOM-TYPE, invar : PRED,
                    opimputil : E-NOM-OP, opdefutil : E-NOM-OP,
                    defopdef : E-DEF-OP]
```

```
invariant
    nom(eq(indef,nom(pres)))
et  (nom(eq(indef,expr(pres)))
       ou appe(type-base,nom(pres)))
et  eq(evide,intersec(opimputil(pres),opdefutil(pres)))
et  typapprof(profil(defnop(pres,nop)),nom(pres))
et  inclu(opdefutil(pres),opdef(pres))
```

```
opérations utilisables
importées :
constructeur  : c
modificateurs : modif-expr-construct,modif-expr-
                sparam,modif-invar
observateurs  : nom,expr-construct,expr-sparam,
                invar,opimputil,opdefutil,
                defopdef
définies :
constructeurs : deftyp,type-expr,type
modificateurs : modif-expr,
                adjinvar,adjopimp,adjopdef,
                supinvar,supopimp,supopdef
observateurs  : defnop,expr,oputil,opdef,
                oputilisables
```

```
variables
pres  : PRES-TYPE
ntyp  : NOM-TYPE

nop   : NOM-OP
defop : DEF-OP
ex    : EXPR
pred  : PRED
```

```
définitions
constructeurs :
    deftyp : PRES-TYPE ← NOM-TYPE,EXPR,PRED
       deftyp(ntyp,ex,pred) =
              c(ntyp,construct(ex),sparam(ex),pred,
                 oputilisables(type(ex)),evide,evide)
```



```
    type-expr : PRES-TYPE ← PRES-TYPE
    ...
    type : PRES-TYPE ← EXPR
    ...

modificateurs
    modif-expr : PRES-TYPE ← PRES-TYPE,EXPR
       modif-expr(pres,ex) = modif-expr-construct(modif-expr-sparam
                              (pres,sparam(ex)),construct(ex))
    adjinvar : PRES-TYPE ← PRES-TYPE,PRED
    ...
    adjopimp : PRES-TYPE ← PRES-TYPE,NOM-OP
       adjopimp(pres,nop) = modif-opimputil(pres,adj(opimputil(pres,nop)))
    adjopdef : PRES-TYPE ← PRES-TYPE,DEF-OP
    ...
    supinvar : PRES-TYPE ← PRES-TYPE
    ...
    supopimp : PRES-TYPE ← PRES-TYPE,NOM-OP
       supopimp(pres,nop) = modif-opimputil(pres,
                             supe(opimputil(pres),nop))
    supopdef : PRES-TYPE ← PRES-TYPE,NOM-OP
    ...
observateurs
    defnop : DEF-OP ← PRES-TYPE,NOM-OP
    ...
    expr : EXPR ← PRES-TYPE
       expr(pres) = c(expr-construct(pres),expr-sparam(pres))
    oputil : OP-UTIL ← PRES-TYPE
       oputil(pres) = c(opimputil(pres),opdefutil(pres))
    opdef : E-NOM-OP ← PRES-TYPE
    ...
    oputilisables : E-NOM-OP ← PRES-TYPE
    ...
```

opérations cachées

adjnomopdef : PRES-TYPE ← PRES-TYPE,NOM-OP

adjnomopdef(pres,nop) = modif-opdefutil(pres,
adj(opdefutil(pres),nop))

adjdefopdef : PRES-TYPE ← PRES-TYPE,DEF-OP

...

supnomopdef : PRES-TYPE ← PRES-TYPE,NOM-OP

supnomopdef(pres,nop) = modif-opdefutil(pres,supe
(opdefutil(pres),nop))

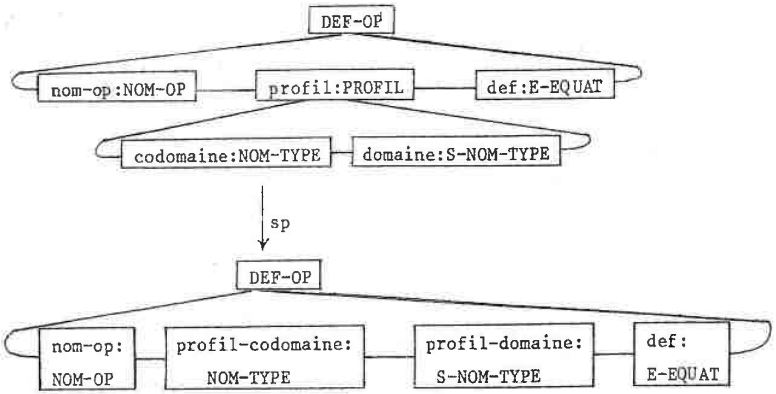
supdefopdef : PRES-TYPE ← PRES-TYPE,NOM-OP

...

b - Simplification du type DEF-OP

Le type DEF-OP est un produit cartésien dont la composante profil est de type produit cartésien. Nous appliquons la transformation sp :

Graphe de la transformation :



Remarques

- Le type PROFIL n'apparaît plus dans le type DEF-OP qui hérite de son invariant.
- Le type DEF-OP n'avait pas d'opérations définies. Les opérations modif-profil et profil étaient importées. Dans le type transformé elles sont définies.

Type DEF-OP après transformation :

TYPE DEF-OP = PC[nom-op:NOM-OP,profil-codomaine:NOM-TYPE,
profil-domaine:S-NOM-TYPE, def:E-EQUAT]

invariant :

non(eq(indef,nom-op(defop))) et non(eq(indef,profil-codomaine(defop)))

opérations utilisables

définies :

modificateurs : modif-profil

observateur : profil

variables

defop : DEF-OP

prof : PROFIL

définitions :

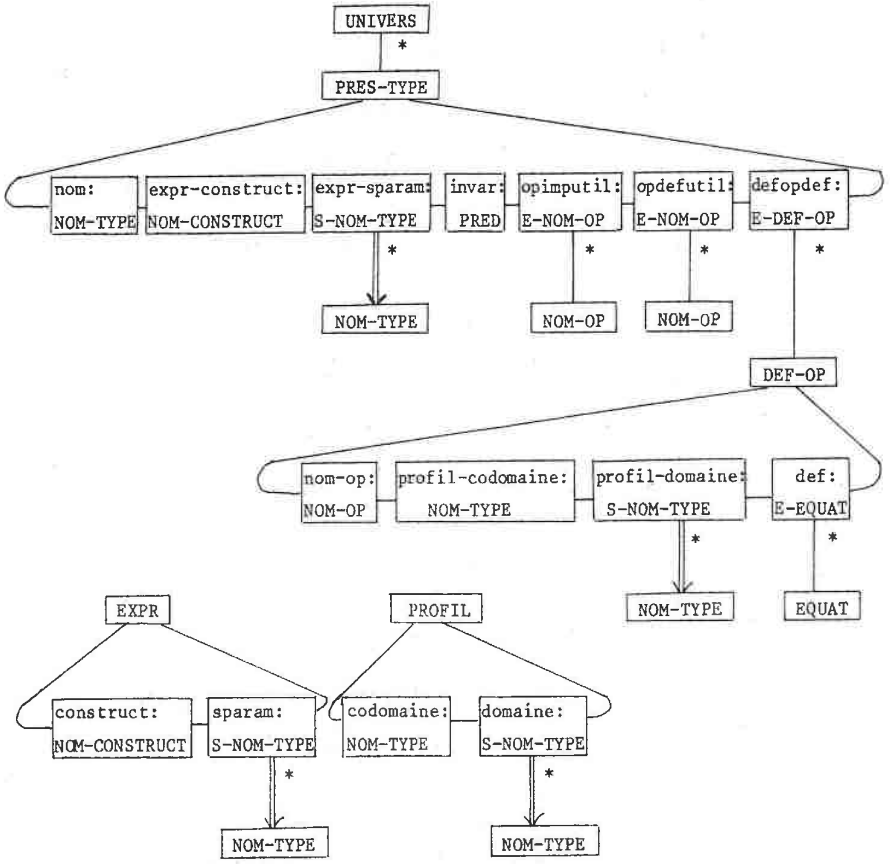
modif-profil : DEF-OP ← DEF-OP,PROFIL

modif-profil(defop,prof)
= modif-profil-domaine(modif-profil-codomaine(defop,
codomaine(prof)),domaine(prof))

profil : PROFIL ← DEF-OP

profil(defop) = c(profil-codomaine(defop),
profil-domaine(defop))

Graphe de l'univers à la fin de l'étape de simplification



Remarque :

Les types EXPR et PROFIL appartiennent à l'univers car ils apparaissent dans le profil de certaines opérations.

3.2.2.- Elimination de la redondance

A - Dans uns structure

La recherche de redondance dans une structure se fait en comparant 2 à 2 les types des différentes composantes, en commençant par les types les plus grands pour la relation $\succcurlyeq$. Examinons le type PRES-TYPE.

Nous remarquons que :

a - Les composantes opimputil et opdefutil sont de même type. Mais il n'y a pas redondance de l'information. En effet l'invariant du type PRES-TYPE précise qu'un type ne peut comporter deux opérations de même nom et l'ensemble des opérations d'un type est égal à l'union des ensembles des composantes opimputil et opdefutil. Pour fusionner ces deux composantes il faudrait adjoindre à chaque nom d'opération sa caractéristique importée ou définie.

b - Les composantes opimputil et defoputil sont ressemblantes mais pour la même raison que ci-dessus, il n'y a pas de redondance de l'information.

c - Les composantes opdefutil et defopdef sont ressemblantes : NOM-OP peut être généralisé par la transformation g en un type de structure équivalente à celle de DEF-OP. Dans ce cas il y a effectivement redondance de l'information : l'invariant du type PRES-TYPE précise que toute opération nommée dans la composante opdefutil doit être définie dans la composante defopdef. Il faut cependant remarquer que la composante defopdef comporte de plus les définitions des opérations cachées. Fusionner les deux composantes nécessite d'adjoindre à chaque définition d'opération sa caractéristique utilisable ou non.

Nous avons donc détecté un cas de redondance. La fusion des composantes opdefutil et defopdef permet de l'éliminer. Pour cela il faut :

1. premièrement généraliser DEF-OP en ajoutant une composante que nous appelons utilisable ?, de type booléen, permettant de savoir d'une opération si elle est utilisable ou non : une opération est utilisable si son nom appartient à l'ensemble opdefutil.

2. Dans un deuxième temps, il faut généraliser le type E-NOM-OP de la composante opdefutil pour lui donner une structure équivalente à celle du type DEF-OP de la composante defopdef.

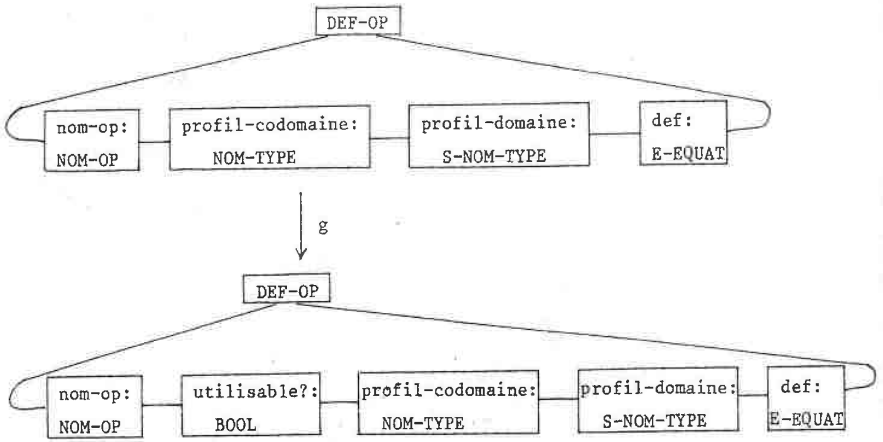
3. La troisième étape est celle de la fusion des deux composantes. On peut dire que la composante defopdef "absorbe" la composante opdefutil car le type de la première est issu du type de la seconde.

4. Enfin il faut répercuter cette transformation sur le type PRES-TYPE. Il comporte une composante de moins mais une opération définie de plus : l'opération opdefutil d'accès à l'ensemble des noms d'opérations définies utilisables. D'autre part il devient inutile de préciser dans l'invariant que toute opération de opdefutil doit être définie dans defopdef.

Fusion des composantes opdefutil et defopdef de PRES-TYPE

1 - Généralisation de DEF-OP

Graphe de la transformation :



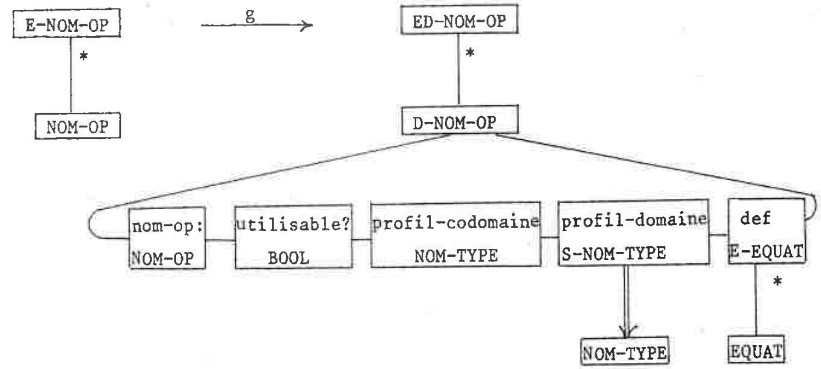
Type DEF-OP après la transformation

TYPE DEF-OP = PC[nom-op : NOM-OP, utilisable? : BOOL, profil-codomaine : NOM-TYPE, profil-domaine :S-NOM-TYPE, def : E-EQUAT]	
invariant : non(eq(indef,nom-op(defop))) et non(eq(indef,profil-codomaine(defop)))	
opérations utilisables définies : modificateur : modif-profil observateur : profil	variables defop : DEF-OP

Remarque : Les définitions des opérations modif-profil et profil n'ont pas été modifiées.

2 - Généralisation de E-NOM-OP

Graphe de la transformation



TYPE ED-NOM-OP = E[D-NOM-OP]
lexique : ensemble de définitions d'opérations

TYPE D-NOM-OP = PC[nom-op : NOM-OP, utilisable? : BOOL, profil-codomaine : NOM-TYPE, profil-domaine : S-NOM-TYPE, def : E-EQUAT]
lexique : définition d'une opération

3 - Fusion des composantes opdefutil de type ED-NOM-OP et defopdef de type E-DEF-OP du type PRES-TYPE : Les types D-NOM-OP et DEF-OP fusionnent. DEF-OP étant issu de D-NOM-OP, leur fusion est égale à DEF-OP.
Les types ED-NOM-OP et E-DEF-OP fusionnent. E-DEF-OP étant issu de ED-NOM-OP, leur fusion est égale à E-DEF-OP.

4 - Répercussion sur le type PRES-TYPE :
Il comporte une composante de moins. L'opération opdefutil est alors définie. Et son invariant ne précise plus que toute opération de opdefutil doit être définie dans defopdef.

Type PRES-TYPE après transformation :

TYPE PRES-TYPE = PC[nom : NOM-TYPE, expr-construct : NOM-CONSTRUCT, expr-sparam : S-NOM-TYPE, invar : PRED, opimputil : E-NOM-OP, defopdef : E-DEF-OP]
invariant : nom(eq(indef,nom(pres))) et (non(eq(indef,expr(pres))) ou appe(type-base,nom(pres))) et eq(evide,intersec(opimutil(pres),opdefutil(pres))) et typapprof(profil(defnop(pres,nop)),nom(pres))

opérations utilisables	variables
importées : constructeur : c modificateur : modif-expr-construct,modif-expr-sparam,modif-invar observateurs : nom,expr-construc,expr-sparam, invar,opimputil,defopdef définies : constructeurs : deftyp,type-expr,type modificateurs : modif-expr,adjinvar,adjopimp, adjopdef,supinvar,supopimp, supopdef observateurs : defnop,expr,oputil,opdefutil, opdef,oputilisables	pres : PRES-TYPE nop : NOM-OP defop : DEF-OP ex : EXPR pred : PRED ntyp : NOM-TYPE

définitions
constructeurs :

```
deftyp : PRES-TYPE ← NOM-TYPE,EXPR,PRED
   deftyp(ntyp,ex,pred) =
      c(ntyp,construct(ex),sparam(ex),pred,
        oputilisables(type(ex)),evide)

type-expr : PRES-TYPE ← PRES-TYPE
...
type : PRES-TYPE ← EXPR
...
```

modificateurs :

```
modif-expr : PRES-TYPE ← PRES-TYPE,EXPR
...
adjinvar   : PRES-TYPE ← PRES-TYPE,PRED
...
adjopimp   : PRES-TYPE ← PRES-TYPE,NOM-OP
...
adjopdef   : PRES-TYPE ← PRES-TYPE,DEF-OP
   adjopdef(pres,defop) = modif-defopdef(pres,
          adj(defopdef(pres),defop))
```

```
    supinvar      : PRES-TYPE ← PRES-TYPE
    ...
    supopimp      : PRES-TYPE ← PRES-TYPE,NOM-OP
    ...
    supopdef      : PRES-TYPE ← PRES-TYPE,NOM-OP
       supopdef(pres,nop) = modif-defopdef(pres,
              supe(defopdef(pres),defnop(pres,nop)))
observateurs :
    defnop        : DEF-OP ← PRES-TYPE,NOM-OP
    ...
    expr          : EXPR ← PRES-TYPE
    ...
    oputil        : OP-UTIL ← PRES-TYPE
    ...
    opdefutil     : E-NOM-OP ← PRES-TYPE
       opdefutil(pres) = eopdefutil(defopdef(pres))
    opdef         : E-NOM-OP ← PRES-TYPE
    ...
    oputilisables : E-NOM-OP ← PRES-TYPE
    ...
```

Remarque : Le type PRES-TYPE ne comportant plus la composante opdefutil, les opérations d'adjonction adjopdef et de suppression supopdef d'une opération définie sont simplifiées : il suffit de modifier la composante defopdef. Les opérations qui réalisaient ces modifications, adjdefopdef et supdefopdef, étaient des opérations cachées. Après transformation adjopdef est égale à adjdefopdef et supopdef à supdefopdef. Nous avons alors simplifié la spécification en supprimant ces deux opérations cachées. Les opérations cachées adjnomopdef et supnomopdef qui modifiaient la composante opdefutil disparaissent elles aussi.

Nous avons utilisé dans la définition de opdefutil une nouvelle opération qui enrichit le type E-DEF-OP dont voici la nouvelle présentation :

TYPE E-DEF-OP = E[DEF-OP]	
opérations utilisables définies : defenop eopdefutil	variables edefop : E-DEF-OP
lexique : Ensemble de définitions d'opérations defenop : définition d'une opération nommée eopdefutil : ensemble des opérations utilisables	
définitions defenop : DEF-OP ← E-DEF-OP,NOM-OP ... eopdefutil : E-NOM-OP ← E-DEF-OP eopdefutil(edefop) = $itere_{nom-op,utilisable?}$(edefop)	

Nous avons éliminé la redondance qui existait entre deux composantes du type PRES-TYPE. En recherchant dans PRES-TYPE d'autres types ressemblants appartenant à des composantes d'un même produit cartésien, on ne trouve que les types NOM-TYPE et S-NOM-TYPE des composantes profil-codomaine et profil-domaine de DEF-OP. Ces deux composantes n'ont pas d'information en commun. Il n'y a pas ici de redondance. Nous avons donc terminé avec l'élimination de la redondance dans la structure UNIVERS.

L'Univers comporte en outre les types EXPR et PROFIL.

Ces deux types ne comportent pas de composantes redondantes.

B - Entre deux structures

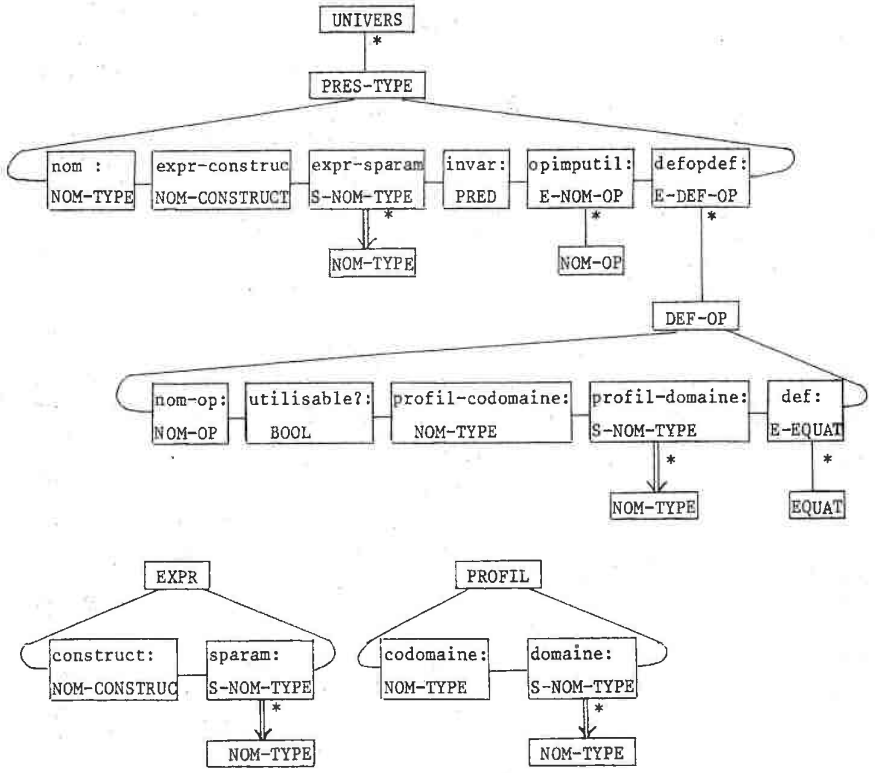
L'Univers est composé d'une structure unique : le type UNIVERS. Les types EXPR et PROFIL ne peuvent pas lui être comparés. Ce sont des types auxiliaires présents dans l'Univers parce qu'apparaissant dans le profil de cer-

taines opérations.

On peut d'autre part remarquer qu'il y a redondance de type entre les deux composantes expr-construct et expr-sparam d'une part et le type EXPR de l'autre. Mais ça nous le savions bien sûr : les deux composantes étant la transformée par simplification de la composante de type EXPR initiale. Il en est de même pour les deux composantes profil-codomaine et profil-domaine d'une part et le type PROFIL de l'autre.

Nous n'avons donc aucune élimination de redondance entre deux structures.

Graphe de l'Univers après l'étape d'élimination de la redondance



3.2.3.- Optimisation

Pour optimiser les opérations définies des types de l'univers nous les transformons à l'aide des transformations de structure. En effet nous avons vu qu'un type vérifiant les critères d'application d'une transformation voit la définition de ses opérations optimisée. Nous étudions les différents types de l'univers, en partant des plus grands pour la relation $\succcurlyeq$. Pour chacun d'eux nous examinons les différentes transformations de structure applicables.

a - TYPE UNIVERS

Le type UNIVERS a la structure d'un ensemble de sextuplets. Il peut être transformé :

- en une suite par la transformation tes. Le type UNIVERS n'utilise pas d'opération ensembliste. Cette transformation pourrait se faire en imposant l'ordre d'adjonction des types dans l'univers comme ordre de la suite, mais cet ordre n'a pas grande signification.
- en une table associant à un nom de type, sa présentation. Le type UNIVERS satisfait les critères d'application de la transformation tet : la fonction accty, très utilisée, est définie comme étant le seul élément de l'ensemble $itere_{id,eq(clé(.),p)}$ où la clé est la composante nom.
  D'autre part le type UNIVERS n'utilise pas les opérations ensemblistes.
- le type UNIVERS ne satisfait pas les conditions d'application de la transformation tep en un produit cartésien d'ensembles : en effet il utilise la relation existant entre les composantes du sextuplet, ne serait-ce que le nom du type.

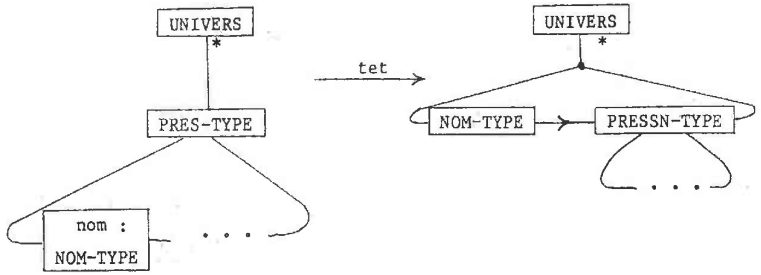
Nous appliquons la transformation tet sur le type UNIVERS. La clé choisie est le nom du type. Nous sommes dans un cas particulier de la transformation tet. En effet, d'après l'invariant du type UNIVERS, à chaque nom n'est associé qu'un seul type. Le type cible aura donc la structure d'une table associant à un nom de type, une présentation (et non un ensemble de présentations). L'invariant du type UNIVERS devient donc superflu : par la structure même du type, il sera toujours vérifié.

La présentation associée à un nom ne comportera pas de composante nom. Nous appelons PRESSN-TYPE le type de cette présentation-sans-nom. Cependant c'est encore le type PRES-TYPE qui apparaît dans le profil des opérations. PRES-TYPE n'est pas modifié et l'univers s'enrichit du type PRESSN-TYPE défini par :

```
TYPE PRESSN-TYPE = PC[expr-construct : NOM-CONSTRUCT,
                      expr-sparam    : S-NOM-TYPE,
                      invar          : PRED,
                      opimputil      : E-NOM-OP,
                      defopdef       : E-DEF-OP]
```

lexique : présentation-sans-nom d'un type

Graphe de la transformation



Etudions la transformation des opérations définies du type UNIVERS :

constructeurs :

uvide : UNIVERS ←
uvide = evide

tet(uvide) : tet(UNIVERS) ←
tet(uvide) = tet(evide) = tvide

adjtyp : UNIVERS ← UNIVERS,PRES-TYPE
adjtyp(uni,pres) = adj(uni,pres)

tet(adjtyp) : tet(UNIVERS) ← tet(UNIVERS),PRES-TYPE
tet(adjtyp)(tet(uni),pres) = tet(adj)(tet(uni),pres)
= si appt(tet(uni),nom(pres))
alors mod(tet(uni),nom(pres),c(expr-construc(pres),
expr-sparam(pres),invar(pres),opimputil(pres),
defopdef(pres)))
sinon insert(tet(uni),nom(pres),c(expr-construct(pres),
expr-sparam(pres),invar(pres),opimputil(pres),
defopdef(pres)))

modificateur :

suptyp : UNIVERS ← UNIVERS,NOM-TYPE
suptyp(uni,ntyp) = supe(uni,acctyp(uni,ntyp))
tet(suptyp) : tet(UNIVERS) ← tet(UNIVERS),NOM-TYPE
tet(suptyp)(tet(uni),ntyp) = tet(supe)(tet(uni),tet(acctyp)(tet(uni),ntyp))
= tet(supe)(tet(uni),c(ntyp,acct(tet(uni),ntyp)))
= supt(tet(uni),ntyp)

observateurs :

acctyp : PRES-TYPE ← UNIVERS,NOM-TYPE
acctyp(uni,ntyp) = elemsingl(itere$_{id,eq(nom(.),ntyp)}$(uni))

tet(acctyp) : PRES-TYPE ← tet(UNIVERS),NOM-TYPE
tet(acctyp)(tet(uni),ntyp)
= tet(elemsingl(itere))$_{id,eq(nom(.),ntyp)}$(tet(uni))
= c(ntyp,acct(tet(uni),ntyp))

existe? : BOOL ← UNIVERS,NOM-TYPE
existe?(uni,ntyp) = non(eq(indef,acctyp(uni,ntyp)))
tet(existe?) : BOOL ← tet(UNIVERS),NOM-TYPE
tet(existe?)(tet(uni),ntyp) = non(eq(indef,acct(tet(uni),ntyp)))
= appt(tet(uni),ntyp)

Les définitions des autres opérations de UNIVERS ne sont pas modifiées parce qu'elles sont la composée d'autres opérations. Remarquons que la définition de acctyp a été optimisée : c'est un accès direct au lieu d'une ité-

ration. Cette opération étant très utilisée, le type UNIVERS bénéficie de la transformation.

D'autre part nous définissons les transformations des noms :

tet(UNIVERS) = UNIVERS

et $\forall op \in opdefutil(UNIVERS)$, tet(op) = op.

Type UNIVERS obtenu par transformation :

TYPE UNIVERS = T[NOM-TYPE] de PRESSN-TYPE

opérations utilisables

importées : aucune

définies :

constructeurs : uvide,adjtyp

modificateur : suptyp

observateurs : acctyp,struct,existe?
issu?, parent?,equiv?, $\geqslant$?

variables

uni : UNIVERS

pres : PRES-TYPE

ntyp,ntyp' : NOM-TYPE

lexique : univers de types de structure égale à une table associant à un nom de type, sa présentation

définitions

```
uvide : UNIVERS ←
   uvide = tvide
adjtyp : UNIVERS ← UNIVERS,PRES-TYPE
   adjtyp(uni,pres) =
         = si existe?(uni,nom(pres))
             alors mod(uni,nom(pres),c(expr-construct(pres),
                   expr-sparam(pres),invar(pres),opimputil(pres),
                    defopdef(pres)))
             sinon insert(uni,nom(pres),c(expr-construct(pres),
                   expr-sparam(pres),invar(pres),opimputil(pres),
                   defopdef(pres)))
suptyp : UNIVERS ← UNIVERS,NOM-TYPE
   suptyp(uni,ntyp) = supt(uni,ntyp)
```



```
acctyp : PRES-TYPE ← UNIVERS,NOM-TYPE
   acctyp(uni,ntyp) = c(ntyp,acct(uni,ntyp))
struct : EXPR ← UNIVERS,NOM-TYPE ...
existe? : BOOL ← UNIVERS,NOM-TYPE
   existe?(uni,ntyp) = appt(uni,ntyp)
issu? : BOOL ← UNIVERS,NOM-TYPE ...
parent? : BOOL ← UNIVERS,NOM-TYPE ...
equiv? : BOOL ← UNIVERS,NOM-TYPE ...
≥ ? : BOOL ← UNIVERS,NOM-TYPE ...
```

b - TYPE PRESSN

Le type PRESSN a la structure d'un produit cartésien. Aucune transformation de structure ne lui est applicable.

c - TYPE E-DEF-OP

Le type E-DEF-OP a la structure d'un ensemble de quadruplets. Il peut être transformé :

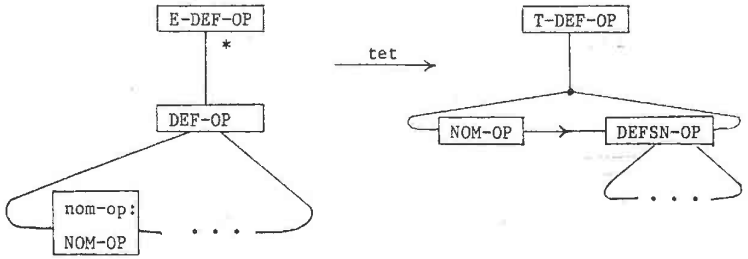
- en une suite par la transformation tes. Le type E-DEF-OP n'utilise pas d'opération ensembliste. Cette transformation pourrait se faire en imposant l'ordre d'adjonction des définitions des opérations comme ordre de la suite, mais cet ordre n'a pas grande signification.
- en une table associant à un nom d'opération, sa définition. Le type E-DEF-OP satisfait les critères d'application de la transformation tet : la fonction defenop est l'accès à la définition d'une opération nommée. D'autre part le type E-DEF-OP n'utilise pas les opérations ensemblistes.
- le type E-DEF-OP ne satisfait pas les conditions d'application de la transformation tep en un produit cartésien d'ensembles : en effet il utilise la relation existant entre les composantes du quadruplet, ne serait-ce que le nom de l'opération.

Nous appliquons la transformation tet sur le type E-DEF-OP. La clé choisie est le nom de l'opération. Comme pour la transformation du type UNIVERS, nous sommes dans le cas particulier où d'après l'invariant du type PRES-TYPE, à chaque nom n'est associée qu'une seule définition d'opération. Le type cible aura donc la structure d'une table associant à un nom d'opération, sa définition (et non un ensemble de définitions). Il est nécessaire de mettre à jour l'invariant du type PRES-TYPE. La définition associée à un nom d'opération ne comportera pas le nom de l'opération, elle sera donc du type DEFSN-OP, type qui enrichit l'univers.

TYPE DEFSN-OP = PC[utilisable? : BOOL, profil-codomaine : NOM-TYPE, profil-domaine : S-NOM-TYPE, def : E-EQUAT]
lexique : définition-sans-nom- d'une opération

Ce type ne comporte pas l'invariant du type DEF-OP : cet invariant portait sur la composante nom-op.

Graphe de la transformation



Le type E-DEF-OP comporte deux opérations définies :

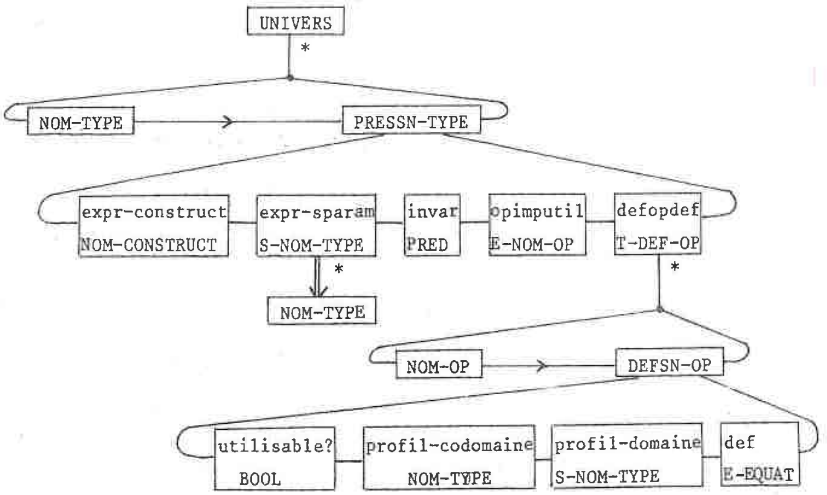
- eopdefutil qui construit l'ensemble des opérations utilisables. Cette opération est transformée en une opération parcourant la table en ne conservant que l'indice (le nom) de l'opération utilisable.
- defenop qui est l'accès à la définition d'une opération nommée. Cette opération est optimisée par la transformation puisque d'un parcourt d'ensemble elle devient accès direct.

TYPE T-DEF-OP = T[NOM-OP] DE DEFSN-OP	
opérations utilisables définies : eopdefutil,defenop	variables tdefop : T-DEF-OP nop : NOM-OP
lexique : Table associant à un nom d'opération, sa définition (sans le nom de l'opération)	
définitions eopdefutil : E-NOM-OP ↛ T-DEF-OP eopdefutil(tdefop) = $itert_{indice,utilisable?}(tdefop)$ defenop : DEF-OP ↛ T-DEF-OP,NOM-OP defenop(tdefop,nop) = c(nop,acct(tdefop,nop))	

d - TYPE DEFSN-OP

Le type DEFSN-OP a la structure d'un produit cartésien. Aucune transformation de structure ne lui est applicable.

Graphe du type UNIVERS après l'étape de transformation de structure



L'univers comprend de plus les types suivants :
PRES-TYPE, DEF-OP, EXPR, PROFIL.

3.2.4.- Développement

Pour rendre une spécification plus apte à évoluer, nous augmentons la hiérarchie dans les structures. Pour cela nous appliquons les transformations de développement applicables. Parmi les simplifications que nous avons faites, certaines nous ont permi des transformations ultérieures ; par exemple, l'élimination de la redondance créée par l'ensemble des noms d'opérations utilisables définies. D'autres n'ont pas entrainé de transformations. Nous allons pour ces dernières, réintroduire les structures hiérarchiques "aplaties". Etudions l'univers en partant des types les plus grands pour la relation $\succcurlyeq$.

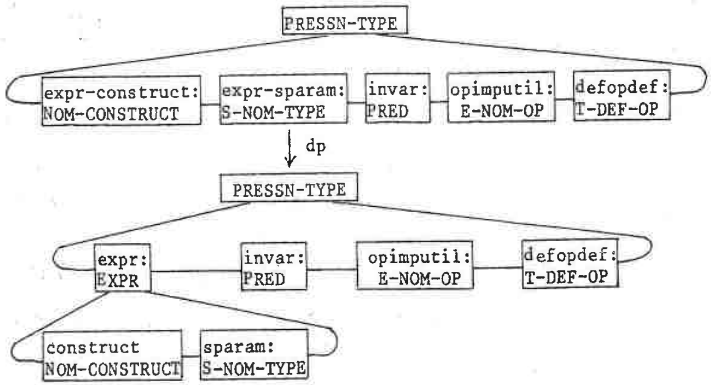
a - TYPE UNIVERS

Le type UNIVERS a une structure d'ensemble. Il n'existe pas de critère général permettant de regrouper certaines présentations d'un univers donné.

b - TYPES PRESSN-TYPE et PRES-TYPE

Ces deux types ont des structures de produits cartésiens. La transformation inverse de la simplification appliquée au début du processus de transformation, est applicable sur les deux types : nous réintroduisons la composante expression regroupant le nom du constructeur et la suite des paramètres.

Graphe de la transformation du type PRESSN-TYPE



Le type PRES-TYPE est transformé de la même manière.

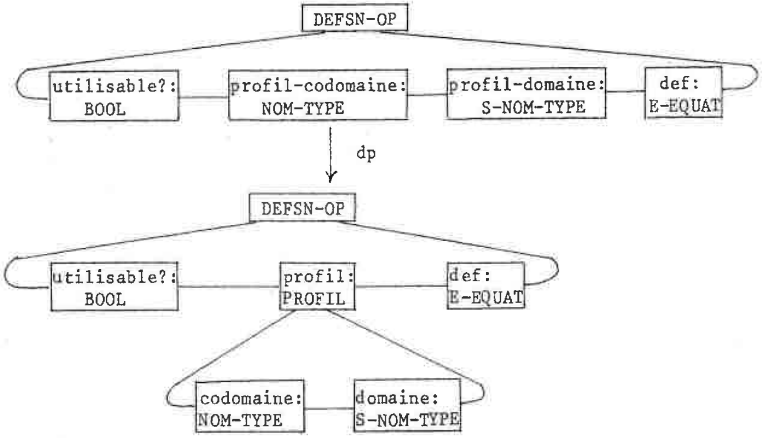
c - TYPE T-DEF-OP

La transformation dt d'une table en une table de tables, n'est pas applicable sur le type T-DEF-OP.

d - TYPES DEFSN-OP et DEF-OP

Ces deux types ont des structures de produits cartésiens. La transformation inverse de la simplification appliquée au début du processus de transformation, est applicable sur les deux types : nous réintroduisons la composante profil regroupant le codomaine et le domaine de la définition d'une opération.

Graphe de la transformation du type DEFSN-OP



Le type DEF-OP est transformé de la même manière.

e - TYPE E-EQUAT

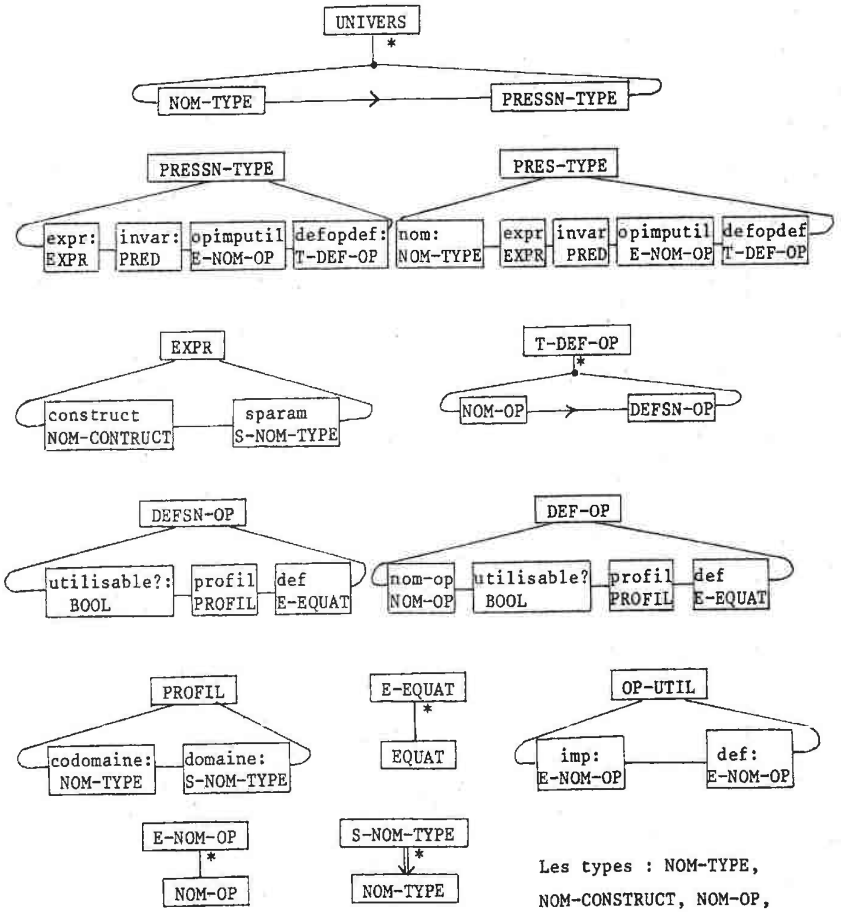
Le type E-EQUAT a une structure d'ensemble. Mais généralement cet ensemble sera composé de deux ou trois équations. Ce qui ne justifie pas l'introduction d'une nouvelle sous-structure.

f - TYPE E-NOM-OP

Le type E-NOM-OP a une structure d'ensemble. Il n'existe pas de critère général permettant de regrouper certains noms d'opérations, même si dans certains cas un tel critère peut exister.

Ces transformations terminent le processus de transformation de l'univers dont voici la spécification résultante.

Graphe de l'univers après transformation



Les types : NOM-TYPE, NOM-CONSTRUCT, NOM-OP, EQUAT, PRED sont prédéfinis.

3.3.- Spécification de l'univers après transformation

TYPE UNIVERS = T[NOM-TYPE] de PRESSN-TYPE

opérations utilisables

importées : aucune

définies :

constructeurs : uvide,adjtyp

modificateurs : suptyp

observateurs : acctyp,struct,existe?
issu?,parent?
equiv?, $\geqslant$?

variables

uni : UNIVERS
pres : PRES-TYPE
ntyp,ntyp' : NOM-TYPE
antyp,antyp' :
PRESSN-TYPE

lexique : univers de types, de structure table associant à un nom de type, sa présentation

définitions

constructeurs

uvide : UNIVERS ←

uvide = tvide

adjtyp : UNIVERS ← UNIVERS,PRES-TYPE

adjtyp(uni,pres) =

si existe?(uni,nom(pres))

alors mod(uni,nom(pres),c(expr(pres),
invar(pres),opimputil(pres),defopdef(pres)))

sinon insert(uni,nom(pres),c(expr(pres),invar(pres),
opimputil(pres),defopdef(pres)))

modificateurs

suptyp : UNIVERS ← UNIVERS,NOM-TYPE

suptyp(uni,ntyp) = supt(uni,ntyp)

observateurs : posons antyp = acct(uni,ntyp) et antyp' = acct(uni,ntyp')

acctyp : PRES-TYPE ← UNIVERS,NOM-TYPE

acctyp(uni,ntyp) = c(ntyp,expr(antyp),invar(antyp),
opimputil(antyp),defopdef(antyp))

```
struct : EXPR ← UNIVERS,NOM-TYPE
   struct(uni,ntyp) = si appe(type-base,nom(type-expr(antyp)))
                      alors expr(antyp)
                      sinon c(construct(expr(antyp)),
                       iter_struct(uni,.),vrai(sparam(expr(antyp))))
existe? : BOOL ← UNIVERS,NOM-TYPE
   existe?(uni,ntyp) = appt(uni,ntyp)
issu? : BOOL ← UNIVERS,NOM-TYPE,NOM-TYPE
   issu?(uni,ntyp,ntyp') =
         equiv?(uni,nom(type-expr(antyp)),
                    nom(type-expr(antyp')))
et       inclu?(opimputil(antyp'),opimputil(antyp))
et       inclu?(opdefutil(antyp),opdefutil(antyp'))
et       invar(antyp) impl invar(antyp')
parent? : BOOL ← UNIVERS,NOM-TYPE,NOM-TYPE
   parent?(uni,ntyp,ntyp') =
         equiv?(uni,nom(type-expr(antyp)),
                    nom(type-expr(antyp')))
et       non(eq(faux,invar(antyp) et invar(antyp')))
equiv? : BOOL ← UNIVERS,NOM-TYPE,NOM-TYPE
   equiv?(uni,ntyp,ntyp') =
         (eq(struct(uni,ntyp),struct(uni,ntyp'))
ou       (eq(construct(struct(uni,ntyp)),
             construct(struct(uni,ntyp')))
et       eq(taille(sparam(struct(uni,ntyp))),
            taille(iter2_id,equiv?(uni,.,.)(sparam(struct(uni,ntyp)),
                                            sparam(struct(uni,ntyp'))))))
et       eq(oputil(antyp),oputil(antyp'))
et       eq(invar(antyp),invar(antyp'))
≽ ? : BOOL ← UNIVERS,NOM-TYPE,NOM-TYPE
   ≽?(uni,ntyp,ntyp') = eq(ntyp,ntyp') ou
         app(sparam(expr(antyp)),ntyp') ou
         infeg(un,taille(iter_id,≽?(uni,.,ntyp')(sparam(expr(antyp)))))
```



```
TYPE PRESSN-TYPE = PC[expr : EXPR, invar : PRED,
                      opimputil : E-NOM-OP, defopdef : T-DEF-OP]
lexique : présentation - sans nom d'un type
```

```
TYPE PRES-TYPE = PC[nom : NOM-TYPE, expr : EXPR, invar : PRED,
                    opimputil : E-NOM-OP, defopdef : T-DEF-OP]
invariant : nom(eq(indef,nom(pres)))
   et       (nom(eq(indef,expr(pres)))
      ou       appe(type-base,nom(pres)))
   et       eq(evide,intersec(opimputil(pres),opdefutil(pres)))
   et       typapprof(profil(defnop(pres,nop)),nom(pres))
lexique : Type des présentations de types (avec la composante nom).
opérations utilisables
importées
   constructeur   : c
   modificateurs  : modif-expr,modif-invar
   observateurs   : nom,expr,invar,opimputil,
                    defopdef
définies
   constructeurs  : deftyp,type-expr,type
   modificateurs  : adjinvar,adjopimp,adjopdef,
                    supinvar,supopimp,supopdef
   observateurs   : defnop,opdefutil,oputil,
                    opdef,oputilisables
variables
   pres  : PRES-TYPE
   nop   : NOM-OP
   ex    : EXPR
   ntyp  : NOM-TYPE
   pred  : PRED
   defop : DEF-OP
définitions
   deftyp : PRES-TYPE ← NOM-TYPE,EXPR,PRED
      deftyp(ntyp,ex,pred) = c(ntyp,ex,pred,oputilisables(type(ex)),
                               evide)
   type-expr : PRES-TYPE ← PRES-TYPE
      type-expr(pres) = type(expr(pres))
```

```
type : PRES-TYPE ← EXPR
   type(ex) = instanciation(construct(ex),sparam(ex))
adjinvar : PRES-TYPE ← PRES-TYPE,PRED
   adjinvar(pres,pred) = modif-invar(pres,invar(pres) et pred)
adjopimp : PRES-TYPE ← PRES-TYPE,NOM-OP
   adjopimp(pres,nop) = modif-opimputil(pres,
         adj(opimputil(pres),nop))
adjopdef : PRES-TYPE ← PRES-TYPE,DEF-OP
   adjopdef(pres,defop) = modif-defopdef(pres,
         insert(defopdef(pres),nom-op(def-op),
         c(utilisable?(defop),profil(defop),def(defop))))
supinvar : PRES-TYPE ← PRES-TYPE
   supinvar(pres) = modif-invar(pres,vrai)
supopimp : PRES-TYPE ← PRES-TYPE,NOM-OP
   supopimp(pres,nop) = modif-opimputil(pres,
         supe(opimputil(pres),nop))
supopdef : PRES-TYPE ← PRES-TYPE,NOM-OP
         supopdef(pres,nop) = modif-defopdef(pres,
         supt(defopdef(pres),nop))
defnop : DEF-OP ← PRES-TYPE,NOM-OP
   defnop(pres,nop) = defenop(defopdef(pres),nop)
opdefutil : E-NOM-OP ← PRES-TYPE
   opdefutil(pres) = def(oputil(pres))
oputil : OP-UTIL ← PRES-TYPE
   oputil(pres) = c(opimputil(pres),opdefutil(pres))
oputilisables: E-NOM-OP ← PRES-TYPE
   oputilisables(pres) = union(opimputil(pres),opdefutil(pres))
opdef : E-NOM-OP ← PRES-TYPE
   opdef(pres) = itere_{nom-op,vrai}(defopdef(pres))
```


TYPE EXPR = PC[construct : NOM-CONSTRUCT, sparam : S-NOM-TYPE]
lexique : Type des expressions de type

TYPE T-DEF-OP = T[NOM-OP] DE DEFSN-OP	
opérations utilisables définies : eopdefutil,defenop	variables tdefop : T-DEF-OP nop : NOM-OP
lexique : Table associant à un nom d'opération, sa définition	
définitions eopdefutil : E-NOM-OP ← T-DEF-OP eopdefutil(tdefop) = $itert_{indice,utilisable?}(tdefop)$ defenop : DEF-OP ← T-DEF-OP,NOM-OP defenop(tdefop,nop) = c(nop,acct(tdefop,nop))	

TYPE DEFSN-OP = PC[utilisable? : BOOL, profil : PROFIL, def : E-EQUAT]
lexique : définition - sans la composante nom - d'une opération

TYPE OP-UTIL = PC[imp : E-NOM-OP, def : E-NOM-OP]
lexique : Couple des ensembles d'opérations utilisables importées et définies

TYPE DEF-OP = PC[nom-op : NOM-OP, utilisable? : BOOL, profil : PROFIL, def : E-EQUAT]	
invariant non(eq(indef,nom-op(defop)))	variable defop : DEF-OP
lexique : Définition d'une opération composée d'un nom d'un booléen indiquant si l'opération est utilisable d'un profil d'un ensemble d'équations définissant l'opération invariant : la composante nom-op d'une définition d'une opération ne doit pas être indéfinie	

TYPE PROFIL = PC[codomaine : NOM-TYPE, domaine : S-NOM-TYPE]	
invariant non(eq(indef,codomaine(prof)))	variable prof : PROFIL

TYPE S-NOM-TYPE = S[NOM-TYPE]

TYPE E-EQUAT = E[EQUAT]

TYPE E-NOM-OP = E[NOM-OP]

La transformation de la spécification nous a permi

- d'éliminer la redondance due à la composante def de la composante oputil d'une présentation de type ;
- d'optimiser les fonctionnalités du système en optimisant les fonctions d'accès à un type de l'univers et à la définition d'une opération dans une présentation.

CONCLUSION

CONCLUSION

Nous avons montré comment transformer une spécification construite suivant une démarche descendante et déductive. Cette transformation a pour but d'optimiser les fonctionnalités spécifiées, d'éliminer la redondance et de manière générale de faciliter l'évolution d'une spécification.

Nous pouvons résumer notre apport aux quatre points suivants :

- Une approche des types abstraits algébriques plus simple que dans la littérature sans introduction de la théorie des catégories.
- Le langage SPES-TYPES présentant les types abstraits algébriques de manière constructive et intuitive.
- Des transformations de types associées aux constructeurs définis.
- Des critères de comparaison de types permettant d'analyser un univers, de choisir les transformations à appliquer et ainsi de transformer un univers. Ces mêmes critères peuvent être utilisés pour la recherche de types dans une bibliothèque.

Nous avons introduit quatre constructeurs de types et les transformations associées. Il serait intéressant de compléter l'ensemble des constructeurs suivant les propositions énoncées ainsi que l'ensemble des transformations.

Ce travail pourrait être poursuivi suivant les trois axes complémentaires suivants :

- Un système conversationnel d'aide à la construction, analyse et transformation d'univers. Il pourrait être composé
  - d'un système d'aide à la construction d'un univers faisant les vérifications en parallèle de telle manière que les versions successives des types soient issues les unes des autres,
  - d'un système d'aide à l'analyse d'un univers permettant de déceler les redondances existantes et les transformations applicables, et

• d'un système d'aide à la transformation d'un univers qui appliquerait les transformations aux types concernés.

Ce système entre dans le cadre du projet SPES. Actuellement un système d'aide à la construction de spécifications fonctionnelles et d'univers de types abstraits algébriques est en cours de réalisation. La construction d'un univers se fait en parallèle de la construction d'une spécification fonctionnelle.

- Une étude de stratégies de transformation d'un univers en fonction
  • du domaine d'application,
  • du contexte matériel et logiciel, ou
  • d'une réutilisation de spécifications existantes.

  Une telle étude pourrait conduire à un système expert d'analyse d'univers.

- La définition d'une famille d'implantations pour chaque constructeur de types introduits. Celles-ci permettraient d'avoir des représentations concrètes des spécifications et donc automatiquement un prototype du système spécifié.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[ABR, 78] J.R. ABRIAL
Z : a specification language.
IFIP Tokyo 1978.

[ADJ,78] J.A. GOGUEN, J.W. THATCHER, E.G. WAGNER
An initial algebra approach to the specification, correctness and implementation of abstract data types.
Current trends in programming methodology, Vol IV
Yeh ed. Prentice hall 1978.

[ADJ, 81] H. EHRIG, H.J. KREOWSKI, J.W. THATCHER, E.G. WAGNER, J.B. WRIGHT
Parameter passing in algebraic specification languages.
Proc. Workshop "Program Specification"
Aarhus, Denmark, LNCS 134, 1981.

[BAR, 83] L.A. BARROS
ECOLOGISTE : Un système d'aide à la spécification complète et consistante d'un type abstrait algébrique.
Rapport interne n° 83-R-039 - CRIN, 1983.

[BAU,PEP,WOS, 78] F.L. BAUER, P. PEPPER, H. WÖSSNER
Notes on the project CIP : Outline of a transformation system Program Construction, F.L. Bauer et M. Broy (ed).
LNCS 69. 1978.

[BAU,WOS, 82] F. BAUER, H. WÖSSNER
Algorithmic Language and program development
Springer Verlag Texts and Monographs in computer science. 1982.

[BER,KLO, 82] J.A. BERGSTRA, J.W. KLOP
Algebraic specification for parametrized data types with minimal parameter and target algebras.
9th Colloquium ICALP 82, Aarhus Denmark.
LNCS 140 - 1982.

[BID, 81] M. BIDOIT
Une méthode de présentation des types abstraits : applications
Thèse de 3ème cycle - Paris-Sud - juin 1981.

[BID,CHO,KAP,83] M. BIDOIT, C. CHOPPY, S. KAPLAN
ASSPEGIC : un environnement de spécification algébrique
Rapport de recherche n° 144. LRI. 1983.

[BID,GAU, 82] Etude des Méthodes de spécification des cas d'exceptions dans les types abstraits algébriques.
Rapport final du poste 1 du marché DAII n° 82-35-033.
Lab. de Marcoussis - Centre de recherche de la CGE. 1982.

[BIE, 84] B. BIEBOW
Application d'un langage de spécification algébrique à des exemples téléphoniques.
Thèse de 3ème cycle - Université Pierre et Marie Curie - Paris 6 - 1984.

[BOI,GUI,PAV, 83] F. BOISSON, G. GUIHO, D. PAVOT
Algèbres à opérateurs multicibles
Rapport LRI Université de Paris Sud.1983.

[BRO,WIR, 82] M. BROY, M. WIRSING
Partial abstract types.
Acta informatica vol. 18 - Fasc. 1 - Nov. 1982.

[BUR,GOG, 77] R.M. BURSTALL, J.A. GOGUEN
Putting theories together to make specifications
Proc 5th IJCAI - Boston 1977.

[BUR,GOG, 79] R.M. BURSTALL, J.A. GOGUEN
The semantics of clear, a specification language abstract software specifications. Copenhagen winter school - LNCS n° 86. 1979.

[DER,FIN, 79] J.C. DERNIAME, J.P. FINANCE
Types abstraits de données : spécification, utilisation et réalisation.
Ecole d'été de l'AFCET Monastir
Rapport CRIN 79.E.57. 1979.

[DER, 82] P. DERANSART
Dérivation de programmes PROLOG à partir de spécifications algébriques. INRIA. 1982.

[DUB, 84] E. DUBOIS
Cadre et méthode de spécification de systèmes d'information fondés sur les types de données.
Thèse de docteur ingénieur en informatique.
Institut national polytechnique de Lorraine. 1984.

[DUB,FIN,LEV,VAN, 83] : E. DUBOIS, J.P. FINANCE, N. LEVY, A. VAN LAMSWEERDE
Spécification technique for large information systems.
Symposium IBM-FNRS, Bruxelles, 1983.

[EHR, 82] H.D. EHRICH
On the theory of specification implementation and parametrization of abstract data type.
JACM vol. 29 n° 1 - 1982.

[EHR,FEY,HAN, 83] : H. EHRIG, W. FEY, H. HANSEN
ACT ONE : An algebraic specification language with two levels of semantics.
Bericht-Nr 83-03 Technische Universität Berlin 1983.

[EHR,KRE,82] H. EHRIG, H.J. KREOWSKI
Parameter passing commutes with implementation of parameterized data types.
9th colloquium ICALP 82 Aarhus Denmark
LNCS 140 - 1982.

[EHR,KRE,MAH,PAD, 82] : H. EHRIG, H.J. KREOWSKI, B. MAHR, P. PADAWITZ
Algebraic implementation of abstract data types.
Theoretical computer Sciences vol. 20 n° 3
North Holland. 1982.

[FIN, 79] J.P. FINANCE
Etude de la construction des programmes : Méthodes et langages de spécification et de résolution de problèmes.
Thèse d'état CRIN - Université de Nancy 1, 1979.

[GAN, 83] H. GANZINGER
Parametrized specifications : parameter passing and implementation with respect to observability.
ACM Transactions on Programming Languages and Systems
Vol. 5 n° 3. 1983.

[GAU, 79] M.C. GAUDEL
Spécifications incomplètes mais suffisantes de la représentation des types abstraits.
Groplan AFCET n° 7. Mai 1979.

[GOG,DRO,LIP,EHR, 82] : M. GOGOLLA, K. DROSTEN, U. LIPECK, H.D. EHRICH
Algebraic and operational semantics of specifications allowing exceptions and errors. Internal report Un. Dortmund n° 140. 1982. Short version LNCS 145. 1983.

[GOG, 78] J.A. GOGUEN
Abstract errors for abstract data types.
Proc. IFIP Conf. "Formal description of programming concepts". 1978.

[GOG,TAR, 79] J.A. GOGUEN, J. TARDO
An introduction to OBJ : a language for writing and testing software specifications.
Specification of reliable software.
IEEE Catalog CH-1401-96. 1979.

[GUT, 81] J.V. GUTTAG
A few remarks on putting formal specifications to productive use.
Proc. workshop "Program specification".
Aarkus - Denmark - LNCS 134. 1981.

[GUT,HOR, 78] J.V. GUTTAG, J.J. HORNING
The algebraic specification of abstract Data types.
Acta Informatica 10. 1978.

[GUT,HOR, 83] J.V. GUTTAG, J.J. HORNING
An introduction to the Larch Shared Language
IFIP 1983.

[GUT,HOR,MUS, 78] : J.V. GUTTAG, E. HOROWITZ, D.R. MUSSER
The Design of data type specifications.
Current trends in programming methodology
Vol. IV. Yeh Ed. Prentice hall. 1978.

[HUE, 77] G. HUET
Démonstration automatique en logique de premier ordre.
Cours DEA Orsay. 1976-77.

[HOA, 72] C.A.R. HOARE
Proof of correctness of data representations.
Acta Informatica 1. 1972.

[JAC, 75] M.A. JACKSON
Principles of Program Design
Academic Press. 1975.

[JAC, 83] M.A. JACKSON
System Development
Prentice Hall - International series on Computer Science. 1983.

[KAM, 79] S. KAMIN
Final data types and their specification
ACM TOPLAS. 1979.

[KAP, 83] S. KAPLAN
Un langage de spécification de types abstraits algébriques
Thèse de 3ème cycle - Université de Paris Sud Orsay. 1983.

[KUT,LIC, 83] B. KUTZLER, F. LICHTENBERGER
Bibliography on Abstract Data Types
Informatik Fachberichte 68 - Springer Verlag. 1983.

[LEH,SMY, 81] D.J. LEHMANN, M.B. SMYTH
Algebraic specification of data types : a synthetic approach.
Mathematical systems theory n° 14. 1981.

[LES, 79] P. LESCANNE
Etude algébrique et relationnelle des types abstraits et de leurs représentations.
Thèse d'état CRIN. 1979.

[LIS,ZIL, 74] B. LISKOV, S. ZILLES
Programming with abstract data types.
Proceedings of ACM Sigplan vol. 9 n° 4 - Conférence on Very high level languages 1974.

[LIS,ZIL 75] B.H. LISKOV, S.N. ZILLES
Spécification techniques for data abstractions
IEEE-SE vol. SE-1 n° 1. 1975.

[LIV, 78] C. LIVERCY
Théorie des programmes. Dunod. 1978.

[McL,BIR, 78] S. Mac LANE, G. BIRKHOFF
Algèbre 2 - Les grands théorèmes.
Cahier scientifiques fascicule XXXVI. Gauthier Villars. 1978.

[MOR, 73] F.L. MORRIS
Types are not sets.
Proc. 1th ACM Symposium on principles of Programming Languages. 1973.

[MUS, 80a] D.R. MUSSER
Abstract data type specification in the AFFIRM system
IEEE trans. on software Engineering.
Vol. SE-6 n° 1. 1980.

[MUS, 80b] D.R. MUSSER
On proving inductive properties of abstract data types.
Proc. 7th POPL - Las Vegas. 1980.

[ORE, 81] F. OREJAS
On the representation of data types. Formalization of programming concepts.
Peniscola Spain. International Colloquium. LNCS 107. 1981.

[PAI, 79] C. PAIR
La construction des programmes.
RAIRO Informatique. Vol. 13, 2, 1979.

[PAI, 80] C. PAIR
Types abstraits et sémantique algébrique des langages de programmation.
Rapport CRIN 80-R-011. 1980.

[PAR, 72] D.L. PARNAS
A technique for the specification of software modules with examples.
CACM n° 15. 1972.

[PAR,PRI, 73] D.L. PARNAS, W.P. PRICE
The design of the virtual Memory Aspects of a virtual machine.
Proc. of the SIGARCH - SIGOPS workskop on Virtual Computer Systems. 1973.

[REM, 82] J.L. REMY
Etude des systèmes de réécriture conditionnels et applications aux types abstraits algébriques.
Thèse d'état CRIN-INPL. 1982.

[SPES, 84] J.P. FINANCE, M. GRANDBASTIEN, N. LEVY, A. QUERE, J. SOUQUIERES
SPES : Un système pour spécifier et transformer
2ème Colloque Génie logiciel Nice 1984.

[WIR,BRO, 81] M. WIRSING, M. BROY
An analysis of semantic models for algebraic specifications.
Proc. Int. Summer School on theoretical foundations of programming methodology. 1981.

[WIR,PEP,PAR,DOS,BRO, 83] : M. WIRSING, P.PEPPER, H.PARTSH, W.DOSCH, M.BROY
On hierarchies of abstract data types.
Acta Informatica vol. 20 fasc. 1. 1983.

[WIR,SAN, 83] M. WIRSING, D. SANNELLA
A kernel language for algebraic specification and implementation.
Proc. Int. Conf. on foundations of computer theory.
Bergholm Sweden. 1983.

[WOR, 81] Workshop on Program Specification
Aarhus - Denmark
A survey on applications of specification techniques.
LNCS 134. 1981.

[ZIL, 79] S. ZILLES
An introduction to data algebras.
Abstract software specification. Copenhagen winter school
LNCS 86. 1979.

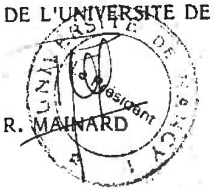
NOM DE L'ETUDIANT : Madame PICARD née LEVY Nicole

NATURE DE LA THESE : Doctorat 3ème cycle en Informatique

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le 3 JUIL. 1984 n° 1844

LE PRESIDENT DE L'UNIVERSITE DE NANCY I



R. MAINARD

RESUME

Parmi les formalismes utilisés pour décrire des spécifications formelles de structures de données, celui des types abstraits algébriques a connu un grand essor depuis une douzaine d'années. Nous avons cherché à définir des mécanismes de transformation de types abstraits algébriques, permettant de les optimiser et de faciliter leur évolution.

Nous nous sommes tout d'abord appliqués à définir un cadre formel pour donner aux types abstraits algébriques une sémantique rigoureuse.

Nous nous sommes ensuite dotés d'un langage d'expression appelé SPES-TYPES et de constructeurs de types, ceux-ci étant des types paramétrés.

Enfin nous avons utilisé pour construire une spécification la méthode déductive et descendante, développée au CRIN dans le cadre du projet SPES. Transformer une spécification doit permettre d'optimiser certaines fonctions, d'éliminer la redondance et de permettre l'évolution des structures de données. Les transformations sont dans notre travail des représentations faibles paramétrées de types.

Nous avons défini des critères d'analyse permettant de mettre à jour les défauts d'une spécification ainsi que de choisir les transformations nécessaires à leur résolution. Les transformations sont choisies parmi un ensemble dépendant des constructeurs de types utilisés dans la spécification fonctionnelle.

La dernière étape du mécanisme de transformation de types abstraits algébriques sera d'appliquer les transformations choisies. Nous développons notre approche sur l'exemple d'un système d'aide à la transformation de types abstraits algébriques suivant l'approche proposée dans ce travail.

Mots-clés: spécification algébrique, type abstrait, paramétrisation,représentation, présentation, constructeur de types, structure, univers, comparaison, redondance, transformation.