Université de Nancy I — U.E.R. Sciences Mathématiques

Centre de Recherche en Informatique de Nancy

ORSEC:
un Outil de Recherche de Spécifications Equivalentes par Comparaison d'exemples

THESE

soutenue publiquement le **10 décembre 1982**

A L'UNIVERSITE DE NANCY I

pour l'obtention du grade de
DOCTEUR DE 3ème CYCLE EN INFORMATIQUE
par
Karol PROCH
devant la Commission d'Examen

Président: . . . . . *Jean-Claude DERNIAME*
Examinateurs: . . . . . *Michel BIDOIT*
Jean-Jacques CHABRIER
Daniel COULON
Pierre-Yves CUNIN
Jean-Luc REMY

Université de Nancy I — U.E.R. Sciences Mathématiques

Centre de Recherche en Informatique de Nancy

ORSEC:
un Outil de Recherche de Spécifications Equivalentes par Comparaison d'exemples

THESE

soutenue publiquement le **10 décembre 1982**

A L'UNIVERSITE DE NANCY I

pour l'obtention du grade de
DOCTEUR DE 3ème CYCLE EN INFORMATIQUE
par
Karol PROCH
devant la Commission d'Examen

Président: ..... *Jean-Claude DERNIAME*
Examinateurs: ..... *Michel BIDOIT*
Jean-Jacques CHABRIER
Daniel COULON
Pierre-Yves CUNIN
Jean-Luc REMY

A mes parents,

à Doris et aux enfants.

C'est à Jean-Claude DERNIAME, qui m'a accueilli dans son équipe, que je dois mes premiers contacts avec la notion de type abstrait. Ses conceptions sur la programmation et sur la production de logiciel ont beaucoup influencé le choix du thème de mon travail. Je lui en suis profondément reconnaissant et le remercie d'avoir bien voulu présider ce jury.

Michel BIDOIT a bien voulu s'intéresser à ce travail ; les remarques qu'il m'a faites m'ont amené à approfondir ma réflexion et à en compléter l'exposé. Je lui en sais gré et le remercie très sincèrement de sa présence dans ce jury.

Jean-Jacques CHABRIER a toujours été présent au cours de la réalisation de cette thèse et les longues conversations que nous avons eues ont fortement contribué à son aboutissement. Je ne sais comment remercier l'ami qu'il est, si ce n'est en l'assurant de me consacrer à nos projets communs.

Les travaux de Daniel COULON en intelligence artificielle ne sont pas directement liés aux types abstraits. Je le remercie de porter ainsi un regard différent sur cette thèse et d'avoir accepté de participer à ce jury.

Dès que je me suis posé le problème de l'équivalence de deux spécifications, Pierre-Yves CUNIN m'a encouragé à creuser l'idée initiative qui a conduit à la réalisation d'ORSEC. Il s'est de plus montré un lecteur particulièrement attentif. Je l'en remercie très chaleureusement.

Sans Jean-Luc REMY j'aurais eu beaucoup de mal à formuler les propositions contenues dans cette thèse. J'ai beaucoup appris lors des nombreuses séances de travail qu'il m'a consacrées avec sa bonne humeur habituelle. Je lui dois beaucoup et l'assure de ma gratitude.

Je tiens à remercier mes collègues qui m'ont déchargé, durant ce premier trimestre, d'une partie de mes tâches d'enseignement, et aussi tous ceux que j'ai pu déranger pour relire un passage, pour obtenir une référence bibliographique, ou pour la réalisation matérielle de cette thèse.

I. INTRODUCTION

C'est un lieu commun de dire que, si les coûts du matériel ont considérablement baissé depuis l'apparition des premières machines commercialisées, les coûts du logiciel n'ont pas suivi cette tendance dans les mêmes proportions. Les raisons en sont multiples, on peut toutefois en distinguer quelques unes :

- Si une machine est généralement destinée à être produite en série, le logiciel est le plus souvent spécifique d'un utilisateur (sauf pour les produits logiciels de base : systèmes, compilateurs ...). Les coûts de conception (donc de "matière grise") sont partagés dans le cas des matériels produits en série et ne le sont généralement pas pour les logiciels.
- Les modes de conception diffèrent; si la conception du matériel relève d'une approche technicienne (normes et langages de normalisation, utilisation de composants existants, outils de simulation et de tests), la conception du logiciel est plus "artisanale" et le "savoir faire" y joue un rôle important.
- La maintenance et l'évolution du matériel sont faites généralement par remplacement et adjonction d'unités sur l'architecture existante, pour le logiciel elles peuvent amener à remettre en cause la conception du produit.

Des tentatives de résolution de ces problèmes ont été et sont tentées :

- La définition de "packages" ou "progiciels" généraux pour des applications standards (paye, gestion de stocks, comptabilité ...) a visé à l'introduction de la notion de "produit logiciel de série", il semble toutefois que leur succès soit tout relatif et que les utilisateurs préfèrent le "sur-mesure". Ce réflexe est très répandu et

il suffit pour s'en convaincre d'énumérer les différents compilateurs pouvant exister pour un "même" langage de programmation. La raison en est très certainement la difficulté de comprendre précisément les fonctions d'un produit en l'absence d'un langage commun d'expression des caractéristiques du logiciel.

- Les éléments de réponses aux problèmes liés à la conception reposent sur un ensemble de concepts, de méthodes et d'outils qu'on recouvre sous le nom "génie logiciel". De très nombreux travaux leurs sont consacrés, on trouvera dans <F&W80> un ensemble d'articles et une importante bibliographie sur ces trois grands thèmes. Généralement les concepts utilisés sont ceux de modularité et d'abstraction, les méthodes s'appuient sur une démarche descendante, quant aux outils ils intègrent des langages et leurs traducteurs, des éditeurs de textes, des metteurs au point, des bases documentaires ... L'intégration d'un tel ensemble d'outils en un système homogène conduit à un "atelier de fabrication de logiciel" ou "système intégré de production de logiciel"; on trouvera dans <KRA82> une description des fonctions principales de tels systèmes.

Nous n'insisterons pas sur l'ensemble des "bonnes propriétés" du logiciel et des méthodes permettant de l'obtenir, toutefois trois notions sont importantes : la modularité, l'abstraction, la réutilisabilité.

I.1. MODULARITE

Il est évident que la modularité est une qualité primordiale tant au niveau de la démarche d'analyse, qu'à celui du logiciel produit. Les "modules" ou "composants logiciels" sont l'analogue des boitiers matériels et leur existence permet d'envisager une partie de la réalisation du produit comme un jeu de construction. De nombreux langages et systèmes de programmation permettent la définition de modules (SIMULA <DAH68>, CIVA <DERn74>, MODULA <WIR76> <WIR80>, CLU <LIS74>, ATM <MIN79> <HEN80>, LEGOS <LEG80>, ADA <ADA82> ...) en proposant des constructions ad-hoc. Toutefois les problèmes surgissent lorsqu'il s'agit de définir "ce que l'on met" dans un module et comment on obtient un découpage modulaire. Ce que l'on peut dire de la façon la plus générale, c'est qu'un module apparait extérieurement comme un ensemble de procédures et/ou de variables (et/ou de types comme dans ADA) regroupées en une unité syntaxique et on peut admettre qu'un "bon" découpage modulaire est tel que les modules soient le plus indépendants possible les uns des autres, ceci en vue de faciliter la localisation des erreurs, les tests, et le remplacement d'un module par un autre. Ces dernières préoccupations ont conduit dans certains cas (ATM, ADA) à définir des unités syntaxiques spécifiques contenant l'"interface" d'un module, c'est à dire l'ensemble des objets du module visibles à l'extérieur ainsi que leurs attributs (profils syntaxiques des procédures, types des variables, ...). Cette séparation physique interfaces/corps permet une plus grande souplesse dans l'ordre des compilations d'un ensemble de modules, tout en permettant d'effectuer les contrôles de cohérence nécessaires (la compilation s'effectue alors dans un contexte qui s'enrichit progressivement). Des mécanismes supplémentaires s'apparentant à l'édition de liens dynamique, comme c'est le cas en ATM, fournissent alors un confort

d'utilisation certain <HEN80>. Nous ne nous attarderons pas plus sur la notion de modularité et nous renverrons à la lecture de <SOK80> pour un exposé plus complet.

I.2. ABSTRACTION

Il n'est pas simple de définir précisemment la notion d'abstraction, bien que ce terme soit couramment utilisé. Son utilisation sous-entend généralement l'existence d'un niveau de détail en dessous duquel on ne veut pas descendre lors d'une démarche descriptive et qu'on ne conserve , pour un objet , que les propriétés nécessaires à son utilisation .

Utiliser l'abstraction lors de la conception d'une application consistera donc à définir un certain nombre de niveaux de détails, et à raisonner en ignorant à chaque moment les propriétés et les objets des niveaux inférieurs. Ainsi lorsque l'on définit un algorithme logique manipulant une liste on fait abstraction du problème de la représentation des listes et des structures de contrôle qu'on utilisera dans le langage de programmation cible. Une telle démarche sous-entend

1) qu'on sache définir les différents niveaux de détails pour le problème à résoudre, et ceci à priori

2) qu'on dispose d'un moyen de préciser les propriétés des objets qu'on utilise à un instant donné

3) qu'on puisse passer de façon cohérente d'un niveau à un autre, c'est à dire en garantissant que les propriétés des objets manipulés sont vérifiées par leurs représentations dans le niveau inférieur.

La combinaison de la modularité et de l'abstraction a conduit à la notion de

type abstrait par laquelle on cherche à caractériser une structure de donnée indépendamment de sa représentation effective, et qui a été introduite dans certains langages de programmation, sous un aspect essentiellement syntaxique (CLU, ATM), ou en vue de faciliter des preuves de programmes (ALPHARD <WLS76>). Nous présenterons de façon plus détaillée ce concept dans le chapitre II, on pourra toutefois lire <D&F79> pour avoir un panorama des différentes méthodes de spécification de types abstraits et des langages permettant leur utilisation.

I.3. REUTILISABILITE

Si la modularité et l'abstraction facilitent l'analyse, la validation et le test des logiciels, elles nécessitent des moyens supplémentaires pour réutiliser des unités déjà construites. L'existence de bibliothèques de sous-programmes et les mécanismes d'édition de liens classiques fournis par les systèmes permettent de faire l'économie de l'écriture et de la mise au point des procédures conservées. Certains langages (COBOL, PL1, PASCAL UCSD, ...) fournissent des directives d'inclusion de textes stockés dans des librairies (COPY, INCLUDE, (*$I ...*)). Ces bibliothèques restent toutefois de simples fichiers et les mécanismes d'accès sont fournis par les utilitaires systèmes standards, lesquels sont conçus à des fins générales, indépendamment des fonctions particulières souhaitables dans ce contexte (accès aux sous-programmes d'un programme, graphe d'appel ...). De la même façon le manuel de référence du langage ADA sous-entend l'existence d'une telle librairie contenant des unités dont l'interface est rendue visible par l'intermédiaire de clauses "USE". Pour l'instant, toutes les fonctionnalités

de l'environnement de programmation ainsi constitué ne sont pas encore définies et font partie du cahier des charges "STONEMAN" <DOD80> présenté par le département de la défense américaine.

De la même façon, l'existence d'unités "génériques" ou "paramétrées", permettant de définir une famille d'objets par un texte unique, contribuera à alléger les tâches de conception et de réalisation.

Des systèmes intégrant des bibliothèques existent, nous en verrons quelques exemples au chapitre II, et fournissent la possiblité de se servir du travail précedemment accompli; toutefois dans chacun de ces cas l'utilisation de la bibliothèque nécessite que l'utilisateur connaisse le nom et le rôle de chaque composant (sa "sémantique"). Si sur des "petites" bibliothèques utilisées par une personne la connaissance du contexte peut se faire rapidement, le problème s'amplifie dans le cas de "bases" plus importantes et multi-utilisateurs ; dans ce cas l'utilité d'un bibliothécaire capable de fournir, non pas une unité de nom donné, mais une unité d'un "comportement" donné devient évidente.

L'objectif des langages de spécification est précisemment de fournir un moyen de description de la sémantique permettant la conduite de preuves, afin de garantir la correction du logiciel. Si un système de développement de logiciel intègre un langage de spécification et une bibliothèque d'unités, on voit l'intérêt d'un outil capable de "comparer" des spécifications et de délivrer les unités de l'environnement ayant une sémantique équivalente à celle d'une spécification donnée.

C'est à ce problème de recherche d'une unité d'un comportement donné que nous avons tenté d'apporter un élément de réponse, en nous situant dans le cadre des types abstraits algébriques, ce choix prolongeant assez naturellement notre participation au langage ATM <PRO79>.

Nous fournissons dans le chapitre III quelques exemples, définitions et

propositions précisant la notion de spécifications équivalentes évoquée dans les lignes qui précèdent.

Compte tenu de nos préoccupations et de notre approche pragmatique, nous n'avons pas effectué tous les développements théoriques qui peuvent sembler nécessaires, nous avons surtout cherché à justifier, à postériori, le prototype (présenté dans le chapitre IV) que nous avons réalisé.

L'écriture et la mise au point de ce prototype, assez facilement modifiable, et permettant de faire des expérimentations a été notre objectif essentiel.

II. TYPES ABSTRAITS ALGEBRIQUES

Le concept de type abstrait vise à prendre en compte à la fois la modularité et l'abstraction. De façon intuitive un type abstrait définit un ensemble d'objets par et avec les opérations servant à "manipuler" ces objets.

Ce concept est utilisé à la fois dans les langages de programmation, comme élément de structuration des programmes, et dans le cadre plus formel de la spécification. Ces deux points de vue, loin d'être antagonistes sont complémentaires.

Son introduction dans un langage de programmation contribue à augmenter la fiabilité des logiciels ; on peut en effet effectuer dans ce cas un contrôle "fort" des types et assurer que toute variable d'un programme est utilisée conformément à son type. On oblige ainsi à une plus grande rigueur dans le choix et la définition des structures de données.

De plus si on distingue ,comme dans le langage ATM , l'interface d'un type abstrait (liste des profils syntaxiques des opérations) de sa réalisation (représentation des objets et algorithmes des opérations) , on aura la possibilité de reporter le plus tard possible (juste avant l'exécution , lors de la connexion) les choix d'implantation et de modifier la réalisation d'une unité sans recompiler l'ensemble des unités l'utilisant. Cette souplesse d'utilisation ,particulièrement appréciable lors de la mise au point d'un logiciel , présente en contrepartie l'inconvénient de produire un code peu efficace , l'allocation de l'espace mémoire pour les différentes variables se faisant dynamiquement. Rien ne s'opposerait cependant à ce qu'une fois les choix d'implantation fixés , un traducteur produise un meilleur code. Cette remarque justifie la position prise par les auteurs du langage ADA dans lequel on précise dans la partie "privée" d'une

"spécification de package" la représentation des types qu'il définit.

A cette vision des types abstraits comme élément des langages de programmation s'ajoute le besoin d'un formalisme de description du comportement d'un module (indépendamment de son implantation dans un langage de programmation) permettant d'effectuer les preuves nécessaires pour garantir la correction du logiciel. Différentes méthodes de spécification ont été proposées , nous utiliserons dans cette thèse l'approche algébrique qui bénéficie de nombreux travaux et se prête bien à notre démarche. Ce chapitre a pour objet de rappeler le vocabulaire et les principales définitions habituellement utilisées.

II.1. SPECIFICATION

On distingue trois parties dans la spécification ou la présentation <GAU80> d'un type abstrait T que l'on notera $\langle S,\Sigma,E\rangle$

. un ensemble noté S de noms de domaines ou sortes et contenant une sorte d'intérêt notée t

. un ensemble Σ ,appelé signature , de noms d'opérations munies de leurs profils construits sur S^+. Ces profils précisent le type des arguments (les domaines) et du résultat (le codomaine) de chaque opération. La sorte d'intérêt doit apparaitre dans chaque profil.

. un ensemble E d'axiomes sous forme d'équations entre des termes , construits en utilisant les symboles de Σ et des symboles de variables quantifiées universellement.

(Ces équations serviront à définir un ensemble de relations d'équivalence sur les objets du type abstrait comme nous le verrons par

la suite.)

Nous confondrons dans la suite du texte , sorte (support des opérations) et type (support + opérations).

On remarquera que l'on privilégie dans la définition précédente un nom de type (le type d'intérêt) dans l'ensemble S , celui que l'on est "en train" de définir , les autres sont considérés comme déjà définis ailleurs et font partie de l'"environnement". On trouvera dans la thèse de M. Bidoit <BID81> une formalisation de cette notion d'environnement.

II.2. SEMANTIQUE

Pour doter une spécification d'une sémantique il faut lui associer un modèle , c'est à dire associer à chaque nom de type un ensemble et à chaque nom d'opération une application (en respectant les profils) de telle manière que les axiomes soient vérifiés, c'est à dire qu'ils soient des théorèmes du modèle.

Une façon d'associer un modèle à une spécification $\langle S,\Sigma,E\rangle$ consiste

1°) à utiliser comme support l'ensemble $T_{\Sigma/\equiv}$ des classes d'équivalence des termes clos (termes sans variables construits en utilisant les symboles de Σ et "bien formés" relativement aux profils des opérations) définies par une relation de congruence compatible avec les axiomes

2°) à faire correspondre à chaque opération o de Σ une application o_T sur les classes d'équivalence définie comme suit :

si on note $(\ ,\ )$ les symboles séparateurs dans le langage des termes et $[t]$ la classe d'équivalence du terme t , l'application o_T sera telle que

pour tout $t_1 \; .. \; t_n$ dans $T_{\Sigma/\equiv}$

$o_T([t_1],..,[t_n]) = [\underline{o}\,(\, t_1 \perp .. \perp t_n \,)]$

ce qui permet de garantir la propriété de congruence.

Comme on le verra dans la suite du texte, plusieurs congruences peuvent en général être retenues ; on devra préciser les choix effectués.

Nous allons illustrer ce mécanisme par un exemple

Considérons la spécification <S,Σ,E> suivante :

S = {booléen}

Σ = {vrai : ->booléen ,faux : ->booléen, et : booléen x booléen -> booléen }

E = {pour tout x dans booléen

et(vrai,vrai) = vrai ,

et(faux,x) = faux ,

et(x,faux) = faux }

Définissons maintenant un modèle de <S,Σ,E>.

Nous allons tout d'abord définir par la grammaire G l'ensemble des termes clos T_Σ . La grammaire est présentée par un quadruplet

(vocabulaire non terminal,vocabulaire terminal, relation de production, axiome)

et par la liste de ses règles de production. Dans ces règles "|" note l'alternative.

Conformément au 1°) nous utilisons les symboles de la spécification comme vocabulaire terminal en les soulignant pour bien distinguer les différents pas de la construction du modèle.

Soit V = { vrai , faux , et , (, ⊥ ,) }

et la grammaire suivante

G = ({X,Y,Z} ,V,::=,Z)

avec

Z ::= X | Y

Y ::= faux | et(Y ⊥ Z) | et(Z ⊥ Y)

X ::= vrai | et(X ⊥ X)

Notons L_α le langage des mots de V^* dérivant de α dans G. On obtient ainsi les langages

L_Z , L_X , L_Y

On remarque que

$L_X \cap L_Y = \emptyset$ en effet un mot de L_Y contient au moins une occurrence de faux alors qu'un mot de L_X n'en contient aucune

et que

$L_X \cup L_Y = L_Z$ grâce à la règle Z ::= X | Y .

L_X et L_Y définissent une partition de L_Z.

A ce stade nous avons défini T_Σ ; pour associer un modèle à <S,Σ,E> nous devons utiliser comme support un ensemble de classes d'équivalence définies dans T_Σ .

Soit ≡ définie sur L_Z par la partition L_X,L_Y on a

pour tout x dans L_Z $[x] = [\underline{vrai}]$ ou $[x] = [\underline{faux}]$

car vrai est dans L_X et faux dans L_Y.

Il est clair que le fait de distinguer vrai et faux en les utilisant dans la représentation de leurs classes d'équivalence respectives s'appuie sur la compréhension intuitive de la sémantique que l'on désire associer à <S,Σ,E> et sur les conventions de notation généralement utilisées.

Nous sommes maintenant en mesure de préciser le support de notre modèle, c'est l'ensemble des classes d'équivalence définies dans T_Σ .

Soit B = { [vrai] , [faux] }

B constituera le support de notre modèle , définissons maintenant des opérations sur B. Conformément au 2°) ces opérations ont des classes d'équivalence comme arguments et comme résultat, leurs profils seront

$vrai_B$: -> B

$faux_B$: -> B

et_B : B x B -> B

et leurs définitions

d1) $vrai_B$ = $\lceil \underline{vrai} \rceil$

d2) $faux_B$ = $\lceil \underline{faux} \rceil$

d3) pour tout x,y dans L_Z

$et_B(\lceil x \rceil, \lceil y \rceil) = \lceil \underline{et}(x , y) \rceil$

Partant du langage L_Z construit sur les symboles de Σ (et les séparateurs), nous avons construit un modèle (B, $vrai_B$, $faux_B$, et_B) en définissant une relation d'équivalence et les classes correspondantes, et en définissant des opérations sur ces classes. Nous l'avons fait en suivant 1^o) et 2^o), de ce fait l'association entre les symboles du modèle et ceux de la spécification est définie par l'application h :

h(booléen) = B

h(vrai) = $vrai_B$

h(faux) = $faux_B$

h(et) = et_B

Nous devons vérifier que les profils sont conservés, ce qui est assuré par construction, et que les axiomes de E sont valides dans le modèle , c'est à dire que :

pour tout $\lceil x \rceil$ dans B

1) $et_B(vrai_B, vrai_B) = vrai_B$

2) $et_B(faux_B, \lceil x \rceil) = faux_B$

3) $et_B(\lceil x \rceil, faux_B) = faux_B$

En effet rien pour l'instant, si ce n'est l'intuition, ne nous le garantit. Nous avons construit la relation d'équivalence sans nous soucier explicitement de sa "compatibilité" avec les axiomes de la spécification.

1) par définition

$vrai_B$ = $\lceil \underline{vrai} \rceil$ par D1)

$et_B(vrai_B, vrai_B) = et_B(\lceil \underline{vrai} \rceil, \lceil \underline{vrai} \rceil)$

et d'après D3)

$et_B(\lceil \underline{vrai} \rceil, \lceil \underline{vrai} \rceil) = \lceil \underline{et} (\underline{vrai} , \underline{vrai}) \rceil$

or $\underline{et} (\underline{vrai} , \underline{vrai})$ est dans L_X

par définition de la congruence on a

$\lceil \underline{et} (\underline{vrai} , \underline{vrai}) \rceil = \lceil \underline{vrai} \rceil$

qui est égal à $vrai_B$ par D1) .

2) $et_B(faux_B, \lceil x \rceil) = et_B(\lceil \underline{faux} \rceil, \lceil x \rceil)$ par D2)

$et_B(\lceil \underline{faux} \rceil, \lceil x \rceil) = \lceil \underline{et} (\underline{faux} , x) \rceil$ par D3)

or $\underline{et} (\underline{faux} , x)$ est dans L_Y d'ou

$\lceil \underline{et} (\underline{faux} , x) \rceil = \lceil \underline{faux} \rceil$

donc $et_B(faux_B, \lceil x \rceil) = \lceil \underline{faux} \rceil = faux_B$ par D2) .

3) se démontre de façon similaire .

Nous avons développé longuement cet exemple simple car il permet de faire la différence entre les symboles de la spécification , leurs représentations dans le modèle, et les terminaux du langage des termes (vrai,$vrai_B$, $\underline{vrai}$ par exemple) ce qui est souvent caché par l'identité des noms. Il est à noter que cette façon d'associer un modèle à une spécification est une possibilité parmi d'autres; rien ne s'oppose à ce que le support soit construit sur un langage utilisant d'autres symboles que ceux de la spécification. D'autre part l'utilisation d'une grammaire pour définir les classes d'équivalence ne sera pas possible dans le cas général.

Par ailleurs cet exemple met en évidence l'importance de la définition de la

relation d'équivalence utilisée. Ici , cette définition ne présente pas de difficulté dans la mesure ou on s'appuie sur la logique "naturelle" pour définir le type booléen ; la situation n'est pas toujours aussi simple.

Considérons la spécification du type ensemble suivante :

S = {ensemble,entier,booléen}

= { vide : -> ensemble ,
estdans : (ensemble,entier) -> booléen ,
ajouter : (ensemble,entier) -> ensemble ,
enlever : (ensemble,entier) -> ensemble }

E = {pour tout x dans ensemble ,
pour tout i1,i2 dans entier ,
estdans (vide,i1) = faux ,
estdans (ajouter(x,i1),i2) = si égal(i1,i2) alors vrai
sinon estdans(x,i2) ,
enlever (vide,i1) = vide ,
enlever (ajouter(x,i1),i2) = si égal(i1,i2) alors enlever(x,i2)
sinon ajouter(enlever(x,i2),i1) }

le type booléen étant défini par ailleurs ainsi que le type entier muni d'une opération

égal : (entier,entier) -> booléen telle que

pour tout entier x égal(x,x) = vrai .

On remarquera l'utilisation de conditionnelles dans les axiomes ,ceci revient à définir pour tout type t une opération

si_t : (booléen,t,t) -> t

avec les axiomes

pour tout x,x' dans t

si(vrai,x,x') = x ,

si(faux,x,x') = x'

Nous utiliserons la "surchage" de l'opérateur "si" supposé implicitement défini pour tout type et une notation "si...alors...sinon.." facilitant la lecture.

Pour définir la relation d'équivalence sur les termes on peut en particulier considérer que

ajouter (ajouter (vide , 1) , 2) et

ajouter (ajouter (vide , 2) , 1)

font partie de la même classe d'équivalence ou de classes distinctes. En effet les axiomes n'imposent aucun choix pour ces deux termes alors qu'ils exigent que

enlever (ajouter (ajouter (vide , 2), 1), 2) et

ajouter (vide , 1) aient la même classe d'équivalence.

Pour une même présentation <S,Σ,E> , plusieurs congruences pourront être retenues , si on choisit la plus petite (celle qui permet de valider les équations de E sans plus) on se place dans le cadre de l' algèbre initiale . Dans ce cas deux termes clos seront considérés comme différents si on ne peut prouver par E qu'ils sont égaux. C'est l'approche du groupe ADJ <ADJ78>,de M. Bidoit <BID81>, J.L. Rémy <REH82> , M. Ehrig <EHR82> , particulièrement retenue car on sait alors, sous certaines conditions, décider de l'égalité de deux termes en interprétant les axiomes comme des règles de réécriture.

On peut au contraire considérer deux termes comme équivalents dès qu'on ne peut prouver qu'ils sont différents. C'est l'approche terminale de M. Wand <WAN79>, S. Kamin <KAM80> ,V. Giarratana et al <GGM76>.

Si on se préoccupe essentiellement du problème de la représentation , on pourra choisir de ne pas privilégier l'une ou l'autre de ces "bornes" , mais de prendre en compte l'ensemble des algèbres possibles. C'est l'approche de P. Lescanne <LES79>, de J.V. Guttag <GHM78>, M. Broy et M. Wirsing <B&W80>

et de C. Pair <PAI80>.

En pratique il semble raisonnable d'utiliser l'algèbre initiale lorsque l'on veut raisonner sur le type lui même pour démontrer des équations , car c'est dans ce contexte que l'on fait en quelque sorte le moins d'hypothèses (deux termes équivalents dans le modèle initial le sont dans tout modèle, mais l'inverse n'est pas vrai).

Par contre au niveau des représentations , il semble plus réaliste de s'autoriser une "gamme" de modèles.

Nous nous placerons pour notre part dans le cadre de l'algèbre initiale.

II.3. PROPRIETES D'UNE SPECIFICATION

Etant donné une spécification $\langle S,\Sigma,E\rangle$ on peut se poser la question de savoir si elle est conforme à ce qu'on attend d'elle, si elle ne conduit pas à des contradictions, et si elle définit complètement l'ensemble d'objets et d'opérations du type présenté.

Ces problèmes sont ceux de correction de consistance et de complétude .

II.3.1. Correction.

Une spécification est correcte vis à vis d'un modèle si celui ci est isomorphe à l'algèbre initiale associée à la présentation. Ceci suppose donc que l'on connaisse le modèle avant d'exhiber la spécification. Ce ne sera pas toujours le cas en informatique ou on utilisera essentiellement une spécification pour définir de nouveaux objets dont on n'a pas à priori une

axiomatisation, mais pour lesquels on souhaite qu'un certain nombre de "lois" soient vérifiées. On trouvera dans <ADJ79> et <GOG80> des méthodes de preuves de correction.

On peut rattacher à ce problème de correction , ou plus exactement à celui du "comportement" d'une spécification le système VEGA <CHA82> ou on "teste" une spécification afin de vérifier (sur des exemples) si les résultats fournis sont conformes à ceux attendus.

II.3.2. Consistance.

La consistance d'une spécification exprime le fait que les axiomes ne conduisent pas à des contradictions vis à vis des types déjà définis et utilisés dans la spécification.

Si on ajoute à la spécification précédente de "ensemble" l'opération

long : ensemble -> entier

et les trois axiomes

1. long(vide) = 0
2. long(ajouter(x,il)) = long(x) + 1
3. ajouter(ajouter(x,il),il) = ajouter(x,il)

on obtient une spécification inconsistante en effet

long(ajouter(ajouter(vide,1),1)) = 2

en appliquant les équations 2. , 2. , 1. , mais

ajouter(ajouter(vide,1),1) = ajouter(vide,1) par 3.

et

long(ajouter(vide,1)) = 1 par 2. et 1. .

II.3.3. Complétude.

Une première forme de complétude exprime qu'on est capable de prouver pour deux termes quelconques, soit qu'ils sont équivalents dans l'algèbre initiale, soit qu'ils sont distincts dans l'algèbre terminale, c'est à dire rendant des valeur distinctes lorsqu'on leurs applique une opération dont le résultat est d'un type primitif. De ce fait la spécification précédente du type "ensemble" n'est pas complète; comme nous l'avons vu en II.2. on ne peut prouver pour

ajouter (ajouter (vide , 1) , 2) et

ajouter (ajouter (vide , 2) , 1)

qu'ils sont équivalents dans l'algèbre initiale. Dans l'algèbre terminale ces deux termes, si on leur applique l'opération "estdans" avec un même argument de type entier, délivrent les mêmes valeurs, de ce fait on ne peut prouver qu'ils sont distincts (on les considèrera comme équivalents dans l'algèbre terminale).

Cette première forme de complétude est trop forte et J.V. Guttag et J.J. Horning proposent la notion plus faible de "complétude suffisante" qui exprime intuitivement qu'une spécification n'ajoute pas de nouveaux objets aux types déjà définis. Dans ce cas, pour chaque opération f de la signature dont le codomaine n'est pas le type d'intérêt et pour chaque terme $f(x_1..x_n)$ on doit trouver un terme u ne contenant pas d'opération de Σ, tel que $f(x_1..x_n) = u$ puisse être démontré en utilisant les axiomes de E.

Les propriétés de consistance et de complétude suffisante sont en général indécidables. On trouvera cependant dans <G&H78> l'énoncé de critères syntaxiques permettant de s'assurer de la complétude suffisante, ainsi que dans <BID81>. Pour ce qui est de la consistance, on peut en interprétant

les axiomes comme des règles de réécriture s'assurer qu'ils forment un système confluent (ou ayant la propriété de Church-Rosser). On utilise pour cela l'algorithme de Knuth-Bendix <K&B70>,<HUE77> à condition que le système soit à terminaison finie. Si les règles ne forment pas un système confluent l'algorithme permet, dans certains cas, d'ajouter des règles assurant la confluence.

II.4. TYPES ABSTRAITS ET ERREURS

De même qu'en algorithmique, la définition des cas d'erreurs est un aspect important de la spécification; on aimerait pouvoir exprimer par exemple que le sommet d'une pile vide n'est pas défini et que son utilisation conduit à une erreur. Cette prise en compte des erreurs ne se fait pas de façon immédiate, l'introduction brutale de "termes-erreur" et des axiomes associés pouvant conduire à des spécifications inconsistantes. On peut citer deux tentatives de résolution de ces problèmes.

II.4.1. Spécification d'erreurs par restriction.

Cette méthode proposée par J.V. Guttag <GUT80> revient à restreindre pour chaque opération l'ensemble des objets auxquels elle peut s'appliquer, et ce au moyen de préconditions ; dans l'exemple précédent on dira que sommet est défini pour toute pile sauf la pile vide.

II.4.2. Algèbres avec erreurs.

Dans cette approche , due à J.A. Goguen <GOG78>, on considérera des spécifications de la forme $\langle S, \Sigma_{OK} \cup \Sigma_{ERR}, E_{OK} \cup E_{ERR} \rangle$ ou Σ_{ERR} est un ensemble d'opérations-erreur et E_{ERR} l'ensemble des axiomes ou figurent les opérations de Σ_{ERR}. Cette façon de procéder permet de définir des classes d'équivalence pour les termes "normaux" d'une part, et pour les termes-erreur (tout terme contenant au moins une occurrence d'une opération de Σ_{ERR} , ou tout terme équivalent à un tel terme) d'autre part , on construira alors le modèle du type abstrait en utilisant l'ensemble de ces classes d'équivalence comme support.

II.5. CONSTRUCTION DE TYPES

La construction de spécifications pourra se faire de façon plus rapide si on est capable de se servir de spécifications existantes. Nous allons citer quelques moyens de le faire.

II.5.1. Types abstraits paramétrés.

Lorsque l'on définit une pile ou une liste, le type des éléments de la pile ou de la liste est secondaire , plus exactement on aimerait définir une "pile d'éléments" et s'en servir comme "modèle" lorsqu'on a besoin d'une pile d'entiers. Si le passage d'une spécification "pile d'éléments" à une présentation de "pile d'entiers" peut se faire sans problème en remplaçant

dans le texte de la première les occurrences d'"élément" par "entier" , ce ne sera pas aussi simple dans le cas général. La définition d'un "ensemble d'éléments" nécessitant l'utilisation d'un prédicat d'égalité sur le paramètre formel "élément" , une spécification paramétrée devra donc exprimer cette condition. Lors d'une instanciation on devra être capable de s'assurer de la conformité du type paramètre effectif par rapport à cette condition. Les auteurs déjà cité ont abordé ces problèmes ainsi que H. Ehrig dans <EHR81>.

II.5.2. Sous-types abstraits.

L'idée de sous-type relève du même souci d'économie , en cherchant à définir un type comme une partie d'un autre, ou à partir d'un autre, par exemple définir les entiers positifs à partir des entiers relatifs , ou les listes triées à partir des listes. Ceci pourra se faire en caractérisant les objets auxquels on s'intéresse par une propriété . On devra examiner comment les opérations du type se "transportent" dans le sous-type.

II.5.3. Types enrichis.

Une autre façon d'obtenir un type à partir d'un type existant consiste à procéder par enrichissement . Cette fois on conserve les mêmes objets (au sens de classes d'équivalence) et on ajoute des opérations et les axiomes qui servent à les définir. On pourra ainsi enrichir le type "entier relatif" $\{zero, succ, pred\}$ avec les opérations "plus" et "moins". Là encore , on doit s'assurer que les axiomes introduits ne modifient pas l'ensemble des

classes d'équivalence <ADJ79>, c'est à dire que la spécification enrichie est suffisamment complète et consistante vis à vis de la spécification initiale <REM82a>.

II.6. REPRESENTATION D'UN TYPE ABSTRAIT

Le problème de la représentation d'un type abstrait consiste à définir , d'une part les objets et d'autre part les opérations en termes de types "plus simples", de telle façon que les axiomes du type soient des théorèmes de la représentation.

Une façon d'atteindre cet objectif nous est donnée indirectement par la définition de la sémantique d'un type abstrait en utilisant comme support le langage des termes. De façon analogue à l'exemple des booléens présenté en II.2. on devra exprimer les opérations du type en termes d'opérations sur les mots du langage et définir une congruence sur ces mêmes mots. Lorsqu'on peut associer un système de réécriture convergent à la spécification, on est en mesure de définir pour chaque classe d'équivalence une forme normale unique. On obtient alors une représentation directe des types abstraits qui correspond à une représentation en terme d'arbres dont les noeuds sont valués par les symboles de la présentation <LES79>. Nous reviendrons plus en détail sur cet aspect dans la suite du texte.

De façon générale , pour donner une représentation d'un type abstrait , on doit d'une part préciser le support de la représentation puis définir une fonction allant soit des objets de la représentation vers les objets du type abstrait (fonction d'abstraction) , soit des objets du type abstrait vers les objets de la représentation (fonction de représentation). On devra

prouver que tout objet du type source est représenté , et que les axiomes sont "conservés". Les preuves pourront être faites par induction.

II.6.1. Fonction d'abstraction.

Nous allons illustrer par un exemple cette approche développée en particulier dans <GHM78> et <GAU78> , et introduite semble-t-il par C.A.R. Hoare dans <HOA72>.

Considérons la présentation des entiers relatifs suivante :

```
type relatif
    profils
     zero : -> relatif
     succ : relatif -> relatif
     pred : relatif -> relatif
    axiomes
     pour tout x dans relatif
     succ(pred(x)) = x
     pred(succ(x)) = x
fin
```

Pour représenter le type "relatif" nous pouvons choisir d'utiliser les entiers naturels "nat" et les caractères "car" en représentant un entier relatif par un couple (signe,valeur absolue). Il faudra alors définir la fonction d'abstraction A qui à un tel couple associe l'entier relatif qu'il représente.

Spécifions les types "nat" et "car" de la façon suivante :

```
type nat
    profils
```

zeronat : ->nat

succnat : nat -> nat

prednat : nat -> nat

egal : nat , nat -> booléen

axiomes

pour tout x,x' dans nat

prednat(zeronat) = zeronat

prednat(succnat(x)) = x

egal(zeronat,zeronat) = vrai

egal(zeronat,succnat(x)) = faux

egal(succnat(x),zeronat) = faux

egal(succnat(x),succnat(x')) = egal(x,x')

fin

type car

profils

'a' : -> car

'b' : -> car

.

.

.

'z' : -> car

'+' : -> car

'-' : -> car

fin

Habituellement on représente le signe par le caractère "+" ou "-" , de ce fait tous les couples de "car" x "nat" ne représentent pas des entiers

relatifs. Le domaine de définition de A n'est pas "car" x "nat" mais $\{'+','-'\}$ x "nat". On appelle invariant de représentation la caractérisation de ce domaine de définition.

La fonction A doit être surjective : tous les objets du type abstrait doivent avoir une représentation, mais n'est pas nécessairement injective , plusieurs représentations concrètes d'un même objet abstrait peuvent exister , dans notre exemple nous aurons :

A('+',zeronat) = A('-',zeronat) = zero .

Il reste à préciser comment se comportent les opérations du type abstrait sur la représentation , ici on aura :

pour tout n dans nat

D1) zero = A('+',zeronat) = A('-',zeronat)

D2) succ(A('+',n)) = A('+',succnat(n))

D3) pred(A('-',n)) = A('-',succnat(n))

D4) succ(A('-',n)) = si egal(n,zeronat) alors A('+',succnat(zeronat))
sinon A('-',prednat(n))

D5) pred(A('+',n)) = si egal(n,zeronat) alors A('-',succnat(zeronat))
sinon A('+',prednat(n))

On doit vérifier

1°) que A est surjective , ce qui est systématique par récurrence sur les constructeurs de "relatif" en utilisant les définitions D1) .. D5)

2°) que A est définie sans ambiguité , on a par exemple :

A('+',succnat(zeronat)) = succ(A('+',zeronat)) par D2)

A('+',succnat(zeronat)) = succ(A('-',zeronat)) par D4)

or A('+',zeronat) = A('-',zeronat) = zero par D1)

3^{o}) que les axiomes de "relatif" sont des théorèmes dans la représentation , ainsi :

succ(pred(x)) = x

compte tenu de l'invariant de représentation on doit montrer que

1) succ(pred(A('+',n))) = A('+',n)

et

2) succ(pred(A('-',n))) = A('-',n)

1) afin d'appliquer D5) on doit envisager deux cas , on a en effet

egal(n,zeronat) = (n = zeronat)

a) succ(pred(A('+',zeronat))) = succ(A('-',succnat(zeronat)))

= A('-',prednat(succnat(zeronat))) par D4)

= A('-',zeronat) par la $2^{ième}$ règle de "nat"

= A('+',zeronat) par D1)

b) n ≠ zeronat

succ(pred(A('+',n))) = succ(A('+',prednat(n))) par D5)

= A('+',succnat(prednat(n))) parD2)

et par induction en posant n = succnat(n')

= A('+',succnat(prednat(succnat(n'))))

= A('+',succnat(n')) par la $2^{ième}$ règle de "nat"

= A('+',n)

La démonstration est analogue pour le deuxième axiome de "relatif".

II.6.2. Fonction de représentation.

Là encore on doit définir le domaine des objets servant à la représentation, et l'égalité de deux objets de la représentation , mais cette fois la fonction de représentation R exprime les opérations du type source en termes des opérations des types cibles. Pour le même exemple on aura :

<'+',zeronat> = <'-',zeronat>

et R sera définie par :

R(zero) = <'+',zeronat>

R(succ)(<'+',n>) = <'+',succnat(n)>

R(succ)(<'-',n>) = si egal(n,zeronat) alors <'+',succnat(zeronat)>

sinon <'-',prednat(n)>

R(pred)(<'-',n>) = <'-',succnat(n)>

R(pred)(<'+',n>) = si egal(n,zeronat) alors <'-',succnat(zeronat)>

sinon <'+',prednat(n)>

et on devra s'assurer de la validité des axiomes dans la représentation , c'est à dire que pour tout axiome G = D de "relatif"

R(G) = R(D) est un théorème de "nat" sur le domaine défini par l'invariant de représentation.

II.7. TYPES ABSTRAITS ET SYSTEMES DE REECRITURE

Les systèmes de réécriture sont l'objet de nombreux travaux <HUE77>,<H&O80>,<K&K82> et nous n'en donnerons qu'un bref aperçu , afin de montrer leur utilisation dans le cadre des types abstraits.

II.7.1. Quelques définitions.

un système de réécriture est un ensemble de règles notées G -> D , ou G et D sont des termes composés de symboles d'opérations et de symboles de variables, et tels que l'ensemble des variables de D , noté $\mathcal{V}(D)$, soit inclus dans l'ensemble $\mathcal{V}(G)$ des variables de G.

On appelle alors substitution un ensemble σ de couples (x_i,t_i) ou les x_i sont des variables et les t_i des termes.

Appliquer une substitution σ à un terme t consiste à substituer chaque occurrence des x_i dans t par le terme t_i correspondant, le terme obtenu est noté σt.

On dira alors qu'un terme t se réécrit en un terme t' si il existe une substitution σ ,un sous-terme s de t et une règle g -> d tels que $\sigma g = s$ et t' est le terme obtenu par remplacement du sous-terme s par σd.

On note -> la relation ainsi introduite sur les termes et $\overset{*}{\rightarrow}$ sa fermeture transitive et réflexive, et on dit qu'un terme t est irréductible si

$t \overset{*}{\rightarrow} t' \Rightarrow t = t'$.

Si $t \overset{*}{\rightarrow} t'$ et t' est irréductible alors t' est une forme normale de t.

Le problème se pose alors de savoir , d'une part si étant donné un terme le nombre de réécritures successives qu'on peut lui appliquer est fini, c'est à dire si on aboutit à une forme normale , d'autre part si l'ordre des

réécritures a une importance.

On dit qu'un système de réécriture est noethérien ou a la propriété de terminaison finie , si pour chaque terme t il n'existe pas de suite infinie telle que :

$t \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots$

Cette propriété est en général indécidable , mais de nombreux travaux cherchent à établir des critères suffisants qui la garantissent, l'idée générale étant d'établir un ordre sur les termes et de s'assurer que pour chaque règle le membre droit est "plus petit" que le membre gauche. On citera notamment les travaux de N. Dershowitz <DERs82>, J.P. jouannaud, P. Lescanne et F. Reinig <JLR82>.

On dit qu'un système de réécriture a la propriété de Church-Rosser ou est confluent si pour tout terme t

$t \overset{*}{\rightarrow} t_1$ et $t \overset{*}{\rightarrow} t_2 \Rightarrow$ il existe t' tel que

$t_1 \overset{*}{\rightarrow} t'$ et $t_2 \overset{*}{\rightarrow} t'$.

L'algorithme de Knuth et Bendix <K&B70>, <HUE77> permet de tester la confluence d'un système de réécriture moyennant l'hypothèse qu'il est noethérien.

Un système noethérien et confluent est dit canonique, dans ce cas tout terme se réduit en une forme normale unique.

II.7.2. Utilisation dans le cadre des types abstraits.

Le lien avec les types abstraits apparait dans la mesure ou en II.2. nous avons vu qu'on peut prendre pour support du modèle d'une spécification $\langle S,\Sigma,E\rangle$ un ensemble de classes d'équivalence de mots du langage des termes construits sur Σ.

En orientant les axiomes de façon à définir un système de réécriture convergent (si c'est possible), et en se dotant d'une "fonction de normalisation" associant à tout terme sa forme normale, on définit à la fois une relation de congruence sur les termes par

$$t \equiv t' \iff FN(t) = FN(t')$$

et une représentation canonique de toute classe d'équivalence. Il reste alors à définir pour chaque opération f de Σ, la "fonction de transfert" f_T qui calcule la forme normale d'un terme dominé par $\underline{f}$

$$f_T(FN(x_1)..FN(x_n)) = FN(\ \underline{f}(\ x_1 , .. , x_n\ )\ )$$

on démontre que l'algèbre ainsi obtenue correspond au modèle initial de $\langle S,\Sigma,E\rangle$ <REM82b>.

Cette approche système de réécriture présente l'avantage d'être opérationnelle dans la mesure ou on est capable d'implanter la relation de réécriture et une "fonction de normalisation" (et/ou les "fonctions de transfert") en utilisant un algorithme d'unification, directement comme dans AFFIRM <MUS80>, ou indirectement à travers le langage PROLOG par exemple <B&D81>,<CHA82>,<DERa82>.

II.8. CONCLUSION

A l'issue de ce chapitre on peut dégager deux caractéristiques intéressantes des spécifications algébrique de types :

- elles fournissent la description d'une sémantique pour un ensemble d'objets et pour les opérations associées,
- on peut leur donner une interprétation "exécutable" en terme de système de réécriture.

Ces deux aspects en font à la fois un moyen de description du comportement d'un module dans la bibliothèque d'un système de développement, et un moyen de tester plus tôt (au niveau de la spécification) l'adéquation d'une solution envisagée pour la réalisation d'un logiciel donné. Il resterait, bien entendu, à examiner le moyen de passer d'une telle spécification à un programme dans un langage de programmation classique, mieux adapté au matériel existant. Nous n'aborderons pas ce problème. On fera toutefois remarquer qu'il n'est pas totalement irréaliste d'envisager la démarche inverse et d'essayer d'adapter le matériel aux concepts utilisés (machines d'unification).

Il reste encore beucoup de travail à mener pour savoir comment structurer ou hiérarchiser un ensemble de spécifications, afin d'aboutir à une méthodologie de la spécification. Nous renverrons le lecteur à <B&G77> pour un exposé plus complet sur ces problèmes.

III. BIBLIOTHECAIRE

La conception d'un logiciel produit en général un grand nombre de textes divers. Si l'on considère uniquement les textes de programmes, on peut déjà les trouver sous trois formes différentes :textes sources, textes en langage intermédiaire déstinés à être "édités" par l'éditeur de liens , code exécutable . Si l'on prend en compte les problèmes de versions différentes et de documentation on conviendra aisément de l'utilité d'un outil charyé de la gestion de l'ensemble ainsi constitué, capable de conserver les relations liant ces différents textes et de garantir sa cohérence au fur et à mesure des modifications introduites par le processus de conception. La présence d'un tel ensemble d'informations lors du développement du produit permet non seulement une utilisation documentaire, mais de plus autorise la mise en oeuvre de la modularité de façon sûre en permettant d'effectuer les contrôles de cohérence nécessaires , soit lors de la compilation d'une unité , soit lors de la connexion d'un ensemble d'unités. En effet , lorsque l'on développe une application de façon modulaire , on est amené lors de l'écriture d'un module à en utiliser d'autres ; généralement l'utilisation des divers objets déclarés dans ceux ci est soumise à des contraintes : types des paramètres des procédures , type des variables , visibilité , ... Leurs vérifications manuelles deviennent alors d'autant moins fiables que la taille de l'application augmente . La présence d'une bibliothèque permet l'automatisation de cette tâche ; on assure ainsi que l'utilisation de tout objet est conforme à sa définition .

Le confort d'utilisation d'une "base" ou d'une "bibliothèque" de modules dépendra de l'outil qui la gère (bibliothécaire) et plus particulièrement des fonctions d'accès qu'il fournit . Comme on le verra en III.1 au travers de deux exemples , les informations sur la "structure" d'un ensemble

d'unités de bibliothèque (les liens existants entre modules) fournissent une aide non négligeable .

Disposant d'une telle base de modules , on pourrait souhaiter ne pas avoir à réécrire et à développer des modules ayant le "même comportement" que ceux de la bibliothèque . Le formalisme des types abstraits algébriques présenté dans le chapitre précédent , permet de préciser cette notion de comportement ; il vient alors naturellement à l'esprit de construire une bibliothèque de spécifications et d'essayer d'accéder à ses unités en utilisant les axiomes.

C'est l'objectif que nous nous sommes fixé. L'intérêt d'un outil possédant une telle fonction d'accès est clair :

> on ne s'évite pas l'écriture des spécifications, mais on "hérite" des développements effectués sur les spécifications de la bibliothèque (implantations).

Pour réaliser un tel produit, on est conduit à "comparer" des spécifications , et à définir ce qu'on entend par spécifications équivalentes . Nous présenterons en III.3 quelques exemples qui nous amènerons à définir en III.4 une notion d'équivalence " aux noms près " .

Nous ne chercherons pas à fournir un moyen de preuve de l'équivalence aux noms près de deux spécifications , mais plutôt un algorithme capable d'en émettre l'hypothèse lorsqu'il y a lieu . Cet algorithme , intégré à une bibliothèque , donnera à l'utilisateur un moyen d'accès "intelligent" supplémentaire , même en l'absence d'un démonstrateur de théorèmes , seul capable de valider ou d'invalider les hypothèses émises .

On peut ainsi espérer augmenter la productivité du processus de développement de logiciel en utilisant au maximum l'"acquis" de l'environnement constitué à partir des développements précédents .

III.1. EXEMPLES

Si on se place au niveau des langages de programmation, une unité de bibliothèque pourra être toute unité syntaxique compilable séparément , et les relations entre unités seront obtenues à partir des constructions du langage traduisant les échanges d'informations (listes d'importations, directives "UTILISE",. . .). Une des fonctions du bibliothécaire sera alors de visualiser le graphe de ces relations. Ces notions se transportent naturellement dans les systèmes intégrant un langage de spécifications si celui-ci est modulaire.

III.1.1. Cas d'un langage de programmation.

Un exemple nous est donné dans <CAL80>,<CUN82> comme élément de réponse au cahier des charges de l'Ada Programming Support Environnement déjà cité <DOD80>. Sans vouloir présenter le langage ADA nous rappellerons que la distinction y est faite entre interface et corps d'une unité("SPECIFICATION"/"BODY") et que les unités susceptibles d'être soumises séparemment au compilateur sont au nombre de cinq:

- les spécifications de package
- les spécifications de sous programmes
- les corps de packages
- les corps de sous-programmes
- les sous-unités

Chaque unité de compilation sera une unité du système. Par ailleurs, une spécification de package précise la liste des objets exportés par le package (procédures,types,variables)et utilisables par d'autres unités , ces objets

. seront des "composants" du système. Les relations entre unités seront obtenues à partir des textes qui permettent d'en définir huit:

- "l'unité . . . est le corps de l'unité . . . "
- "l'unité . . . est une sous-unité de l'unité . . . "
- "l'unité . . . utilise l'unité . . . "
- "l'unité . . . utilise le composant . . . de l'unité . . . "

ainsi que leurs inverses. Le système s'articule en deux couches :système de base et système enveloppe

* au niveau du système de base on garantit la cohérence de l'ensemble des unités en vérifiant en particulier :
  . qu'un corps n'est introduit qu'après la spécification correspondante et qu'il y est conforme.
  . qu'une sous-unité n'est introduite qu'après l'unité dont elle est détachée.
  . qu'une unité utilisant une autre unité ("USE") n'est introduite qu'après cette dernière

  et on conserve les relations précitées.
* au niveau du système enveloppe on a la possibilité de s'affranchir (temporairement) de ces contraintes et de conserver des versions différentes pour une "même unité". C'est cette couche qui servira d'interface entre l'utilisateur et le système de base.

Les fonctions de consultation de l'ensemble des unités gérées par un tel système viennent rapidement à l'esprit et exploitent les relations définies entre les unités; après la définition d'un langage de requêtes on sera à même de poser des questions telles que:

Quel est le corps du package X?

Quelles sont les unités utilisées par l'unité X?

Il est clair qu'en l'absence d'informations supplémentaires associées aux

unités les réponses ainsi fournies renseignent plus sur l'"architecture" de l'ensemble que sur son "comportement" , celui ci ne pouvant être connu que par l'examen des algorithmes mis en oeuvre dans les textes des corps. Nous nous garderons bien de négliger ces informations , précieuses aussi bien pour la construction structurée d'une application que pour sa mise au point et sa maintenance. De plus on peut construire des systèmes comparables quel que soit le langage utilisé pour peu que l'on puisse y définir des unités et des relations analogues ainsi qu'il est montré dans <ROY83>.

III.1.2. Cas des langages de spécification.

Le projet de conception de programmes assistée par ordinateur SPRAC <SPR81>,<FOI82> (Système de Programmation par Réutilisation Assistée des Connaissances) apparait comme un projet ambitieux cherchant à bâtir un système d'aide à la réalisation et à la validation de logiciels par l'utilisation d'une "base de connaissances" dans le domaine d'application retenu. Mettant l'accent sur la "réutilisation" il est proche de nos préoccupations.

Le système propose trois niveaux de langages:

- le langage LF , langage de spécification de fonctions et de types abstraits de données est un sous-ensemble du calcul des prédicats du premier ordre. Il impose la définition d' une fonction ou d' un type à la fois et autorise l'importation de fonctions et de types (partie "USE"). Un type de données abstrait (ADT) peut être générique et comporte :
  * une partie "INTERFACE" précisant les fonctions du type visibles à l'extérieur ,

* une partie "CONSTRUCTORS" ou on nomme les opérations génératrices du type

* une partie "SPECS" définissant les opérations.

Après transformation les spécifications sont interprétables par un interprète du type "PROLOG".

- le langage LA est un langage, purement fonctionnel, d'expression d'algorithmes contenant les structures de contrôle habituelles. Il permet de donner la définition algorithmique d'une fonction ou des fonctions d'un ADT definies en LF.

- le langage LM est un langage de programmation classique

Ces trois couches de langages conduisent à trois types d'objets pour les fonctions et pour les types de données:fonctions-algorithmes-modules, types abstraits de données-représentations d'ADT-concrétisations d'ADT. On définit par là même les relations "représenté par" entre un objet du niveau abstrait et un objet du niveau algorithmique , et "concrétisé par" entre un algorithme et un texte en langage LM . Ces relations viennent s'ajouter à la relation "utilise" issue des listes d'importations.

L'ensemble ainsi constitué est géré par deux bases de données , une Base de Données Projet (BdP) et une Base de Connaissances (BdC).

La BdP est constituée des ensembles de versions rangées dans un historique. A chaque version est associée un "graphe de développement" et un "agenda". Le graphe de développement permet de connaitre l'état d'une version et des unités qui la compose, l'agenda représente la liste des tâches à effectuer pour la terminer. C'est par l'intermédiaire de cette BdP que le système pourra guider l'utilisateur en lui proposant des tâches à accomplir afin de compléter un graphe de développement. On retrouve là des préoccupations analogues à celles présentées dans <CJQ80>.

La BdC contient des connaissances sur le domaine d'application , c'est à

dire en pratique des descriptions de fonctions ou d'ADT sous les trois formes possibles ainsi que les relations qui les lient. Ces objets auront été testés ou prouvés et seront considérés comme valides. De plus, le modèle de BdC présenté dans <SPR81> associe aux unités des "mots-clés" et des "synonymes". L'interrogation de la BdC peut alors se faire à l'aide d'un ensemble de questions prédéfinies utilisant les mots-clés, les synonymes, et les relations déjà citées. La définition d'un langage d'interrogation plus sophistiqué permettra de répondre aux mêmes questions que précédemment ,la possibilité d'adjontion de mots-clés et de synonymes introduisant des possibilités supplémentaires par rapport aux relations purement structurelles "représenté par" , "concrétisé par","utilise".

III.2. QUELQUES REMARQUES

Ces deux exemples laissent penser que, quels que soient les langages utilisés pour aboutir à un logiciel, on peut en se fondant sur la syntaxe définir ce que seront les objets et les relations intéressantes à conserver dans la bibliothèque. On peut de même adjoindre des attributs tels que mot-clé ou synonymes afin d'associer des "thèmes" aux textes conservés , l'utilisation que l'on en fait est alors analogue à celle des mots-clés figurant dans l'entête d'une communication. Comme nous l'avons déjà dit ces fonctions ne sont pas négligeables, on peut toutefois leur reprocher de donner des informations plus globales à une application que locales à une unité et de ce fait de nécessiter une phase plus ou moins longue d'apprentissage du contenu de l'environnement ainsi défini. Si des accès par mot-clé sont possibles on devra se familiariser avec le vocabulaire utilisé

et avec sa signification. En tout état de cause les systèmes proposés reposent sur des accès par les "noms". En pratique, si on se place dans un domaine d'application précis, on doit reconnaitre qu'il est relativement aisé d'obtenir un consensus sur les unités indispensables dans l'environnement et sur un vocabulaire commun de synonymes et de mots-clés. De ce fait l'approche de SPRAC semble raisonnable. Si le domaine l'exige on peut même envisager une structuration du vocabulaire utilisé pour aboutir à un véritable système documentaire. On aimerait toutefois en faire "plus".

Comment envisager ce "plus" ?

Plaçons nous dans le contexte d'une bibliothèque d'unités (de spécifications et/ou d'algorithmes et/ou de programmes) "opérationnelles" (interprétables ou exécutables), indépendamment du ou des langages utilisés; une aide effective à la résolution d'un problème donné par utilisation de cette bibliothèque peut se voir de diverses façons selon la forme des questions admises par le bibliothécaire:

-Etant donné un problème exprimé dans un langage d'énoncé existe-t-il une unité solution du problème?

Il est clair que , compte tenu de l'état de l'art , la réalisation d'un système capable de répondre à ce genre de question dans un domaine d'application général n'est pas envisageable à moyen terme. On notera toutefois que ce type de problème est différent de la construction d'une solution à partir d'un énoncé <PAI79>,<FIN79>,les questions et les réponses s'énoncent différemment. Prenons l'exemple de la résolution d'une équation du second degré :

. Si on dispose d'une bibliothèque B d'unités écrites dans un langage de définition de solution (langage algorithmique par exemple) et que l'on cherche dans cette bibliothèque une fonction f calculant les racines d'une équation du second degré, on posera :

"Existe-t-il une fonction f dans B de profil

f : réel x réel -> réel x réel U (indéfini,indéfini) telle que

$f(a,b,c)=(x,y)$ et $ax^2+bx+c=0$ et $ay^2+by+c=0$?"

une réponse pourra alors être:

"Oui, utilisez pour f(a,b,c) la fonction racine2dg(a,b,c)"

ou

"Non, pas de telle fonction en bibliothèque. "

. Si on désire construire une telle fonction, on demandera :

"Construire une fonction f de profil

réel x réel x réel -> réel x réel telle que

$f(a,b,c)=(x,y)$ et $ax^2+bx+c=0$ et $ay^2+by+c=0$!"

une réponse possible sera:

"f(a,b,c)=si $b^2-4ac < 0$ alors (indéfini,indéfini)

sinon si $b^2-4ac=0$ alors $(-b/2a,-b/2a)$

sinon$((-b+\sqrt{b^2-4ac})/2a,(-b-\sqrt{b^2-4ac})/2a)$ ".

La solution fournie pourra utiliser des unités de la bibliothèque, mais celle ci ne contiendra pas nécessairement une unité solution du problème posé. Eventuellement le système pourra répondre qu'il ne parvient pas à construire la fonction désirée.

-Un autre type de question envisageable est:

Etant donnée la description d'une unité existe-t-il dans la bibliothèque une unité "équivalente" ?

Le problème est sensiblement différent du précédent puisque le langage est cette fois unique , mais il faut définir cette notion d'"équivalence". Bien évidemment celle ci ne se réduit pas à une équivalence syntaxique; intuitivement on a envie de parler d'équivalence de comportements, ou d'unités qui "font la même chose". Si on se place dans le contexte d'un système tel que SPRAC on peut se poser ce type de question dans chacune des trois couches LF,LA,LM c'est à dire comparer des spécifications , des algorithmes ou des programmes. A priori et bien que des essais de comparaison de petits programmes FORTRAN aient été tentés <ADAm78> et <LAU78> , il semble plus intéressant de se poser d'abord le problème au niveau des unités de spécification , on héritera dans ce cas des unités des couches inférieures en relation avec l'unité recherchée. De plus c'est au niveau des spécifications, parce c'est le niveau le plus abstrait, qu'on définira l'essentiel des comportements.

Indépendamment des méthodes utilisées pour aboutir à une spécification et des étapes de développement conduisant au(x) programme(s), l'utilisation d'un langage de spécification, dans un système comprenant une bibliothèque ou une "base" d'unités,

* permet d'effectuer des preuves de théorèmes portant sur les fonctions et les types de données que l'on définit, preuves manuelles ou partiellement automatisables comme dans AFFIRM ou OASIS <BJM82>,
* permet d'effectuer des "tests" de spécifications si on peut l'interpréter et permet donc de vérifier plus tôt lors de la démarche de conception la

validité des choix effectués par rapport aux besoins de l'utilisateur final,
* enfin, et c'est l'aspect essentiel pour notre travail, fournit une possiblité d'accès supplémentaire aux objets de l'environnement si on réussit à définir cette notion vague pour l'instant d'"équivalence" , et un outil capable de l'exploiter et économise de ce fait une partie du travail de développement.

III.3. EXEMPLES DE SPECIFICATIONS "EQUIVALENTES" DE TYPES ABSTRAITS ALGEBRIQUES

Nous allons maintenant essayer d'illustrer par des exemples la notion de spécifications équivalentes citée dans ce qui précède. Nous irons un peu au delà en montrant ce qu'on pourrait appeler des "ressemblances" dont on souhaiterait qu'elles puissent être décelées par un bibliothécaire plus sophistiqué. Nous nous placerons dans le cadre des spécifications de types abstraits algébriques . La syntaxe utilisée est décrite complètement en annexe, afin de faciliter la lecture nous précisons le rôle des différents symboles:

profils introduit la liste des opérations du type et de leurs profils sous la forme

(liste des domaines) : codomaine ;

var introduit la liste des variables universellement quantifiées utilisées dans les équations ainsi que leurs types

axiomes introduit la liste des axiomes; chaque axiome est de la forme

g -> d ;

et est interprété comme une règle de réécriture.

Par ailleurs erreur désigne tout "terme-erreur" , la propagation des erreurs étant faite implicitement . On se place pour ces exemples dans un environnement ou les types "booléen" et "entier" sont définis . Le modèle sémantique associé à une présentation est le modèle initial.

III.3.1. Exemple 1.

Considérons les deux présentations suivantes :

```
type listel
  profils
     nill : listel;
     makel (entier) : listel;
     appendl (listel,listel) : listel;
     consl (entier,listel) : listel;
     dell (listel) : listel;
  var
     i :entier;
     l1,l2 :listel;
  axiomes
     appendl(nill,l1) -> l1;
     appendl(makel(i),nill) -> makel(i);
     appendl(appendl(makel(i),l1),l2)->appendl(makel(i),appendl(l1,l2));
     consl(i,l1) -> appendl(makel(i),l1);
     dell(nill) -> erreur ;
     dell(appendl(makel(i),l1)) -> l1;
fin
```



```
type liste2
  profils
     nil2 : liste2;
     cons2 (entier,liste2) : liste2;
     append2 (liste2,liste2) : liste2;
     make2 (entier) : liste2;
     del2 (liste2) : liste2;
  var
     i:entier;
     l1,l2 :liste2;
  axiomes
     make2 (i) -> cons2 (i,nil2);
     append2 (nil2,l1) -> l1;
     append2 (cons2(i,l1),l2) -> cons2 (i,append2(l1,l2));
     del2 (cons2(i,l1)) -> l1;
     del2 (nil2) -> erreur ;
fin
```

Ces deux spécifications sont "équivalentes" dans la mesure ou si on substitue dans la première les symboles listel,nill,consl,appendl,dell respectivement par liste2,nil2,cons2,append2,del2 les spécifications obtenues possédent le même modèle initial, c'est à dire que les classes d'équivalence définies sur les termes par la plus petite congruence engendrée par les axiomes sont les mêmes dans chaque spécification; on en trouvera la preuve dans <R&V81> . On notera que les deux spécifications diffèrent notablement par leurs axiomes qui conduisent à des formes normales formées , dans un cas des opérateurs nill,makel,appendl et dans l'autre de

nil2,cons2 . La relation d'équivalence aux noms près qui apparait dans cet exemple, permet d'affirmer intuitivement que si un des deux types "listel" ou "liste2" est présent dans l'environnement considéré on peut se dispenser d'y faire figurer le deuxième , sauf si des critères supplémentaires tels que lisibilité ou efficacité interviennent. Plus exactement, la relation d'équivalence existant entre les deux spécifications autorise l'utilisation de l'une à la place de l'autre.

III.3.2. Exemple 2.

Considérons la spécification "listel" précédente et ajoutons dans la partie profils l'opération :

premierl (listel) :entier;

et dans la partie axiomes les règles:

premierl(nill) -> erreur;

premierl(makel(i)) -> i;

premierl(appendl(makel(i),ll)) -> i;

le type "liste2" restant inchangé.

On devrait s'assurer que l'on n'a pas introduit d'inconsistance dans la présentation de "listel" ainsi enrichie , ni dans l'environnement . Ceci est garanti par la forme des axiomes définissant "premierl" sur les formes normales de "listel" et par le fait qu'un terme premierl(t) se réduit en un terme ne contenant pas d'occurrence de l'opérateur "premierl". Les types "listel" et "liste2" ne sont plus "équivalents" , on pourrait toutefois introduire dans la spécification "liste2" une fonction "premier2" et les axiomes correspondants pour obtenir ce résultat.

En pratique il n'est pas indifférent de savoir qu'une "partie" de "listel"

est équivalente à "liste2" ; disposant de "la plus grande" on n'a pas besoin d'introduire l'autre dans l'environnement , inversement il peut être avantageux de modifier la "plus petite" et les unités qui l'implantent (soit en terme de types abstraits , soit en terme de langage algorithmique comme dans SPRAC) plutot que de refaire tout le travail.

On peut poursuivre ce mécanisme d'enrichissement à loisir:

considérons le type "listel" enrichi de "premierl" comme précédemment , et le type "liste2" enrichi de

hauteur2 (liste2):entier;

hauteur2 (nil2) -> 0;

hauteur2 (cons2(i,ll)) -> hauteur2(ll) + 1;

Les types ainsi enrichis ne vérifient plus cette relation d'"inclusion" évoquée , la mise en évidence de leur partie "commune" présenterait cependant les mêmes avantages. L'inconvénient est que pour deux types quelconques on peut être amené à toujours trouver une telle partie commune , ainsi pour une pile et une file on trouvera un sous-ensemble des opérations répondant à ce critère : (vide,empiler) , (vide ajouter) . Si on le prend en compte on introduira ainsi beaucoup de "bruit" dans les réponses à moins d'être capable de fixer un seuil en dessous duquel la réponse n'est pas pertinente,par ailleurs on augmente la complexité du problème à résoudre.

III.3.3. Exemple 3.

Considérons le type "liste2" de l'exemple 1 mais changeons le profil de l'opération "cons2"

```
cons2 (liste2,entier) : liste2;
```

ainsi que les règles ou "cons2" apparait

```
make2(i) -> cons2(nil2,i);
append2(cons2(l1,i),l2) -> cons2(append2(l1,l2),i);
del2(cons2(l1,i)) -> l1;
```

on conviendra aisément que cette modification mineure ne devrait pas affecter grandement la relation existant avec le type "liste1" . Le problème se complique si on effectue le même type de permutation sur les deux arguments de "append2" , on définit ainsi une opération de concaténation du premier argument " à gauche " du second . Cette fois le profil n'est pas modifié mais les axiomes deviennent :

```
append2(l1,nil2) -> l1;
append2(l2,cons2(i,l1)) -> cons2(i,append2(l1,l2));
```

Dans le premier cas une normalisation de la présentation en ordonnant les profils conformément à un ordre sur les noms des types (ordre d'introduction dans l'environnement par exemple) permet de se ramener aux cas précédents , il n'en est pas de même lorsqu'il y a plusieurs occurrences du même type dans le profil.

III.3.4. Exemple 4.

Remplaçons dans la présentation de "liste1" de l'exemple 1 toutes les occurrences du mot "entier" par le mot "caractère" , on définit ainsi une liste-de-caractères au lieu d'une liste-d-entiers , la différence peut encore apparaitre négligeable et justifier une tentative de prise en compte de ce type de "ressemblances" . On touche là sans le dire à la notion de paramétrisation . Intuitivement le nom du type des éléments de la liste joue le rôle d'un paramètre formel pour lequel aucune propriété n'est éxigée ;on conçoit des listes de "n'importe quoi". Dans un tel cas on pourrait envisager de faire figurer dans la spécification une liste de noms de types paramètres, on devra alors associer ces noms de la même façon qu'on le fait pour les noms d'opérations . Si on s'autorise de plus des permutations sur l'ordre des domaines dans les profils le problème devient complexe car on ne dispose plus de critère de normalisation des profils.

Dans le cas général lorsqu'on paramétrise on souhaite munir les paramètres formels d'opérations sans fournir une spécification complète comme on le fait pour le type d'intérêt . On préfère doter les opérations "formelles" de lois telles que associativité,commutativité,ou affirmer qu'elles définissent des relations d'ordre , d'équivalence ... On sort alors du cadre que nous nous sommes fixé.

III.3.5. Premières conclusions.

Par ces quelques exemples nous avons voulu montrer quelques relations entre types dont on souhaiterait qu'elles puissent être découvertes par un bibliothécaire sophistiqué. L'imagination étant sans limite on pourrait en souhaiter d'autres : deux "piles" qui ne différent que par la règle

dépiler(vide) -> vide pour l'une et

dépiler(vide) -> erreur pour l'autre

peuvent être considérées comme très "voisines". On voit vite les dangers d'un tel engrenage. En fait les motivations qui susciteraient un tel besoin sont différentes des précédentes :

dans un cas on a construit une spécification dont on désire savoir si elle n'est pas présente dans l'environnement afin d'éviter les travaux de développements ultérieurs ,

dans l'autre on construit partiellement une spécification en exhibant quelques axiomes et on souhaite obtenir une aide pour la compléter en s'appuyant sur des spécifications existantes.

L'"équivalence" que l'on cherche à définir semble pouvoir s'énoncer en terme de "renommage" des symboles et de permutations sur l'ordre des arguments des fonctions afin de se ramener à une notion plus habituelle de spécifications équivalentes . Pour d'éventuelles spécifications paramétrées on devra de même "renommer" les noms des paramètres formels et s'assurer qu'ils sont munis des "mêmes" contraintes <EHR81>. On pourrait de plus souhaiter que l'ordre d'apparition des paramètres dans la spécification ne soit pas significatif . Compte tenu des remarques faites en III.3.4 , le problème devient beaucoup plus complexe et nous ne l'aborderons pas dans ce travail . Pour ce qui est des spécifications enrichies on peut se ramener au cas précédent par l'utilisation de contraintes méthodologiques en demandant au

spécifieur qu'il distingue syntaxiquement les opérations d'enrichissement de celles qui sont indispensables à la définition du "noyau".Ceci allègera la tâche du bibliothécaire et évitera un nombre trop important de réponses plus ou moins pertinentes lors des interrogations.

III.4. VERS UNE FORMALISATION

Nous allons maintenant tenter de définir plus précisément la relation présentée intuitivement dans les lignes précédentes .Cette relation peut se définir comme une équivalence "aux noms près" , on verra qu'elle n'est pas totalement suffisante pour permettre l'obtention de tous les résultats souhaités.

Nous rappellerons donc la définition classique de l'équivalence de deux spécifications ainsi que quelques façons d'en conduire la preuve. Cette définition exigeant en particulier l'égalité des signatures des deux présentations ne correspond pas au résultat recherché, on peut cependant essayer de se ramener à ce cas de figure après avoir renommé les symboles figurant dans l'une des deux spécifications en présence .On héritera ainsi partiellement des résultats concernant la conduite des preuves.

Une démarche analogue sera utilisée pour définir la notion d'extension à un renommage près.

III.4.1. Equivalence de deux spécifications.

Nous allons tout d'abord rappeler la définition de l'équivalence de deux spécifications au sens habituel.

Définition 1:

Deux présentations $\langle S,\Sigma,E\rangle$ et $\langle S',\Sigma',E'\rangle$ sont équivalentes si et seulement si

1. $S=S'$
2. $\Sigma=\Sigma'$
3. Les axiomes de E sont des formules valides dans l'algèbre initiale définie par les axiomes de E' (et ceux de l'environnement) et réciproquement.

ou encore :

3'. Leurs algèbres initiales sont isomorphes.

ou encore :

3''. La plus petite congruence $\equiv_E$ définie sur l'ensemble des termes $T_{S,\Sigma}$ est égale à la plus petite congruence $\equiv_{E'}$, définie sur l'ensemble des termes $T_{S',\Sigma'}$.

Remarque :

Pour montrer la validité d'une formule dans l'algèbre initiale on utilise en pratique le raisonnement par induction.

Ces définitions s'énoncent plus intuitivement en disant que dans les deux spécifications on utilise les mêmes noms de types ,les mêmes noms

d'opérations avec les mêmes profils et que deux termes égaux dans l'une le sont aussi dans l'autre.

III.4.2. Preuves d'équivalence.

Pour prouver l'équivalence de deux spécifications on devra , après s'être assuré de l'égalité des signatures , prendre chaque axiome de l'une ,le prouver en utilisant les axiomes de l'autre (éventuellement en raisonnant par induction structurelle sur les opérations de la signature ou sur les constructeurs),puis effectuer le même travail en inversant le rôle des spécifications.En pratique on essaiera d'abord de prouver un lemme de forme normale pour chacun des types de façon à diminuer le nombre de cas dans le raisonnement par induction .

M. Bidoit propose dans <BID81> une méthode systématique et partiellement automatisable pour transformer une spécification en une spécification équivalente pour laquelle les formes normales sont différentes.

J.L. Rémy et P.A.S Veloso proposent eux dans <R&V81> de prouver les axiomes de T' dans T comme précédemment , puis de faire correspondre aux formes normales de T' leurs formes normales dans T et de vérifier que les théorèmes ainsi obtenus sont valides dans T' c'est à dire que :

$$\text{pour tout } x \text{ dans } FN_{T'},\ \text{on a } FN_T(x) \equiv_{T'} x$$

on a ainsi

$$E \models E' \text{ et } y \equiv_{T'} x \Rightarrow y \equiv_T x$$

Ceci permet d'économiser la preuve d'un certain nombre de théorèmes.

III.4.3. Notion de ϱ-équivalence.

Nous allons maintenant présenter un certain nombre de définitions permettant d'aboutir à l'introduction de la notion d'équivalence recherchée. Celle ci nécessite la définition d'une application dont le domaine est l'ensemble des symboles de l'une et le codomaine l'ensemble des symboles de l'autre , qu'on appellera renommage . Par extension on parlera de terme renommé , de spécification renommée et d'algèbre renommée .

Définition 2:

Soient (S,Σ) et (S',Σ') deux signatures ,on appelle renommage de (S,Σ) dans (S',Σ') toute application injective r de $S+\Sigma$ dans $S'+\Sigma'$ (associant les symboles de sortes aux symboles de sortes et les symboles d'opérations aux symboles d'opérations)
telle que
pour tout o dans Σ
$profil(o)=s_1..s_n \rightarrow s \Leftrightarrow profil(r(o))=r(s_1)..r(s_n) \rightarrow r(s)$

Nous étendons maintenant la notion de renommage de signature à celle de renommage de termes.

Définition 3:

Soient (S,Σ) et (S',Σ') deux signatures ,r un renommage de (S,Σ) dans (S',Σ') et t dans T_Σ , on appelle terme t renommé par r et on note r(t) le terme obtenu en remplaçant tous les symboles o figurant dans t

et appartenant à Σ, par r(o) c'est à dire :

r(t)= si t est une variable alors t sinon

si $t=o(t_1..t_n)$ alors $r(o)(r(t_1)..r(t_n))$

Définition 4:

Soient T :<S,Σ,E> et T' :<S',Σ',E'> deux spécifications ,r un renommage de (S,Σ) dans (S',Σ') , on appelle spécification T renommée par r et on note r(T) la spécification <r(S),r(Σ),r(E)> avec :

$r(S)=\{r(s) \text{ tels que } s \text{ dans } S\}$

$r(\Sigma)=\{(r(o) :r(s_1)..r(s_n)\rightarrow r(s)) \text{ tels que } (o :s_1..s_n\rightarrow s) \text{ dans } \Sigma\}$

$r(E)=\{(r(g)=r(d)) \text{ tels que } (g=d) \text{ dans } E\}$

Définition 5:

On appelle algèbre $T_{S,\Sigma,E}$ renommée par r et on note $r(T_{S,\Sigma,E})$ la $(r(S),r(\Sigma))$-algèbre $T_{r(S),r(\Sigma),r(E)}$.

On peut maintenant définir la relation de ϱ-équivalence :

Définition 6:

On dira que deux spécifications $\langle S,\Sigma,E\rangle$ et $\langle S',\Sigma',E'\rangle$ sont équivalentes à un renommage près ou ϱ-équivalentes et on notera :

$$T_{S',\Sigma',E'} \overset{\varrho}{\simeq} T_{S,\Sigma,E}$$

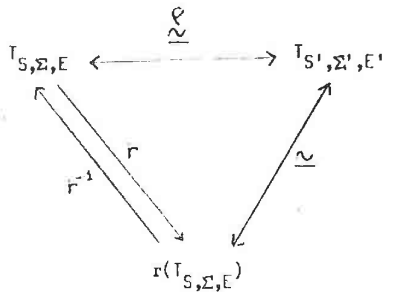
si et seulement si il existe un renommage surjectif r tel que :

$$T_{S',\Sigma',E'} \simeq r(T_{S,\Sigma,E})$$

Remarque :

on a alors $\equiv_{r(E)} = \equiv_{E'}$

On obtient ainsi le diagramme :



On définit bien une relation d'équivalence car :

- $T_{S,\Sigma,E} \overset{\varrho}{\simeq} T_{S,\Sigma,E}$ avec le renommage identité

- $T_{S,\Sigma,E} \overset{\varrho}{\simeq} T_{S',\Sigma',E'} \Leftrightarrow T_{S',\Sigma',E'} \overset{\varrho}{\simeq} T_{S,\Sigma,E}$ r étant bijective
- $T_{S1,\Sigma1,E1} \overset{\varrho}{\simeq} T_{S2,\Sigma2,E2}$ et $T_{S2,\Sigma2,E2} \overset{\varrho}{\simeq} T_{S3,\Sigma3,E3}$
  $\Rightarrow T_{S1,\Sigma1,E1} \overset{\varrho}{\simeq} T_{S3,\Sigma3,E3}$ par composition des renommages.

Exemple :

Les spécifications de "listel" et "liste2" de l'exemple 1 sont ϱ-équivalentes en effet l'application r définie par :

r(listel)=liste2

r(entier)=entier

r(nill)=nil2

r(consl)=cons2

r(appendl)=append2

r(dell)=del2

r(makel)=make2

est bien un renommage étant bijective et conservant les profils ,et "listel" renommée par r est équivalente à "liste2" (cf <R&V81>).

III.4.4. Théorème caractéristique de la ϱ-équivalence.

La définition précédente semble répondre en partie aux préoccupations exprimées en III.3 ,il est clair que d'un point de vue pratique le problème essentiel va être la mise en évidence d'un renommage entre deux spécifications . Nous allons essayer de fournir une proposition plus constructive en nous situant dans le cadre de systèmes de réécriture canoniques(confluent noethérien). Dans un tel cadre on peut représenter toute classe d'équivalence de termes dans l'algèbre initiale par une forme normale unique .On voit bien intuitivement que si il existe un renommage

entre deux spécifications , pour qu'elles soient $\wp$-équivalentes on devra être capable d'associer les formes normales de l'une aux formes normales de l'autre ; ce de façon biunivoque ; cette application dépendra à la fois du renommage et des axiomes des présentations.

Proposition 1 : théorème caractéristique

Soient $T : \langle S,\Sigma,E\rangle$ et $T' : \langle S',\Sigma',E'\rangle$ deux spécifications telles que E et E' soient des systèmes de réécriture canoniques , T et T' sont $\wp$-équivalentes si et seulement si il existe un renommage surjectif r de (S,Σ) dans (S',Σ') et une application f injective de $FN(T_{S,\Sigma,E})$ dans $FN(T_{S',\Sigma',E'})$ tels que :

pour tout t,t_i dans $FN(T_{S,\Sigma,E})$, pour tout $o : s_1..s_n \to s$ dans Σ

on ait

$o(t_1..t_n) \equiv_E t \Rightarrow r(o)(f(t_1)..f(t_n)) \equiv_{E'} f(t)$ (I)

Preuve :

1)

montrons que $T \overset{\wp}{\sim} T' \Rightarrow$ il existe un renommage surjectif r de (S,Σ) dans (S',Σ') et une application f injective de $FN(T_{S,\Sigma,E})$ dans $FN(T_{S',\Sigma',E'})$ tels que :

pour tout $t,t_1..t_n$ dans $FN(T_{S,\Sigma,E})$, pour tout $o : s_1..s_n \to s$ dans Σ

on ait

$o(t_1..t_n) \equiv_E t \Rightarrow r(o)(f(t_1)..f(t_n)) \equiv_{E'} f(t)$

soient $o,t_1,..,t_n,t$ tels que

$o(t_1..t_n) \equiv_E t$ (HYP)

par un renommage r quelconque on a

(1) $r(o)(r(t_1)..r(t_n)) \equiv_{r(E)} r(t)$

comme $T \overset{\wp}{\sim} T'$, il existe r tel que $\equiv_{r(E)} = \equiv_{E'}$ donc tel que

(1') $r(o)(r(t_1)..r(t_n)) \equiv_{E'} r(t)$

or par définition on a

(2) $FN_{T'}(r(x)) \equiv_{E'} r(x)$ pour tout x dans $T_{S,\Sigma,E}$

d'ou par (2) la formule :

(3) $r(o)(FN_{T'}(r(t_1))..FN_{T'}(r(t_n))) \equiv_{E'} r(o)(r(t_1)..r(t_n))$

en appliquant (1') $\equiv_{E'} r(t)$

en appliquant (2) $\equiv_{E'} FN_{T'}(r(t))$

pour obtenir (I) , il suffit donc de poser :

$f(t) = FN_{T'}(r(t))$

f est injective , en effet

$t \neq t' \Rightarrow r(t) \not\equiv_{E'} r(t')$ par définition de la $\wp$-équivalence

$\Rightarrow FN_{T'}(r(t)) \neq FN_{T'}(r(t'))$ par définition des formes normales

2)

Montrons maintenant la réciproque :

Soient un renommage surjectif r de (S,Σ) dans (S',Σ') et une

application f injective de $FN(T_{S,\Sigma,E})$ dans $FN(T_{S',\Sigma',E'})$ tels que :

pour tout t_i dans $FN(T_{S,\Sigma,E})$, pour tout $o : s_1..s_n \to s$ dans Σ on ait

(H) $\quad o(t_1..t_n) \equiv_E t \Rightarrow r(o)(f(t_1)..f(t_n)) \equiv_{E'} f(t)$

on veut montrer qu'alors $T \overset{\rho}{\simeq} T'$.

Montrons d'abord le lemme suivant :

pour tout x dans T_Σ , pour tout t dans $FN(T_{S,\Sigma,E})$
on a $\quad x \equiv_E t \Rightarrow r(x) \equiv_{E'} f(t)$

Démonstration :

par récurrence sur les termes

a)

posons $x = op_0$

soit $t_0 = FN_T(op_0)$ alors $op_0 \equiv_E t_0$

et $r(op_0) \equiv_{E'} f(t_0)$ par l'hypothèse (H)

b)

posons $x = op(x_1..x_n)$

Soient $t_1,..,t_n$, t dans $FN(T_{S,\Sigma,E})$

tels que $\quad t_1 \equiv_E x_1,..,t_n \equiv_E x_n$, $t \equiv_E x$

on a $\quad op(t_1..t_n) \equiv_E t$

et par l'hypothèse (H) $\quad r(op)(f(t_1)..f(t_n)) \equiv_{E'} f(t) \qquad$ i)

or par définition d'un terme renommé

$r(x) = r(op(x_1..x_n)) = r(op)\ (r(x_1)..r(x_n))$

$\equiv_{E'} r(op)\ (f(t_1)..f(t_n))$ par hypothèse de récurrence

$\equiv_{E'} f(t)$ par i)

Le lemme étant démontré prouvons que :

pour tout x,x' dans T_Σ on a $x \equiv_E x' \Leftrightarrow r(x) \equiv_{E'} r(x')$

$\Rightarrow$)

Soient x,x' dans T_Σ avec $x \equiv_E x'$

et soit $\quad t$ dans $FN(T_{S,\Sigma,E})$ tels que

$x \equiv_E t \equiv_E x'$

par application du lemme on a alors

$r(x) \equiv_{E'} f(t) \equiv_{E'} r(x')$

$\Leftarrow$) réciproquement

Soient t,t' dans $FN(T_{S,\Sigma,E})$ tels que

$t \equiv_E x \quad$ et $\quad t' \equiv_E x'$

$r(x) \equiv_{E'} r(x') \quad \Rightarrow \quad f(t) \equiv_{E'} f(t')$

$\Rightarrow \quad f(t) = f(t')$ puisque ces termes sont des formes normales

$\Rightarrow \quad t = t' \quad$ car f est injective

$\Rightarrow \quad x \equiv_E x'$

ce qui termine la preuve de la proposition 1.

On a de plus :

pour tout x dans T_Σ $\quad f(FN_T(x)) = FN_{T'}(r(x))$

En effet

$x \equiv_E FN_T(x) \quad \Rightarrow \quad r(x) \equiv_{E'} f(FN_T(x))$ par application du lemme

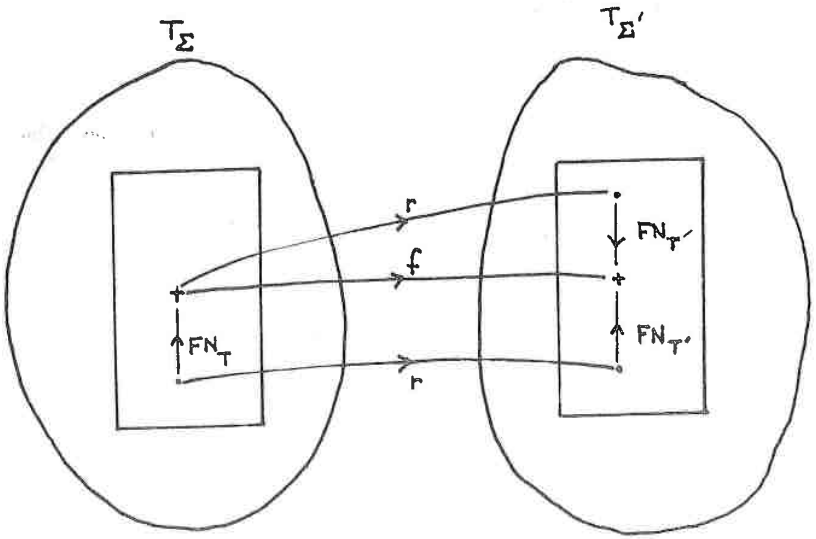
or

$r(x) \equiv_{\mathcal{E}'} FN_{T'}(r(x))$ par définition des formes normales

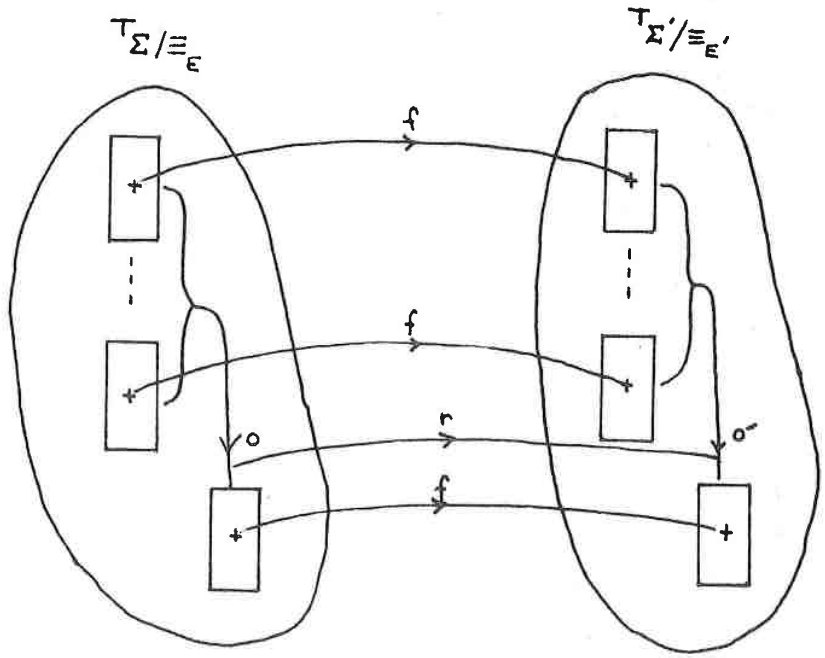
d'où

$f(FN_T(x)) = FN_{T'}(r(x))$ par unicité de la forme normale.

Les différentes formules encadrées apparaissent dans la démonstration précédente sont caractéristiques de la ρ-équivalence. Elles établissent le lien entre le renommage et l'application f portant sur les formes normales. On peut les visualiser par le schéma suivant, ou les "+" notent les termes canoniques, les "." des termes, et ou on représente une classe d'équivalence de termes dans chacun des ensembles T_Σ et $T_{\Sigma'}$ par un rectangle :



La formulation retenue pour le théorème caractéristique met plus en évidence la distinction opérations/objets (classes d'équivalence représentées par un terme canonique) :



III.4.5. Extension à un renommage près.

Le théorème caractéristique de la ρ-équivalence présente l'intérêt d'avoir une formulation dissymétrique :

la formule (I) est une implication logique et l'équivalence de ses deux membres provient de la surjectivité du renommage .

Pour établir la ρ-équivalence on essaiera de construire un renommage r (injectif par définition) et une fonction f satisfaisant (I) , et on s'assurera que r est surjectif . S'il ne l'est pas , on met cependant en évidence une propriété interessante entre les deux spécifications qui touche aux notions d'extension et d'enrichissement . La ρ-équivalence en est alors un cas particulier.

Définition 7:

Soient $T:\langle S,\Sigma,E\rangle$ et $T':\langle S',\Sigma',E'\rangle$,on dira que T' est une extension de T et on notera :

$$T_{S,\Sigma,E} \subseteq T_{S',\Sigma',E'}$$

si et seulement si :

$S \subseteq S'$

$\Sigma \subseteq \Sigma'$

et pour tout t,t' dans $T_{S,\Sigma}$

$$t \equiv_E t' \iff t \equiv_{E'} t'$$

Remarque :

Etant donné une signature (S',Σ') contenant une signature (S,Σ) et une (S',Σ')-algèbre A', on peut définir deux types de (S,Σ)-réduite :

** $A_1 = A'|_{S,\Sigma}$ est définie par :

$$A_1 = A'_s \text{ pour } s \text{ dans } S$$

$$o_A = o_{A'} \text{ pour } o \text{ dans } \Sigma$$

** $A_2 = A'|_{\langle S,\Sigma\rangle}$ est la plus petite (S,Σ)-algèbre incluse dans $A'|_{S,\Sigma}$ c'est à dire l'algèbre engendrée par les Σ-opérations.

Avec ces notations

T' est une extension de T si et seulement si $T_{S',\Sigma',E'|_{\langle S,\Sigma\rangle}}$ et $T_{S,\Sigma,E}$ sont isomorphes.

Définition 8:

Soient $T:\langle S,\Sigma,E\rangle$ et $T':\langle S',\Sigma',E'\rangle$,on dira que est T' est une φ-extension de T et on notera :

$$T_{S,\Sigma,E} \overset{\varphi}{\subseteq} T_{S',\Sigma',E'}$$

si et seulement si :

il existe un renommage r de $S+\Sigma$ dans $S'+\Sigma'$ tel que :

$$r(T_{S,\Sigma,E}) \subseteq T_{S',\Sigma',E'}$$

Proposition 2:

Une spécification T' est une φ-extension d'une spécification T si et seulement si il existe un renommage r de (S,Σ) dans (S',Σ') et une application f injective de $FN(T_{S,\Sigma,E})$ dans $FN(T_{S',\Sigma',E'})$ tels que :

pour tout t,t_i dans $FN(T_{S,\Sigma,E})$,pour tout $o:s_1..s_n \to s$ dans Σ

$$o(t_1..t_n) \equiv_E t \Rightarrow r(o)(f(t_1)..f(t_n)) \equiv_{E'} f(t)$$

La démonstration est analogue à la précédente compte tenu des remarques faites en début de ce paragraphe.

Proposition 3:

Si $T_{S,\Sigma,E} \overset{\varphi}{\subseteq} T_{S',\Sigma',E'}$ et $T_{S',\Sigma',E'} \overset{\varphi}{\subseteq} T_{S,\Sigma,E}$ alors

$$T_{S,\Sigma,E} \overset{\varphi}{\simeq} T_{S',\Sigma',E'}$$

En effet

$T_{S,\Sigma,E} \overset{\rho}{\simeq} T_{S',\Sigma',E'}$ <=> il existe r de (S,Σ) dans (S',Σ') tel que

$$r(S) \subseteq S'$$

$$r(\Sigma) \subseteq \Sigma'$$

pour tout t,t' dans T_{Σ} $t \equiv_E t'$ <=> $r(t) \equiv_{E'} r(t')$

et

$T_{S',\Sigma',E'} \overset{\rho}{\simeq} T_{S,\Sigma,E}$ <=> il existe r' de (S',Σ') dans (S,Σ) tel que

$$r'(S') \subseteq S$$

$$r'(\Sigma') \subseteq \Sigma$$

pour tout t,t' dans $T_{\Sigma'}$ $t \equiv_{E'} t'$ <=> $r'(t) \equiv_E r'(t')$

ainsi les renommages étant injectifs on a

$$card(S) = card(S') \quad \text{et} \quad card(\Sigma) = card(\Sigma')$$

r et r' sont alors surjectifs et la proposition 1 s'applique

on a donc $T_{S,\Sigma,E} \overset{\rho}{\simeq} T_{S',\Sigma',E'}$

Exemple :

Soient les spécifications suivantes :

```
type nat
  profils
   zeronat          :nat;
   snat (nat)  :nat;
   pnat (nat)  :nat;
   addnat (nat,nat) :nat;
   moinsnat(nat,nat):nat;
   infnat (nat,nat) :booléen;
  var
   n1,n2 :nat;
  axiomes
```



```
   pnat(zeronat) -> zeronat;
   pnat(snat(n1)) -> n1;
   moinsnat(n1,zeronat) -> n1;
   moinsnat(n1,snat(n2)) -> pnat(moinsnat(n1,n2));
   addnat(n1,zeronat) -> n1;
   addnat(n1,snat(n2)) -> snat(addnat(n1,n2));
   infnat(zeronat,snat(n1)) -> vrai;
   infnat(n1,zeronat) -> faux;
   infnat(snat(n1),snat(n2)) -> infnat(n1,n2);
fin
```

```
type relatif
  profils
   zerorel                  :relatif;
   c(booléen,nat)           :relatif;
   addrel (relatif,relatif) :relatif;
   opprel (relatif)         :relatif;
   succrel (relatif)        :relatif;
   predrel (relatif)        :relatif;
  var
   r1,r2:relatif;
   b:booléen;
   n1,n2:nat;
  axiomes
   zerorel -> c(vrai,zeronat);
   c(faux,zeronat) -> c(vrai,zeronat);
   succrel(c(vrai,n1)) -> c(vrai,snat(n1));
```

```
    succrel(c(faux,zeronat)) -> c(vrai,snat(zeronat));
    succrel(c(faux,snat(nl))) -> c(faux,nl);
    predrel(c(faux,nl)) -> c(faux,snat(nl));
    predrel(c(vrai,zeronat)) -> c(faux,snat(zeronat));
    predrel(c(vrai,snat(nl))) -> c(vrai,nl);
    opprel(c(vrai,nl)) -> c(faux,nl);
    opprel(c(faux,nl)) -> c(vrai,nl);
    addrel(c(b,nl),c(b,n2)) -> c(b,addnat(nl,n2));
    addrel(c(vrai,nl),c(faux,n2)) -> si infnat(nl,n2)
                                     alors c(faux,moinsnat(n2,nl))
                                     sinon c(vrai,moinsnat(n2,nl));
    addrel(c(faux,nl),c(vrai,nl)) -> si infnat(nl,n2)
                                     alors c(vrai,moinsnat(n2,nl))
                                     sinon c(faux,moinsnat(nl,n2));
fin
```

```
type integer
  profils
    zeroint  :integer;
    succint (integer)  :integer;
    predint (integer)     :integer;
    oppint (integer)      :integer;
    addint (integer,integer):integer;
  var
    xl,x2:integer;
  axiomes
    succint(predint(xl)) -> xl;
    predint(succint(xl)) -> xl;
    oppint(zeroint) -> zeroint;
    oppint(succint(xl)) -> predint(oppint(xl));
    oppint(predint(xl)) -> succint(oppint(xl));
    addint(zeroint,xl) -> xl;
    addint(succint(xl),x2) -> succint(addint(xl,x2));
    addint(predint(xl),x2) -> predint(addint(xl,x2));
fin
```

Un a "integer" $\overset{e}{\subseteq}$ "relatif" avec le renommage r défini par :

r(integer) =relatif

r(booléen) = booléen

r(zeroint) = zerorel

r(succint) = succrel

r(predint) = predrel

r(oppint) = opprel

r(addint) = addrel

et une fonction entre formes normales définie par :

$$f(zeroint) = c(vrai, zeronat)$$

$$f(succint^n(zeroint)) = c(vrai, snat^n(zeronat))$$

$$f(predint^n(zeroint)) = c(faux, snat^n(zeronat))$$

En effet :

$r(zeroint)=zerorel =_{relatif} c(vrai,zeronat) = f(zeronat)$

$r(succint)(f(succint^n(zeroint))) = succrel(c(vrai,snat^n(zeronat)))$

$$=_{relatif} c(vrai,snat^{n+1}(zeronat))$$

ceci par application de la troisième règle de "relatif"

$$= f(succint^{n+1}(zeroint))$$

par définition de f .

On procédera de même pour chaque opération et chaque forme normale de "integer".

On remarquera que le renommage r' défini par:

$$r'(\text{succint}) = \text{predrel}$$

$$r'(\text{predint}) = \text{succrel}$$

et pour les autres symboles $r'(x) = r(x)$

et la fonction f' définie par :

$$f'(\text{zeroint}) = c(\text{vrai}, \text{zeronat})$$

$$f'(\text{succint}^n(\text{zeroint})) = c(\text{faux}, \text{snat}^n(\text{zeronat}))$$

$$f'(\text{predint}^n(\text{zeroint})) = c(\text{vrai}, \text{snat}^n(\text{zeronat}))$$

conduiraient aux mêmes résultats.

Cet exemple montre de plus que les symboles intervenant dans les formes normales du "plus grand" type ne sont pas nécessairement dans l'image du renommage.

III.4.6. Notion d'environnement.

Dans les définitions et propositions précédentes, nous ne faisons pas explicitement référence à un type d'intérêt dans les spécifications. Ce sont en fait des spécifications d' environnement et le renommage porte sur tous les symboles (de sortes et d'opérations) de l'un de ces environnements vers l'autre. En pratique, il apparait important de ne définir qu'un type à la fois. On aboutit ainsi à la notion de spécification structurée d'un environnement. Dans ce cas la spécification $\langle S,\Sigma,E\rangle$ est munie d'un ordre

noté $<<$ sur S et est telle qu'on puisse partitionner Σ et E en associant à chaque sorte s de S une partition Σ_s et une partition E_s de E de façon à ce que toute sorte s' utilisée dans Σ_s (ou dont les opérations de $\Sigma_{s'}$ sont utilisées dans E_s) vérifie $s'<<s$.

L'environnement spécifié est ainsi obtenu par enrichissements successifs de l'environnement vide par des présentations de types abstraits dans l'ordre $<<$, sans "perturber" les types déjà définis (problème de complétude suffisante) et de façon consistante <BID81>.

Etant donnée une spécification $\langle S,\Sigma,E\rangle$ on pourra en général la structurer en définissant plusieurs ordres vérifiant la contrainte précédente; de ce fait l'ordre n'est pas nécessairement conservé par renommage pour deux spécifications structurées (d'environnements) ρ-équivalentes.

Dans la suite nous nous intéresserons à la comparaison de deux présentations de types abstraits dans un même environnement structuré; de ce fait on pourra "standardiser" les profils des opérations d'une présentation en se fondant sur l'ordre défini sur les sortes de l'environnement. Dans un tel contexte il est intéressant de voir sous quelles conditions deux environnements

$\langle S \cup \{s\}, \Sigma \cup \Sigma_s, E \cup E_s\rangle$ et

$\langle S \cup \{s'\}, \Sigma \cup \Sigma_{s'}, E \cup E_{s'}\rangle$

(avec $\langle s,\Sigma_s,E_s\rangle$ telle que

non (s dans S)

$\Sigma_s \cap \Sigma = \emptyset$

pour tout o dans Σ_s s apparait au moins une fois dans le profil de o

et il existe o dans Σ_s telle que $\text{profil}(o)=s_1..s_n->s$.

De même pour $\langle s', \Sigma_{s'}, E_{s'}\rangle$)

obtenus par enrichissement d'un même environnement $\langle S,\Sigma,E\rangle$ seront ρ-équivalents.

Il serait plus agréable que le renommage défini entre les environnements puisse être obtenu à partir du renommage identité (sur les symboles de l'environnement commun de départ) étendu sur les symboles nouveaux introduits par les présentations.

Par définition de la $\mathcal{C}$-équivalence

$\langle S \cup \{s\}, \Sigma \cup \Sigma_s, E \cup E_s \rangle \overset{\mathcal{C}}{\simeq} \langle S \cup \{s'\}, \Sigma \cup \Sigma_{s'}, E \cup E_{s'} \rangle$

si et seulement si il existe un renommage r surjectif de

$S \cup \{s\} + \Sigma \cup \Sigma_s \rightarrow S \cup \{s'\} + \Sigma \cup \Sigma_{s'}$

tel que

$r(T_{S \cup \{s\}, \Sigma \cup \Sigma_s, E \cup E_s}) \simeq T_{S \cup \{s'\}, \Sigma \cup \Sigma_{s'}, E \cup E_{s'}}$

de par la structuration de l'environnement on a alors

$r(T_{S,\Sigma,E}) \overset{\mathcal{C}}{\simeq} T_{S,\Sigma,E}$

mais ceci n'entraine pas que

pour tout symbole dans S+E r(symbole) = symbole.

Pour pouvoir l'affirmer on devrait avoir

$r(T_{S,\Sigma,E}) \simeq T_{S,\Sigma,E} \Rightarrow r = r_I$

ou r_I note le renommage identité; propriété qu'on pourrait qualifier de non redondance de l'environnement spécifié par $\langle S,\Sigma,E \rangle$.

Compte tenu de la réalisation et de l'état de nos réflexions, nous allons définir une notion de $\mathcal{C}$-équivalence de deux présentations dans un même environnement structuré de la façon suivante :

Définition 9:

Deux présentations T_s : $\langle s,\Sigma_s,E_s \rangle$ et $T_{s'}$: $\langle s',\Sigma_{s'},E_{s'} \rangle$ dans l'environnement $\mathcal{E}$ spécifié par $\langle S,\Sigma,E \rangle$ sont $\mathcal{C}$-équivalentes dans $\mathcal{E}$, ce que l'on note $T_s \overset{\mathcal{C}}{\simeq}_{\mathcal{E}} T_{s'}$, si et seulement si

il existe un renommage surjectif r de $S \cup \{s\} + \Sigma \cup \Sigma_s$ dans

$S \cup \{s'\} + \Sigma \cup \Sigma_{s'}$ tel que

pour tout symbole u dans $S + \Sigma$ $r(u) = u$

et $r(T_{S \cup \{s\}, \Sigma \cup \Sigma_s, E \cup E_s}) \simeq T_{S \cup \{s'\}, \Sigma \cup \Sigma_{s'}, E \cup E_{s'}}$

Cette définition est très liée à la notion d'environnement et appauvrit quelque peu la définition générale en effet

si on note $\mathcal{E} + T_s$ l'environnement $\mathcal{E}$ enrichi par la présentation T_s et $\mathcal{E} + T_{s'}$ l'environnement $\mathcal{E}$ enrichi par la présentation $T_{s'}$

$\mathcal{E} + T_s \overset{\mathcal{C}}{\simeq} \mathcal{E} + T_{s'} \quad \langle\!\not\Rightarrow\!\rangle \quad T_s \overset{\mathcal{C}}{\simeq}_{\mathcal{E}} T_{s'}$

par contre dans le cas d'environnement non redondants

$\mathcal{E} + T_s \overset{\mathcal{C}}{\simeq} \mathcal{E} + T_{s'}$ et $(r(T_{S,\Sigma,E}) \simeq T_{S,\Sigma,E} \Rightarrow r = r_I)$

$\langle\!\Rightarrow\!\rangle \quad T_s \overset{\mathcal{C}}{\simeq}_{\mathcal{E}} T_{s'}$.

Compte tenu de l'objectif poursuivi, c'est à dire particulariser la proposition 2 au cas de la $\mathcal{C}$-équivalence de deux présentations dans un même environnement et obtenir une proposition "locale", nous allons devoir ajouter une contrainte supplémentaire sur les présentations considérées. La proposition 2 porte sur des termes clos, nous désirons maintenant faire intervenir des variables à la place des termes de types "externes" (autres que le type d'intérêt) de façon à ce que l'instanciation des termes que nous devrons effectuer pour construire le renommage puisse être faite uniquement à partir des présentations à "comparer".

Soit $\langle s,\Sigma_s,E_s \rangle$ une présentation dans l'environnement spécifié par $\langle S,\Sigma,E \rangle$ et $\mathcal{V}_e$ un ensemble de variables (typées) de types différents de s.

Soit I_s l'ensemble des opérations de Σ_s dont le codomaine est le type d'intérêt s.

Nous noterons :

* $T_{I_s,\mathcal{V}_e}$ l'ensemble des termes (de type s) librement engendrés par I_s

sur $\mathcal{V}_e$ (conformément aux profils des opérations et aux types des variables)

* $FN_{\mathcal{C}+T}$ (t) la forme irréductible d'un terme t dans le système $E \cup E_s$
* $FN(T_{I_s \cup \mathcal{V}_e}, E \cup E_s) = \{ FN_{\mathcal{C}+T}(t)$ pour tout t dans $T_{I_s,\mathcal{V}_e} \}$
* $\equiv^{ind}_{E \cup E_s}$ l'équivalence <u>inductive</u> définie sur les termes avec variables par les règles $E \cup E_s$. Par définition

pour tout t,t'

$t \equiv^{ind}_{E \cup E_s} t'$ <==> pour toute substitution $\sigma : \mathcal{V}_e \to T_{S,\Sigma,E}$

$\sigma t \equiv^{close}_{E \cup E_s} \sigma t'$

Nous utiliserons les mêmes notations pour $T_{s'}$.

Proposition 4:

Soient T_s : $\langle s,\Sigma_s,E_s\rangle$ et $T_{s'}$: $\langle s',\Sigma_{s'},E_{s'}\rangle$ deux présentations dans l'environnement $\mathcal{E}$ spécifié par $\langle S,\Sigma,E\rangle$, soit $\mathcal{V}_e$ un ensemble de variables de types externes tels que :

1) $E \cup E_s$ et $E \cup E_{s'}$ forment des systèmes de réécriture canoniques

2)

a) $FN(T_{I_s \cup \mathcal{V}_e}, E \cup E_s) \subseteq T_{I_s,\mathcal{V}_e}$

b) pour tout t,t' dans $T_{I_s,\mathcal{V}_e}$

$t \equiv^{ind}_{E \cup E_s} t'$ <==> $FN_{\mathcal{C}+I_s}(t) = FN_{\mathcal{C}+I_s}(t')$

c) pour tout o dans $\Sigma_s - I_s$ de profil $s^i s_{i+1}..s_n \to s_{n+1}$ avec $s_{i+1}..s_{n+1} \neq s$

pour tout $t_1..t_n$ dans $T_{I_s,\mathcal{V}_e}$

pour tout $x_{i+1}..x_n$ dans $\mathcal{V}_e$

si $FN_{\mathcal{C}+I_s}(o(t_1..t_i, x_{i+1}..x_n)) = t$

alors t dans $FN(T_{S,\Sigma,E}) \cup \mathcal{V}_e$.

On fait les mêmes hypothèses sur $T_{s'}$.

T_s et $T_{s'}$ sont $\mathcal{C}$-équivalentes dans $\mathcal{E}$ si et seulement si

il existe un renommage r surjectif de $\{s\} + \Sigma_s$ dans $\{s'\} + \Sigma_{s'}$ et

f de $FN_{I_s,\mathcal{V}_e} + FN(T_{S,\Sigma,E}) + \mathcal{V}_e$ dans $FN_{I_{s'},\mathcal{V}_e} + FN(T_{S,\Sigma,E}) + \mathcal{V}_e$ injective

tels que :

I) $f(x) = x$ pour tout x dans $\mathcal{V}_e \cup FN(T_{S,\Sigma,E})$

II) pour tout o : $s^i s_{i+1}..s_n \to s_{n+1}$ avec $s_{i+1},..,s_n \neq s$

pour tout $x_{i+1}..x_n$ dans $\mathcal{V}_e$

pour tout $t_1..t_i$ dans $FN(T_{I_S} \cup \mathcal{V}_e, E \cup E_S)$

pour tout t dans $FN(T_{I_S} \cup \mathcal{V}_e, E \cup E_S) \cup \mathcal{V}_e \cup FN(T_{S,\Sigma,E})$

$FN_{\mathcal{C}+T_S}(o(t_1..t_i, x_{i+1}..x_n)) = t \Rightarrow$

$FN_{\mathcal{C}+T_{S'}}(r(o)(f(t_1)..f(t_i), f(x_{i+1})..f(x_n))) = f(t)$

Les restrictions du 2) assurent :

* que pour tout o dans Σ_s de profil $s^i s_{i+1}..s_n \rightarrow s_{n+1}$ avec $s_{i+1}..s_n \neq s$

pour tout $t_1..t_i$ dans $T_{I_S}, \mathcal{V}_e$

pour tout $x_{i+1}..x_n$ dans $\mathcal{V}_e$

$FN_{\mathcal{C}+T_S}(o(t_1..t_i, x_{i+1}..x_n))$ est dans $FN(T_{I_S} \cup \mathcal{V}_e, E \cup E) \cup \mathcal{V}_e \cup FN(T_{S,\Sigma,E})$

* et que si on affecte une <u>valeur</u> (terme clos sous forme normale) aux <u>variables</u> figurant dans les termes normalisés on ne pourra pas appliquer de règle supplémentaire.

Tout se passe alors comme si les variables étaient des constantes de types externes.

Une façon d'avoir 2) est de ne pas autoriser dans les règles l'occurrence d'un sous-terme de la forme $o_e(..x..)$ ou o_e est dans Σ et x est une variable.

Preuve

==>)

Montrons que, avec les hypothèses 1) et 2),

$T_S \overset{e}{\underset{\mathcal{C}}{\sim}} T_{S'}$ => il existe un renommage r surjectif et f injective tels que on

ait I), II).

Soient o dans Σ_s,

$t_1,..t_i$ dans $FN(T_{I_S} \cup \mathcal{V}_e, E \cup E_S)$,

$x_{i+1},..x_n$ dans $\mathcal{V}_e$,

t dans $FN(T_{I_S} \cup \mathcal{V}_e, E \cup E_S) \cup FN(T_{S,\Sigma,E}) \cup \mathcal{V}_e$

tels que

a) $FN_{\mathcal{C}+T_S}(o(t_1..t_i, x_{i+1}..x_n)) = t$

En appliquant un renommage r quelconque on a

b) $FN_{r(\mathcal{C}+T_S)}(r(o)(r(t_1)..r(t_i), r(x_{i+1})..r(x_n)) = r(t)$

Par définition des termes renommés on a

$r(x_{i+1}) = x_{i+1}$ .. $r(x_n) = x_n$

d'ou la formule

c) $FN_{r(\mathcal{C}+T_S)}(r(o)(r(t_1)..r(t_i), x_{i+1}..x_n)) = r(t)$

Compte tenu de l'hypothèse 2), c) est équivalente à

c') $r(o)(r(t_1)..r(t_i), x_{i+1}..x_n) \equiv^{ind}_{r(E \cup E_S)} r(t)$

Par hypothèse de $\mathcal{C}$-équivalence de T_s et $T_{s'}$

d) $r(o)(r(t_1)..r(t_i), x_{i+1}..x_n) \equiv^{ind}_{E \cup E_{S'}} r(t)$

Par hypothèse les $r(t_1)..r(t_i)$ sont dans $T_{I_{S'}}, \mathcal{V}_e$ et r(o) est dans $\Sigma_{s'}$

En réduisant et compte tenu des restrictions 2) sur $T_{s'}$ cette fois

e) $FN_{\mathcal{C}+T_{S'}}(r(o)(r(t_1)..r(t_i), x_{i+1}..x_n)) = FN_{\mathcal{C}+T_{S'}}(r(t))$

Par calcul des formes normales

f) $FN_{\mathcal{C}+T_{S'}}(r(o)(FN_{\mathcal{C}+T_{S'}}(r(t_1))..FN_{\mathcal{C}+T_{S'}}(r(t_i)), x_{i+1}..x_n)) = FN_{\mathcal{C}+T_{S'}}(r(t))$

posons

$f(t) = FN_{\mathcal{C}+T_{S'}}(r(t))$

$f(t_1) = FN_{\mathcal{C}+T_{S'}}(r(t_1))$, ,

$f(t_i) = FN_{\mathcal{C}+T_{S'}}(r(t_i))$

en substituant dans f) on obtient

$FN_{\zeta+I_S}(r(o)(f(t_1)..f(t_i),x_{i+1}..x_n)) = f(t)$

qui est le second membre de la formule II), la formule a) en étant le premier membre.

Pour établir I) on a :

* Si $t = x$ avec x dans $\mathcal{V}_e$ alors

$r(x) = x$ par définition d'un terme renommé

$FN_{\zeta+I_S'}(x) = x$ une variable ne se réduit pas

d'ou $f(x) = x$

* Si $t = y$ avec y dans $FN(T_{S,\Sigma,E})$ alors

$r(y) = y$ le renommage ne portant pas sur les symboles de l'environnement

$FN_{\zeta+I_S'}(y) = y$ puisque y est un terme clos sous forme normale

d'ou $f(y) = y$

<==)

Montrons la réciproque c'est à dire que

s'il existe un renommage r surjectif et f injective tels qu'on ait I) et II) (sous l'hypothèse 2))

alors $T_S \underset{\zeta}{\overset{e}{\sim}} T_{S'}$

Pour ce faire nous avons besoin du lemme suivant :

pour tout x dans $T_{I_S,\mathcal{V}_e}$, t dans $FN(T_{I_S} U \mathcal{V}_e, E U E_S)$ on a

$t = FN_{\zeta+I_S}(x) \Rightarrow FN_{\zeta+I_S'}(r(x)) = f(t)$

Démonstration :

par récurrence

a) posons $x = op_0$ avec op_0 dans I_S

soit $t_0 = FN_{\zeta+I_S}(op_0)$

t_0 est dans $FN(T_{I_S} U \mathcal{V}_e, E U E_S)$ par 2)

appliquons II)

$FN_{\zeta+I_S}(op_0) = t_0 \Rightarrow FN_{\zeta+I_S'}(r(op_0)) = f(t_0)$

b) posons $x = op(x_1..x_i, x_{i+1}..x_n)$ avec op dans I_S

avec $x_1..x_i$ dans $T_{I_S,\mathcal{V}_e}$ et

$x_{i+1}..x_n$ dans $\mathcal{V}_e$

x est dans $T_{I_S,\mathcal{V}_e}$ par construction

Soient $t_1 = FN_{\zeta+I_S}(x_1) \ldots t_i = FN_{\zeta+I_S}(x_i)$

$t_1..t_i$ sont dans $FN(T_{I_S} U \mathcal{V}_e, E U E_S)$ par définition de $FN(T_{I_S} U \mathcal{V}_e, E U E_S)$

soit $t = FN_{\zeta+I_S}(op(x_1..x_i, x_{i+1}..x_n))$ avec op dans I_S

t est dans $FN(T_{I_S} U \mathcal{V}_e, E U E_S)$ par définition

on a $FN_{\zeta+I_S}(x) = FN_{\zeta+I_S}(op(x_1..x_i, x_{i+1}..x_n)) = t$

c'est à dire en remplaçant $x_1..x_i$ par $t_1..t_i$

$FN_{\zeta+I_S}(op(t_1..t_i, x_{i+1}..x_n)) = t$

On peut appliquer l'hypothèse II) d'ou

$FN_{\zeta+I_S}(x) = t \Rightarrow FN_{\zeta+I_S'}(r(op)(f(t_1)..f(t_i), f(x_{i+1})..f(x_n))) = f(t)$

Par I) on obtient

$FN_{\zeta+I_S}(x) = t \Rightarrow FN_{\zeta+I_S'}(r(op)(f(t_1)..f(t_i), x_{i+1}..x_n)) = f(t)$

Par hypothèse de récurrence on a

$FN_{\zeta+I_S'}(r(x_1)) = f(t_1)$

.

.

$FN_{\zeta+I_S'}(r(x_i)) = f(t_i)$

d'ou

$FN_{\mathcal{C}+T_S}(x) = t \Rightarrow FN_{\mathcal{C}+T_{S'}}(r(op)(FN_{\mathcal{C}+T_S}(r(x_1))..x_{i+1}..x_n)) = f(t)$

En remplaçant les $FN_{\mathcal{C}+T_{S'}}(r(x_1))$ .. par les $r(x_1)$..

$FN_{\mathcal{C}+T_S}(x) = t \Rightarrow FN_{\mathcal{C}+T_{S'}}(r(op)(r(x_1)..r(x_i),x_{i+1}..x_n)) = f(t)$

Et comme par définition des termes renommés

$r(op)(r(x_1)..r(x_i),x_{i+1}..x_n) = r(op(x_1..x_i,x_{i+1}..x_n))$

on a $FN_{\mathcal{C}+T_S}(x) = t \Rightarrow FN_{\mathcal{C}+T_{S'}}(r(op(x_1..x_n))) = f(t)$

ou encore

$FN_{\mathcal{C}+T_S}(x) = t \Rightarrow FN_{\mathcal{C}+T_{S'}}(r(x)) = f(t)$

ce qui termine la preuve du lemme.

Le lemme étant démontré montrons que

pour tout x,x' dans $T_{S\ \cup\{s\},\Sigma\cup\Sigma_S}$

$x \equiv_{E\cup E_S} x' \iff r(x) \equiv_{E\cup E_{S'}} r(x')$

pour cela nous allons énumérer les différents cas possibles.

a)

Le renommage ne portant pas sur l'environnement on a

pour tout x,x' dans $T_{S,\Sigma,E}$ $r(x) = x$, $r(x') = x'$

d'ou par hypothèse de complétude suffisante et de consistance de T_S et $T_{S'}$

$x \equiv_{E\cup E_S} x' \iff r(x) \equiv_{E\cup E_{S'}} r(x')$

b)

Considérons maintenant les termes de $T_{S\cup\{s\},\Sigma\cup I_S}$

Le lemme qui porte sur des termes avec variables de $T_{I_S,\mathcal{V}_e}$ s'applique sur les termes clos obtenus par substitutions des variables par des termes clos <u>irréductibles</u> car l'hypothèse 2) garantit qu'on ne peut faire d'autre réduction après remplacement. Ainsi si t de $T_{I_S,\mathcal{V}_e}$ contient les variables

$x_1..x_n$ de $\mathcal{V}_e$

et si on note σ la substitution

$(x_1 <- v_1),..,(x_n <- v_n)$ ou les $v_1,..,v_n$ sont dans $FN(T_{S,\Sigma,E})$ on a

$FN_{\mathcal{C}+T_S}(\sigma.t) = \sigma.FN_{\mathcal{C}+T_S}(t)$

De plus l'hypothèse de canonicité du système de réécriture permet d'affirmer que

pour tout $t_1,..,t_n$ de $T_{S,\Sigma}$,

pour tout σ de $\Sigma\cup\Sigma_S$

$FN_{\mathcal{C}+T_S}(\sigma(t_1..t_n)) = FN_{\mathcal{C}+T_S}(\sigma(FN_{\mathcal{C}+T_S}(t_1)..FN_{\mathcal{C}+T_S}(t_n)))$

Compte tenu de ces rappels

soient x,x' dans $T_{S\cup\{s\},\Sigma\cup I_S}$

posons $t = FN_{\mathcal{C}+T_S}(x)$

$t' = FN_{\mathcal{C}+T_S}(x')$

le lemme s'applique, donc on a

$FN_{\mathcal{C}+T_{S'}}(r(x)) = f(t)$

$FN_{\mathcal{C}+T_{S'}}(r(x')) = f(t')$

on a alors

* $x \equiv_{E\cup E_S} x' \Rightarrow t = t' \Rightarrow f(t) = f(t')$
  $\Rightarrow FN_{\mathcal{C}+T_{S'}}(r(x)) = FN_{\mathcal{C}+T_{S'}}(r(x'))$
  $\Rightarrow r(x) \equiv_{E\cup E_{S'}} r(x')$
* $r(x) \equiv_{E\cup E_{S'}} r(x') \Rightarrow FN_{\mathcal{C}+T_{S'}}(r(x)) = FN_{\mathcal{C}+T_{S'}}(r(x'))$
  $\Rightarrow f(t) = f(t')$
  $\Rightarrow t = t'$ par injectivité de f
  $\Rightarrow FN_{\mathcal{C}+T_S}(x) = FN_{\mathcal{C}+T_S}(x')$
  $\Rightarrow x \equiv_{E\cup E_S} x'$

on donc

pour tout x,x' dans $T_{S\cup\{s\},\Sigma\cup I_S}$

$x \equiv_{E\cup E_S} x' \iff r(x) \equiv_{E\cup E_{S'}} r(x')$

c)

considérons maintenant les termes de la forme

$$o(x_1..x_n)$$

avec o dans $\Sigma_s - I_s$

et $x_1..x_n$ dans $FN(T_S \cup \{s\}, \Sigma \cup I_S, E \cup E_S)$

la propriété de complétude suffisante permet d'affirmer que

$FN_{\mathcal{C}+T_S}(o(x_1..x_n))$ est dans $FN(T_S,\Sigma,E)$

posons

$t = FN_{\mathcal{C}+T_S}(o(x_1..x_n))$

$t' = FN_{\mathcal{C}+T_S}(o'(x'_1..x'_n))$

pour les mêmes raisons qu'en b) l'hypothèse II) s'applique à ces termes clos on a alors

$FN_{\mathcal{C}+T_S}(o(x_1..x_n)) = t \Rightarrow FN_{\mathcal{C}+T_S'}(r(o)(f(x_1)..f(x_n))) = f(t)$

et

$FN_{\mathcal{C}+T_S}(o'(x'_1..x'_n)) = t' \Rightarrow FN_{\mathcal{C}+T_S'}(r(o')(f(x'_1)..f(x'_n))) = f(t')$

de ce fait

$o(x_1..x_n) \equiv_{E \cup E_S} o'(x'_1..x'_n) \quad <==>$

$t = t' \quad <==> \quad f(t) = f(t')$ puisque par l'hypothèse I) dans ce cas

$f(t) = t$ et $f(t') = t'$

qui est équivalent à

$FN_{\mathcal{C}+T_S'}(r(o)(f(x_1)..f(x_n))) = FN_{\mathcal{C}+T_S'}(r(o')(f(x'_1)..f(x'_n)))$

ou en terme de congruence

$r(o)(f(x_1..f(x_n)) \equiv_{E \cup E_S'} r(o')(f(x'_1)..f(x'_n))$

parmi les $x_1..x_n,x'_1..x'_n$ certains sont dans $FN(T_S,\Sigma,E)$ pour ceux là

$x_i = r(x_i) = f(x_i)$

pour les autres on a

$x_i = FN_{\mathcal{C}+T_S'}(x_i)$ car ils sont sous forme normale

et donc

$f(x_i) = FN_{\mathcal{C}+T_S'}(r(x_i))$

formule équivalente à

$f(x_i) \equiv_{E \cup E_S'} r(x_i)$

le remplacement des f(..) par les r(..) est alors fondé et on obtient

$r(o)(r(x_1)..r(x_n)) \equiv_{E \cup E_S'} r(o')(r(x'_1)..r(x'_n))$

ou par définition du renommage

$r(o(x_1..x_n)) \equiv_{E \cup E_S'} r(o'(x'_1..x'_n))$

on a donc établi

$o(x_1..x_n) \equiv_{E \cup E_S} o'(x'_1..x'_n)$

<==>

$r(o(x_1..x_n)) \equiv_{E \cup E_S'} r(o'(x'_1..x'_n))$

pour tout o,o' dans $\Sigma_s - I_s$ et

$x_1..x_n,x'_1..x'_n$ dans $FN(T_S \cup \{s\}, \Sigma \cup I_S, E \cup E_S)$

Compte tenu de b) cette équivalence reste vraie en prenant les

$x_1..x_n,x'_1..x'_n$ dans $T_S \cup \{s\}, \Sigma \cup I_S, E \cup E_S$

On a ainsi énuméré tous les cas.

III.4.7. Remarques.

La $\mathcal{C}$-équivalence de deux présentations standardisées dans un même environnement couvre une bonne partie des exemples présentés en III.2. Elle ne prend pas en compte les permutations des rôles de deux ou plusieurs arguments de même type dans le profil d'une opération; ainsi si dans une présentation "int1" des entiers

moins1(x,y) vaut x - y

et dans une présentation "int2"

moins2(x,y) vaut y - x

toutes chose égales par ailleurs, les deux présentations ne sont pas ε-équivalentes, et on ne met pas en évidence qu'au lieu d'utiliser "intl" et moinsl(a,b) on peut utiliser "int2" et moins2(b,a).

De la même façon les présentations de types abstraits paramétrés n'entrent pas dans le cadre de la ε-équivalence telle qu'elle est définie. Nous ne tenterons pas d'étendre les définitions et propositions précédentes; il est toutefois clair que le renommage devra porter sur les paramètres (types et opérations sur ces types) et que les "contraintes" <EHR81> sur ces paramètres devront être équivalentes après renommage.

Par ailleurs ainsi qu'il a été noté , le renommage n'associe pas nécessairement les constructeurs (symboles figurant dans les formes normales , et opérations par la composition desquelles on peut "atteindre" tous les objets du type abstrait) deux à deux .

De ce fait ,et compte tenu de l'objectif que nous cherchons à atteindre , nous ne les avons pas distingués syntaxiquement dans la présentation des spécifications .

III.5. COMMENT PROPOSER UN RENOMMAGE ?

Deux problèmes se posent si on veut utiliser les propositions précédentes pour comparer deux spécifications :

- exhiber une application de renommage et une application entre formes normales
- prouver les implications induites .

Les critères syntaxiques dont nous disposons sont :

- l'isomorphie des profils
- l'association par le renommage du type d'intérêt de l'une au type d'intérêt de l'autre.

Ceci n'est pas suffisant pour éviter une explosion combinatoire des choix possibles, on devra donc s'attacher à proposer une méthode permettant d'y remédier . Ce problème est particulièrement crucial dans le contexte d'utilisation retenu , celui d'une "base" ou bibliothèque de spécifications ou l'on désire accéder par les "règles" .

Nous allons proposer , à travers des exemples , deux approches pour le faire .

Dans l'une on tentera d'effectuer partiellement les preuves nécessaires garantissant la ε-équivalence , alors que dans l'autre on ne fera pas de preuves mais on essaiera d'obtenir un renommage qui a "toutes les chances" de convenir.

III.5.1. Approche permettant une preuve partielle.

Considérons les spécifications suivantes inspirées de <BID81> :

```
type intl
  profils
   zero1          :intl;
   succl (intl)   :intl;
   predl (intl)   :intl;
   oppl (intl)    :intl;
  var x:intl;
  axiomes
   succl(predl(x)) -> x;
   predl(succl(x)) -> x;
   oppl(zero1) -> zero1;
   oppl(succl(x)) -> predl(oppl(x));
   oppl(predl(x)) -> succl(oppl(x));
fin
```

```
type int2
  profils
   zero2            :int2;
   succ2 (int2)     :int2;
   opp2  (int2)     :int2;
   pred2 (int2)     :int2;
  var x:int2;
  axiomes
   succ2(opp2(succ2(x))) -> opp2(x);
```



```
   opp2(zero2) -> zero2;
   opp2(opp2(x)) -> x;
   pred2(zero2) -> opp2(succ2(zero2));
   pred2(succ2(x)) -> x;
   pred2(opp2(x)) -> opp2(succ2(x));
fin
```

Pour montrer que ces deux spécifications sont $\mathcal{C}$-équivalentes nous allons être amené à utiliser leurs formes normales.

Pour "intl" on définit l'ensemble FN_{intl} des formes normales par:

$$FN_{intl} = \{ zero1 \} \cup \{ FN_{succl} \} \cup \{ FN_{predl} \}$$

avec

$$FN_{succl} = \{ succl(zero1) \} \cup \{ succl(x) \text{ avec } x \text{ dans } FN_{succl} \}$$

$$FN_{predl} = \{ predl(zero1) \} \cup \{ predl(x) \text{ avec } x \text{ dans } FN_{predl} \}$$

et pour "int2" , FN_{int2} par:

$$FN_{int2} = \{ zero2 \} \cup \{ FN_{succ2} \} \cup \{ FN_{opp2} \}$$

$$FN_{succ2} = \{ succ2(zero2) \} \cup \{ succ2(x) \text{ avec } x \text{ dans } FN_{succ2} \}$$

$$FN_{opp2} = \{ opp2(x) \text{ avec } x \text{ dans } FN_{succ2} \}$$

Pour prouver la $\mathcal{C}$-équivalence on doit trouver r et f satisfaisant :

pour tout o dans $\{ zero1, succl, predl, oppl \}$

pour tout $t_1..t_n, t$ dans FN_{intl}

$$o(t_1..t_n) \equiv_{intl} t \Rightarrow r(o)(f(t_1)..f(t_n)) \equiv_{int2} f(t)$$

avec $f(t_1)..f(t_n), f(t)$ dans FN_{int2} .

Comme on se place dans le cadre de systèmes de réécriture canoniques ceci est équivalent à:

(I) $FN_{intl}(o(t_1..t_n)) = t \Rightarrow FN_{int2}(r(o)(f(t_1)..f(t_n))) = f(t)$

Pour déterminer r et f on précise la formule (I) pour chaque opération et

chaque type de forme normale de "intl" (il y en a cinq). On obtient ainsi les seize formules suivantes, qui se structurent ainsi :

* la formule 1) est construite pour o=zerol
* les formules 2) à 16) sont construites pour o dans succl,predl,oppl
* les formules 2) à 4) sont construites pour t_1 = zerol
* les formules 5) à 7) sont construites pour t_1 = succl(zerol)
* les formules 8) à 10) sont construites pour t_1 = predl(zerol)
* les formules 11) à 13) sont construites pour t_1 = succl(x) avec x dans FN_{succl}
* les formules 14) à 16) sont construites pour t_1 = predl(x) avec x dans FN_{predl}

1) FN_{intl}(zerol) = zerol => FN_{int2}(r(zerol)) = f(zerol)

2) FN_{intl} (succl(zerol)) = succl(zerol)
=> FN_{int2} (r(succl)(f(zerol)) = f(succl(zerol))
3) FN_{intl} (predl(zerol)) = predl(zerol)
=> FN_{int2} (r(predl)(f(zerol)) = f(predl(zerol))
4) FN_{intl} (oppl(zerol)) = zerol => FN_{int2} (r(oppl)(f(zerol)) = f(zerol)

5) FN_{intl} (succl(succl(zerol))) = succl(succl(zerol))
=> FN_{int2} (r(succl)(f(succl(zerol)))) = f(succl(succl(zerol)))
6) FN_{intl} (predl(succl(zerol))) = zerol
=> FN_{int2} (r(predl)(f(succl(zerol))) = f(zerol)
7) FN_{intl} (oppl(succl(zerol))) = predl(zerol)
=> FN_{int2} (r(oppl)(f(succl(zerol)))) = f(predl(zerol))

8) FN_{intl} (succl(predl(zerol))) = zerol

=> FN_{int2} (r(succl)(f(predl(zerol)))) = f(zerol)
9) FN_{intl} (predl(predl(zerol))) = predl(predl(zerol))
=> FN_{int2} (r(predl)(f(predl(zerol)))) = f(predl(predl(zerol)))
10) FN_{intl} (oppl(predl(zerol))) = succl(zerol)
=> FN_{int2} (r(oppl)(f(predl(zerol)))) = f(succl(zerol))

11) FN_{intl} (succl(succl(x))) = succl(succl(x))
=> FN_{int2} (r(succl)(f(succl(x)))) = f(succl(succl(x)))
Ouvrons une parenthèse :
par hypothèse x est dans FN_{succl} de ce fait succl(x) et succl(succl(x)) le sont aussi et l'égalité de gauche est valide .
12) FN_{intl} (predl(succl(x))) = x => FN_{int2} (r(predl)(f(succl(x))) =f(x)
13) FN_{intl} (oppl(succl(x))) = FN_{intl} (predl(oppl(x)))
=> FN_{int2} (r(oppl)(f(succl(x)))) = f(FN_{intl} (predl(oppl(x))))

14) FN_{intl} (succl(predl(x))) = x => FN_{int2} (r(succl)(f(predl(x)))) = f(x)
15) FN_{intl} (predl(predl(x))) = predl(predl(x))
=> FN_{int2} (r(predl)(f(predl(x)))) = f(predl(predl(x)))
16) FN_{intl} (oppl(predl(x))) = FN_{intl} (succl(oppl(x)))
=> FN_{int2} (r(oppl)(f(predl(x)))) = f(FN_{intl} (succl(oppl(x))))

Les formules ainsi obtenues doivent être satisfaites par r et f , par construction ,et sans oublier les prédicats d'appartenance attachés aux variables , les égalités des membres de droite sont toutes valides , il suffit donc de valider les égalités des membres de gauche.
Comme nous l'avons vu en III.4.4. , si "intl" et "int2" sont $\mathcal{C}$-équivalentes on a :

D1) pour tout t dans FN_{intl} $f(t) = FN_{int2}(r(t))$

On va maintenant essayer de construire r et f progressivement en procédant par remplacement des termes r(...) et f(...) puis en calculant les $FN_{int2}(..)$ ainsi obtenus .

En l'absence d'autre choix possible posons:

r(zero1) = zero2

Remplaçons toutes les occurrences de r(zero1) par zero2 dans les formules 1) à 16)

1') ainsi obtenue est satisfaite par :

f(zero1) = zero2

conformément à D1) .

Effectuons les substitutions correspondantes :

2),3),4),6),8) sont modifiées , choisissons 4') ou deux substitutions ont été effectuées

4') FN_{intl} (oppl(zero1)) = zero1 => FN_{int2} (r(oppl)(zero2)) = zero2

on a trois choix possibles :

r(oppl) = succ2

r(oppl) = pred2

r(oppl) = opp2

Si l'on prend r(oppl) = succ2

on obtient FN_{int2} (succ2(zero2)) = succ2(zero2)

d'ou succ2(zero2) = zero2 qui est faux

de la même façon FN_{int2} (pred2(zero2)) = opp2(succ2(zero2))

conduirait à opp2(succ2(zero2)) = zero2

par contre FN_{int2} (opp2(zero2)) = zero2 est valide posons donc :

r(oppl) = opp2

et effectuons les substitutions correspondantes .

Considérons la formule

2') FN_{intl} (succl(zero1))=succl(zero1)

=> FN_{int2} (r(succl)(zero2))=f(succl(zero1))

les deux choix possibles sont :

r(succl) = pred2

r(succl) = succ2

posons arbitrairement :

r(succl) = pred2

on devra revenir ultérieurement sur ce choix.

afin d'avoir FN_{int2} (pred2(zero2)) = opp2(succ2(zero2))

et FN_{int2} (pred2(zero2)) = f(succ1(zero1))

on pose :

f(succ1(zero1)) = opp2(succ2(zero2))

après les substitutions correspondantes 5),6),7),11),14) sont modifiées

5')FN_{int1} (succ1(succ1(zero1))) = succ1(succ1(zero1))

=> FN_{int2} (pred2(opp2(succ2(zero2)))) = f(succ1(succ1(zero1)))

est satisfaite car :

FN_{int2} (pred2(opp2(succ2(zero2)))) = opp2(succ2(succ2(zero2)))

et

f(succ1(succ1(zero1))) = FN_{int2} (pred2(pred2(zero2))) par D1)

et

FN_{int2} (pred2(pred2(zero2))) = opp2(succ2(succ2(zero2)))

3') FN_{int1} (pred1(zero1)) = pred1(zero1)

=> FN_{int2} (r(pred1)(zero2)) = f(pred1(zero1))

est satisfaite en posant :

r(pred1) = succ2 f(pred1(zero1)) = succ2(zero2)

6') FN_{int1} (pred1(succ1(zero1))) = zero1

=> FN_{int2} (succ2(opp2(succ2(zero2)))) = zero2

est satisfaite par application de la règle 1. de "int2" . On vérifie de même 7'),8'),10') .

9') FN_{int1} (pred1(pred1(zero1))) = pred1(pred1(zero1))

=> FN_{int2} (succ2(succ2(zero2))) = f(pred1(pred1(zero1))

est valide conformément à D1) .

Les formules contenant des variables x de FN_{int1} seront traitées de la même manière en appliquant systématiquement D1) et les règles de "int2" . On valide ainsi simplement les formules 11') , 14') , 15') , 16') .

Pour 12') et 13') on ne peut le faire en utilisant les seuls mécanismes de remplacement , application de D1) , normalisation .

12') FN_{int1} (pred1(succ1(x))) = x => FN_{int2} (succ2(f(succ1(x)))) = f(x)

en appliquant D1) et comme pour tout terme y $\equiv_{int2}$ FN_{int2} (y)

on doit vérifier :

FN_{int2} (succ2(pred2(r(x)))) = FN_{int2} (r(x))

aucune règle ne s'appliquant ,on considérera cette égalité comme un théorème à démontrer :

TH1 : pour tout x dans FN_{int1} FN_{int2} (succ2(pred2(r(x)))) = FN_{int2} (r(x))

13') FN_{int1} (opp1(succ1(x))) = FN_{int1} (pred1(opp1(x)))

=> FN_{int2} (opp2(f(succ1(x)))) = f(FN_{int1} (pred1(opp1(x))))

en appliquant D1)

on obtient l'égalité suivante à vérifier :

FN_{int2} (opp2(pred2(r(x)))) = FN_{int2} (r(FN_{int1} (pred1(opp1(x)))))

comme FN_{int1} (pred1(opp1(x))) $\equiv_{int1}$ pred1(opp1(x)) par définition de FN_{int1}

on obtient :

FN_{int2} (opp2(pred2(r(x)))) = FN_{int2} (succ2(opp2(r(x))))

aucune règle ne s'appliquant , on posera :

TH2 : pour tout x dans FN_{intl}

FN_{int2} (opp2(pred2(r(x)))) = FN_{int2} (succ2(opp2(r(x))))

On obtient finalement le résultat suivant :

Sous réserve de la preuve des théorèmes TH1 et TH2 les types "intl" et "int2" sont $\mathcal{C}$-équivalents et les applications r et f vérifient la proposition 1 avec :

r(intl) = int2

r(zerol) = zero2

r(succl) = succ2

r(oppl) = opp2

et

$$f(x) = FN_{int2}(r(x))$$

On remarquera que la preuve de ces théorèmes nécessite un raisonnement par induction , mais que toute les formes normales de FN_{intl} ne sont pas à considérer ,dans la mesure ou les variables x utilisées dans les formules 1) à 16) vérifiaient un prédicat d'appartenance à un sous ensemble de FN_{intl} . De plus ,dans cet exemple une seule règle était à chaque fois applicable pour la réécriture d'un terme contenant une variable , ce n'est pas le cas général , on devra alors satisfaire une des égalités engendrées , sous

l'hypothèse de confluence du système de réécriture .

La simplicité des mécanismes mis en oeuvre laisse envisager une possibilité d'automatisation de cette méthode pour une classe raisonnable de spécifications ; sans en donner l'algorithme , on peut toutefois présenter quelques remarques :

- il nécessite la donnée des formes normales et la donnée des prédicats permettant de décider si un terme est ou non sous forme normale ,
- il fournit comme résultat un (ou plusieurs si l'on développe tous les choix possibles) renommage(s) et une liste de théorèmes (éventuellement vide) sous l'hypothèse desquels la $\mathcal{C}$-équivalence est établie .

De ce fait il peut proposer un renommage pour des types qui ne sont pas $\mathcal{C}$-équivalents ,ainsi pour une "pile" {vide,empiler,depiler} et une "file" {filevide,ajouter,enlever} on obtiendra la proposition de renommage suivante:

r(pile) = file

r(vide) = filevide

r(empiler) = ajouter

r(depiler) = enlever

avec le théorème à "démontrer" :

$$FN_{file}(enlever(ajouter(r(x)))) = FN_{file}(r(x))$$

cette formule ne pouvant être ni satisfaite ni réfutée sans induction .

De ce fait , si un tel algorithme présente des avantages en cas de $\mathcal{C}$-équivalence par les preuves qu'il effectue en même temps qu'il construit le renommage ,il introduirait si on l'utilisait dans le contexte d'une recherche en bibliothèque , beaucoup trop de "bruit" dans les réponses fournies ,à moins bien entendu d'être capable d'automatiser les preuves par

induction.

III.5.2. Approximation de la $\mathcal{E}$-équivalence par comparaison d'exemples.

On peut faire deux remarques sur le développement de l'exemple précédent:

- les hypothèses de définition du renommage r ont été fondées sur l'examen d'égalités entre termes clos
- les égalités entre termes avec variables ont servi à valider partiellement les implications découlant de la proposition 1.

L'"euréka" conduisant à la proposition d'un renommage procède d'un mécanisme d'essais/erreur sur des exemples "bien choisis" , puis d'une généralisation des résultats obtenus. La démarche est d'une nature quasi-naturelle ou expérimentale suivant le schéma:

1. Examen d'un ensemble fini de cas
2. Emission d'hypothèse par généralisation
3. Essai de preuve .

Compte tenu du contexte dans lequel nous nous situons (bibliothèque de spécifications) nous estimons qu'il est souhaitable de "sophistiquer" le mécanisme d'émissions des hypothèses de façon à ce que les propositions faites soient "pertinentes" et qu'il y ait ainsi moins de tentatives de preuve (à priori couteuses) inutiles.

L'idée simple qui vient alors à l'esprit consiste à traiter un plus grand nombre d'exemples (égalités entre termes clos) pour établir les hypothèses .

Intuitivement ces ensembles d'égalités ne devront pas être quelconques mais construits progressivement à partir des valeurs "initiales" (constantes)

des types considérés en appliquant systématiquement toutes les opérations de la signature .Ceci revient à développer les premiers pas d'un raisonnement par récurrence .

Reprenons les spécifications "intl" et "int2" précédentes et construisons pour chacune d'elle un ensemble d'égalités entre termes clos .Nous associerons de plus un nom à chaque terme . On obtient pour "intl" :

1) FN_{intl} (zerol) = $tintl_1$
   $tintl_1$ = zerol
2) FN_{intl} (succl($tintl_1$)) = $tintl_2$
   $tintl_2$ = succl(zerol)
3) FN_{intl} (predl($tintl_1$)) = $tintl_3$
   tint3 = predl(zerol)
4) FN_{intl} (oppl($tintl_1$)) = $tintl_1$
5) FN_{intl} (succl($tintl_2$)) = $tintl_4$
   $tintl_4$ = succl(succl(zerol))
6) FN_{intl} (predl($tintl_2$)) = $tintl_1$
7) FN_{intl} (oppl($tintl_2$)) = $tintl_3$
8) FN_{intl} (succl($tintl_3$)) = $tintl_1$
9) FN_{intl} (predl($tintl_3$)) = $tintl_5$
   $tintl_5$ = predl(predl(zerol))
10) FN_{intl} (oppl($tintl_3$)) = $tintl_2$

.
.
.

et pour "int2" :

1) FN_{int2} (zero2) = $tint2_1$
   $tint2_1$ = zero2
2) FN_{int2} (pred2($tint2_1$)) = $tint2_2$

$tint2_2$ = opp2(succ2(zero2))

3) FN_{int2} (opp2($tint2_1$)) = $tint2_3$

4) FN_{int2} (succ2($tint2_1$)) = $tint2_3$

$tint2_3$ = succ2(zero2)

5) FN_{int2} (pred2($tint2_2$)) = $tint2_4$

$tint2_4$ = opp2(succ2(succ2(zero2)))

6) FN_{int2} (opp2($tint2_2$)) = $tint2_3$

7) FN_{int2} (succ2($tint2_2$)) = $tint2_1$

8) FN_{int2} (pred2($tint2_3$)) = $tint2_1$

9) FN_{int2} (opp2($tint2_3$)) = $tint2_2$

10) FN_{int2} (succ2($tint2_3$)) = $tint2_5$

$tint2_5$ = succ2(succ2(zero2))

.

.

.

Une représentation graphique ou les noeuds représentent les termes sous forme normale t_i et les arcs $t \xrightarrow{\sigma} t'$ les égalités FN(σ(t)) = t' permet de visualiser les résultats obtenus par la figure 1 .

On voit ainsi apparaitre les deux renommages possibles en "superposant" deux à deux les noeuds et les arcs étiquettés de façon que si un arc $\xrightarrow{\sigma_{ij1}}$ se superpose à un arc $\xrightarrow{\sigma_{ij2}}$ alors tous les arcs $\xrightarrow{\sigma_{ij1}}$ se superposent deux à deux aux arcs $\xrightarrow{\sigma_{ij2}}$.

Considérons les présentations suivantes :

type pile

profils

vide :pile;

empiler(pile,entier):pile;

depiler(pile) :pile;

var

p:pile;

e:entier;

axiomes

depiler(vide) -> erreur ;

depiler (empiler(p,e)) -> p;

fin

type file

profils

filevide :file;

ajouter (file,entier) :file;

enlever (file) :file ;

var

f :file;

el,e2 :entier;

axiomes

enlever(filevide) -> erreur ;

enlever(ajouter(filevide,el)) -> filevide ;

enlever(ajouter(ajouter(f,el),e2)) ->ajouter(enlever(ajouter(p,el)),e2);

fin

Le processus de construction est légérement différent de l'exemple précédent dans la mesure ou l'on doit définir des arguments de type "entier" pour obtenir des égalités; conformément à la proposition 4 , considérons il,i2,...,jl,j2,... comme des variables universellement quantifiées,on

posera par ailleurs

$tpile_0 = erreur_{pile} = \underline{erreur}$

et

$tfile_0 = erreur_{file} = \underline{erreur}$

ou erreur est une abréviation désignant , pour chaque type , la classe d'équivalence des termes-erreurs du type . On obtient alors la représentation graphique de la figure 2. Cette fois les deux graphes ne se superposent plus .

D'une façon générale l'intervention de types externes dans les profils conduira à utiliser des variables dans les égalités entre termes , lors d'une tentative de définition d'un renommage on devra les "superposer" deux à deux de la même manière que les termes "internes" et les opérations , cette "superposition" revient à renommer les symboles de variables , opération licite dans la mesure ou les variables sont implicitement universellement quantifiées . Ces superpositions de termes traduisent la fonction f des propositions 1 et 2 .

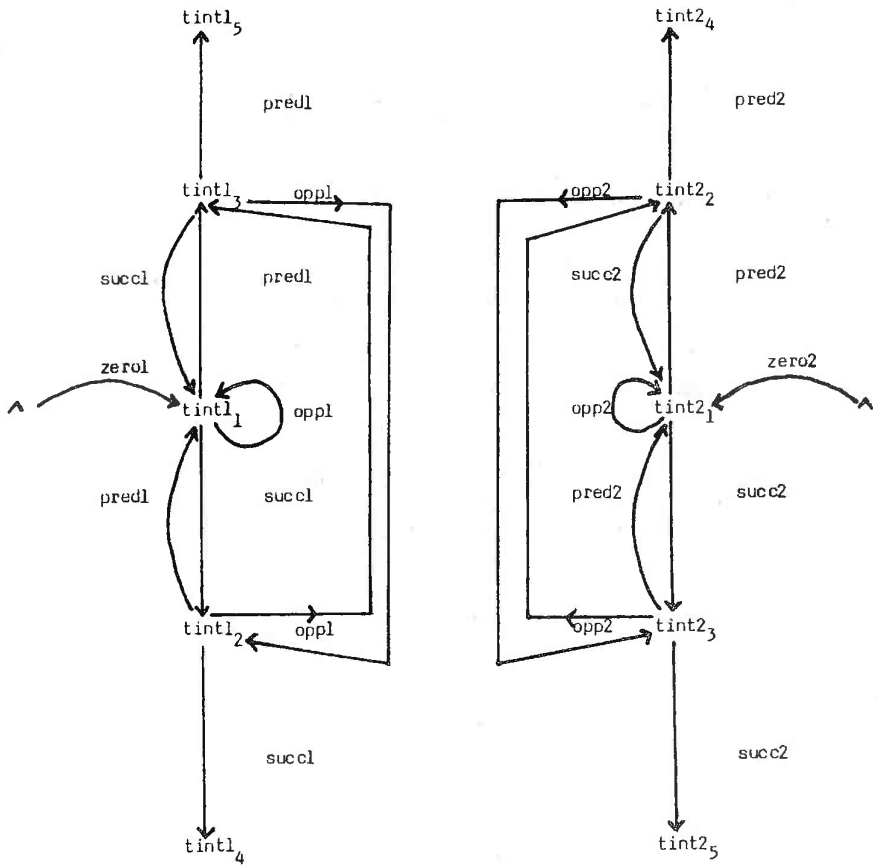


Figure 1.

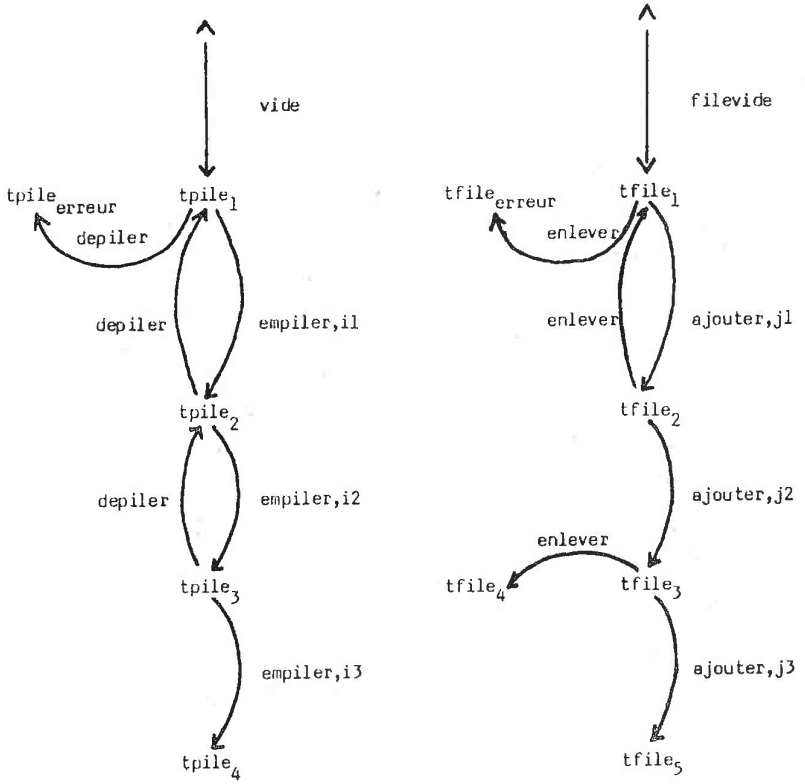


Figure 2.

III.6. PRESENTATION DE LA METHODE RETENUE

La méthode d'approximation de la ζ-équivalence (et de construction du renommage) par comparaison d'exemples évoquée dans les lignes précédentes est celle que nous avons retenue . Nous allons la présenter plus en détail . Les exemples traités mettent en évidence deux étapes , d'une part la construction de termes et d'égalités entre ces termes , d'autre part la construction du renommage . On pourrait construire les termes en même temps que l'on cherche un renommage , il est toutefois plus intéressant de procéder en deux temps , la construction de l' ensemble d'égalités (le jeu d'exemples ou "support") sera faite une fois pour toute , et le support ainsi constitué sera attaché à la spécification .

La représentation graphique utilisée dans le paragraphe précédent nous a conduit dans un premier temps à aborder la construction du renommage en terme de cheminement dans les graphes , la version "zero" du prototype ainsi obtenue n'ayant pas donné entière satisfaction et présentant des difficultés à évoluer , nous avons cherché une autre façon de résoudre le problème , que nous exposerons dans ce qui suit . Nous ferons toutefois référence à la représentation graphique pour illustrer notre propos.

III.6.1. Jeu d'exemples : principes de construction et représentation.

Si on représente l'ensemble d'égalités entre termes sous forme de graphe comme précédemment ,et compte tenu du fait que l'on va chercher à faire se "superposer" deux graphes , on se rend compte que les résultats seront obtenus plus simplement et de façon plus "sûre" si les graphes sont connexes (fortement) .En d'autres termes tout "point" t_i doit être

successeur d'un point t_j ou encore :

pour tout t_i il existe t_j et o dans Σ tels que la formule

$t_i = FN(o(t_j...))$

est dans l'ensemble de formules . Une façon d'obtenir ce résultat consiste à construire une suite d'égalités en appliquant à chaque terme t_i , figurant dans les égalités précédentes , les opérations de la signature et en prenant la forme normale du terme ainsi constitué ; la suite sera initialisée à partir des opérations dont le codomaine est le type d'intérêt et n'ayant pas le type d'intérêt dans leur domaine .

La définition informelle précédente ne précise pas comment on doit procéder lorsque des types externes figurent dans les profils des opérations , on doit alors instancier des arguments aux emplacements correspondants . Prenons l'exemple de la pile , ayant t_1 = vide et FN(vide) = t_1 on doit calculer FN(empiler(t_1,?) ,

on est donc amené à fixer le deuxième argument ,de type "entier" , représenté ici par un point d'interrogation . Deux choix sont possibles :

soit on pose l'égalité t_2 = FN(empiler(t_1,1)) avec t_2 = empiler(vide,1))

c'est à dire que l'on choisit une <u>valeur</u> arbitraire pour les arguments de type externe et on obtient bien ainsi des termes clos ,

soit on pose :

pour tout x_1 t_2 = FN(empiler(t_1,x_1)) avec t_2 = empiler(vide,x_1)

c'est à dire que l'on utilise comme arguments externes des variables quantifiées universellement . On n'a plus dans ce cas de termes clos , on se situe alors dans le cadre de la proposition 4 ; les symboles de variables ne peuvent apparaitre qu'aux places des arguments externes et on devra s'assurer que le remplacement de ces variables par des termes irréductibles (et clos) des types correspondants ne permet pas d'effectuer d'autre réduction . Tout se passe alors comme si ces variables nommaient des

constantes externes quelconques . Leur quantification universelle est alors justifiée et les formules obtenues sont plus générales que celles construites sur des termes clos .

Poursuivons l'exemple :

ayant t_1 = vide , t_2 = empiler(vide,x_1) ,

t_1 = FN(vide)

t_2 = FN(empiler(t_1,x_1))

on va devoir calculer un terme de la forme FN(empiler(t_2,x))

si on choisit pour x la variable x_1 précédente on va obtenir :

t_1 = vide , t_2 = empiler(vide,x_1) , t_3 = empiler(empiler(vide,x_1),x_1)

FN(vide) = t_1

FN(empiler(t_1,x_1)) = t_2

FN(empiler(t_2,x_1)) = t_3

si on choisit pour x une autre variable notée x_2 on obtient :

t_1 = vide , t_2 = empiler(vide,x_1) , t_3 = empiler(empiler(vide,x_1),x_2)

FN(vide) = t_1

FN(empiler(t_1,x_1)) = t_2

FN(empiler(t_2,x_2)) = t_3

Ce deuxième choix fournit un ensemble d'équations plus générales , le second ensemble d'équations impliquant le premier , mais l'inverse étant faux . On aura donc intérêt à construire un ensemble de formules

$$FN(o(t_i...)) = t_j$$

tel qu'en remplaçant tous les symboles t par leur définition (termes sous forme normale) les égalités engendrées contiennent au plus une occurrence de chaque variable dans chaque membre . On devra cependant s'efforcer de minimiser le nombre de symboles de variables figurant dans l'ensemble afin de faciliter la tâche ultérieure .

Cette façon de procéder conduit à construire un ensemble de termes par

"couches" successives , chaque couche T_i étant obtenue à partir de la précédente en appliquant à tout terme de T_{i-1} toutes les opérations de la signature (pour lesquelles le type d'intérêt ne figure qu'une fois dans le domaine) et en donnant un nouveau nom aux termes dont la forme normale ne figure pas dans une couche inférieure T_j ($j<i$) . Le nombre de couches que nous appellerons profondeur sera fixé arbitrairement . La première couche t_1 sera obtenue à partir des opérations pour lesquelles le type d'intérêt ne figure pas dans le domaine . On obtient parallèlement un ensemble d'égalités de la forme $t = FN(o(t'...))$ correspondant aux arcs de la représentation graphique . Ayant construit $T = \bigcup_{i=1}^{\text{profondeur}} T_i$ on pourra calculer les formes normales de termes dominés par les opérations pour lesquelles le type d'intérêt apparait plusieurs fois dans le domaine pour toutes combinaisons des t d'un sous-ensemble de T comme arguments . Nous choisirons arbitrairement les deux premières couches de T pour former ce sous ensemble .

Les présentations que nous traiterons seront faites dans un environnement noté $\mathcal{C}$ et spécifié par $<S, \Sigma, E>$; lors de la construction d'un renommage les symboles de $\mathcal{C}$ seront invariants par le renommage .

Appelons arité_en_s(o) le nombre d'occurences du type s dans le domaine de l'opération o du type d'intérêt présenté par $<s, \Sigma_s, E_s>$. Il est clair que la façon de construire le support que nous avons décrite suppose l'existence d' au moins une opération de codomaine s et d'arité_en_s nulle et d' au moins une opération de codomaine s et d'arité_en_s égale à 1 . Plus précisemment un tel ensemble d'opérations doit constituer une "famille génératrice" du type , c'est à dire que si on note Σ_g ce sous-ensemble on doit avoir :

pour tout t dans $T_{\{s\} \cup S, \Sigma \cup \Sigma_s}$

il existe t' dans $T_{\{s\} \cup S, \Sigma \cup \Sigma_g}$ tel que

$$t' =_{E \cup E_s} t$$

Intuitivement cela signifie qu'on peut atteindre tous les objets du type (classes d'équivalence) en n'utilisant que les opérations de la famille génératrice Σ_g (et les opérations de l'environnement) .

N.B : Les constructeurs (opérations intervenant dans les formes normales) forment une famille génératrice , mais une opération d'une famille génératrice n'est pas nécessairement un constructeur pour une présentation donnée .

Le processus de construction du support , si on se représente celui ci comme un graphe , consiste à définir les points origines du graphe à partir des opérations d'arité_en_s nulle , puis de construire simultanément les points (termes) et les arcs (égalités) en utilisant les opérations d'arité_en_s égale à 1 , en s'"éloignant" de plus en plus des points origines .

Cette première limite due au mode de construction choisi n'est pas la seule , elle s'ajoute au fait que les termes obtenus doivent être irréductibles pour toute affectation des variables par un terme irréductible (contrainte de la proposition 4) .

Nous avons choisi de représenter les égalités sous forme de prédicats. De plus ,comme l'arité des opérations fournit un critère de choix lors de la construction du renommage , nous l'avons fait intervenir dans la représentation .

Si arité(o) = n nous représenterons l'égalité

$$y_{n+1} = FN(o(y_1..y_n))$$

par

$$FNn\ o\ y_1..y_n\ y_{n+1}$$

ou les y_i sont soit des symboles t indicés (désignant des termes du type d'intérêt) , soit des symboles x indicés (désignant des variables de types externes) .

la forme

$$FNarité(o)\ o\ y_j..y_{j+n}\ t_i$$

ou o est dans Σ_i et les y sont des symboles x indicés ou t indicés .

L'opérateur "+" désignera , dans l'algorithme , l'adjonction d'une formule en queue de liste et "Λ" la liste vide.

- Les T_i et T sont des ensembles de symboles t indicés
- indice_x (indice_t) désigne le prochain indice à affecter lors de la définition d'un symbole x (respectivement t)
- terme est une suite de symboles représentant un terme
- [début..fin] est l'intervalle de variation des indices associés aux symboles x .

Algorithme

%initialisations %

$T_1 \leftarrow \emptyset$;

$VAL[t_0] \leftarrow$ "erreur" ;

%on rappelle que le symbole "erreur" désigne pour tout type la forme normale de tout terme-erreur %

EG $\leftarrow \Lambda$;

indice_x $\leftarrow$ 1 ;

indice_t $\leftarrow$ 1 ;

% construction de la 1ère "couche" T_1 de termes , et des formules correspondantes %

pour tout o dans Σ_0 faire

début $\leftarrow$ indice_x ; fin $\leftarrow$ indice_x +arité(o) - 1 ;

terme $\leftarrow$ "$o(x_{début}..x_{fin})$" ;

si il existe t_i dans $T_1 \cup \{t_0\}$ tel que calculFN(terme) = $VAL[t_i]$

alors EG $\leftarrow$ EG + "$FNarité(o)\ o\ x_{début}..x_{fin}\ t_i$" ;

sinon $T_1 \leftarrow T_1 \cup \{t_{indice\_t}\}$;

VAL[t_{indice_t}] <- calculFN(terme) ;

EG <- EG +

"FNarité(o) o $x_{début}..x_{fin}$ t_{indice_t}" ;

indice_t <- indice_t + 1 ;

fsi

fpour

indice_x <- sup (arité(o) pour tout o dans Σ_0) ;

% construction des "couches" $T_2..T_{profondeur}$ %

pour i de 2 à profondeur faire

T_i <- $\emptyset$;

pour tout o dans Σ_1 faire

début <- indice_x ; fin <- indice_x +arité(o) -2 ;

pour tout t dans T_{i-1} tels que type(VAL[t]) = s faire

terme <- calculFN("o(VAL t ,$x_{début},..,x_{fin}$)") ;

cas il existe k tel que terme = "x_k"

alors EG <- EG +

"FNarité(o) o t $x_{début}..x_{fin}$ x_k" ;

il existe t_j dans $U_{k=1}^{i} T_k \cup \{t_0\}$ tel que

terme = VAL[t_j]

alors EG <- EG +

"FNarité(o) o t $x_{début}..x_{fin}$ t_j" ;

autre T_i <- $T_i \cup \{t_{indice\_t}\}$;

VAL[t_{indice_t}] <- terme ;

EG <- EG +

"FNarité(o) o t $x_{début}..x_{fin}$ t_{indice_t}" ;

indice_t <- indice_t + 1 ;

fcas

fpour

fpour

indice_x <- indice_x + sup (arité(o) pour tout o dans Σ_1) - 1 ;

fpour

T <- $\emptyset$;

pour tout o dans $\Sigma - (\Sigma_0 \cup \Sigma_1)$ faire

% il existe par hypothèse Σ_p tel que o dans Σ_p %

début <- indice_x ; fin <- indice_x + arité(o) - p +1 ;

pour tout $t_1..t_p$ dans $(T_1 \cup T_2)^p$

tels que type(VAL[t_1]) = s ,.., type(VAL[t_p]) = s faire

terme <- calculFN("o(VAL[t_1],..,VAL[t_p],$x_{début}..x_{fin}$)") ;

cas il existe k tel que terme = "x_k"

alors EG <- EG +

"FNarité(o) o $t_1..t_p x_{début}..x_{fin}$ x_k" ;

il existe t_j dans $U_{k=1}^{profondeur} T_k \cup T \cup \{t_0\}$ tel que

terme = VAL[t_j]

alors EG <- EG +

"FNarité(o) o $t_1..t_p$ $x_{début}..x_{fin}$ t_j" ;

autre T <- T U $\{t_{indice\_t}\}$;

VAL[t_{indice_t}] <- terme ;

EG <- EG + "FNarité(o) o $t_1..t_p$ $x_{début}..x_{fin}$ t_{indice_t}" ;

indice_t <- indice_t + 1 ;

fcas

fpour

fpour

fin

Exemple :

avec la spécification "listel" de III.3.1. , on obtient la liste EG suivante

pour une profondeur 3 :

FN0 nill t_1

FN1 makel x_1 t_2

FN2 consl t_1 x_2 t_3

FN2 consl t_2 x_2 t_3

FN1 dell t_1 t_0

FN1 dell t_2 t_1

FN2 consl t_3 x_3 t_5

FN2 consl t_4 x_3 t_6

FN1 dell t_3 t_1

FN1 dell t_4 t_2

FN2 appendl t_1 t_1 t_1

FN2 appendl t_1 t_2 t_2

.

.

FN2 appendl t_1 t_6 t_6

FN2 appendl t_2 t_1 t_2

FN2 appendl t_2 t_2 t_7

.

.

.

nous postulons que les ensembles de formules ainsi obtenus sont une bonne représentation de l'algèbre initiale d'une spécification . La proposition 4 , compte tenu du mode de construction exhaustif de l'algorithme 1 ,permet d'affirmer que si deux présentations T_s et $T_{s'}$ sont $\mathcal{C}$-équivalentes il existe

un renommage r et une fonction f des t_i ,x_i de EG_T dans les t_i et les x_i de $EG_{T'}$ telle que pour toute formule

$$FNn\ o\ v_1..v_n\ v_{n+1}$$

dans EG_T

la formule

$$FNn\ r(o)\ f(v_1)..f(v_n)\ f(v_{n+1})$$

soit dans $EG_{T'}$.

De plus ,pour une profondeur infinie , la réciproque est vraie .On va se servir de cette constatation pour émettre des <u>hypothèses</u> pour r et f .

L'algorithme 1 nécessite de calculer les formes normales de tout terme . Pour le faire on adoptera une stratégie de réduction par "valeur" (ou au plus profond d'abord) en normalisant d'abord les sous-termes de tout terme , et ceci par filtrage avec les membres gauches des règles de la spécification (et de l'environnement) et remplacement par les membres droits correspondant (après avoir effectué les substitutions de variables nécessaires) .

Les termes contenant des symboles de variables x_i , on ne pourra pas toujours calculer leur forme normale ,comme c'est le cas pour la spécification suivante :

<u>type</u> ensemble

<u>profils</u>

vide : ensemble ;

ajouter(ensemble,entier): ensemble ;

enlever(ensemble,entier): ensemble ;

<u>var</u>

e : ensemble ;

i1,i2 : entier ;

<u>axiomes</u>

enlever(vide,il) -> vide ;

enlever(ajouter(e,il),i2) -> si egal(il,i2) alors enlever(e,i2)

sinon ajouter(enlever(e,i2),il) ;

fin

en effet on va obtenir :

t_1 = FN(vide)

t_2 = FN(ajouter(t_1,x_1))

t_1 = FN(enlever(t_1,x_1))

et on va devoir calculer :

FN(enlever(ajouter(vide,x_1),x_2))

= FN(si egal(x_1,x_2) alors vide sinon ajouter(enlever(vide,x_2),x_1)))

= FN(si egal(x_1,x_2) alors vide sinon ajouter(vide,x_1))

qu'on ne peut plus réduire . On trouvera ce cas de figure à chaque fois que lors du déroulement de l'algorithme 1 on sera amené à calculer la forme normale d'un terme dont un sous terme est de la forme :

$$g(x_i..x_{i+n})$$

ou g n'est pas dans la signature et les $x_i..x_{i+n}$ sont des variables quantifiées universellement .

III.6.3. Algorithme de construction du renommage.

Ayant deux listes de formules EG_T et $EG_{T'}$ construites par l'algorithme 1 , on va s'en servir pour proposer un (éventuellement plusieurs) renommage(s) . Comme nous l'avons remarqué précédemment , on doit faire en sorte qu'à toute formule

$$FNn\ o\ y_1..y_n\ y_{n+1}$$

dans EG_T corresponde une formule

$$FNn\ r(o)\ f(y_1)..f(y_n)\ f(y_{n+1})$$

dans $EG_{T'}$.

Compte tenu de l'ordre de construction des formules , on va construire r et f de façon progressive .

Supposons qu'à un instant donné on ait :

$f(t_1) = t'_1$,....,$f(t_n) = t'_n$, f(t) et r(o) n'étant pas définis ,

ayant $FNn\ o\ t_1..t_k\ x_1..x_m\ t_{n+1}$

on va chercher dans $EG_{T'}$ les formules :

$$FNn\ o\ f(t_1)..f(t_k)\ x'_1..x'_m\ t'_{n+1}$$

telles qu'on n'ait pas déjà $t' = f(t_x)$ ou $o' = r(o_x)$

pour un x quelconque ;

on posera alors $t' = f(t)$, $f(x_1) = x'_1$,..,$f(x_n) = x'_n$ et $r(o) = o'$ puis on passera à la formule suivante de EG_T .

Si plusieurs choix sont possibles on devra tous les essayer successivement.

Si on ne trouve pas de telle formule , on aboutit à un échec , on devra revenir en arrière pour essayer d'éventuels choix momentanément écartés .

Initialement on posera $f(t_0) = t'_0$ pour associer deux à deux les termes $erreur_T$ et $erreur_{T'}$.

Si on représente les égalités sous forme de graphe comme dans les figures 1 et 2 précédentes , la superposition de deux graphes va se faire en superposant deux à deux les points origines , les points t_0 et t'_0 puis en passant aux successeurs en faisant en sorte que l'association des "étiquettes" deux à deux soit cohérente .

Compte tenu de la représentation choisie , étant donné deux ensembles de formules EG_T et $EG_{T'}$, nous construirons les divers renommages r et les diverses fonctions f possibles en filtrant les formules de EG_T avec les formules de $EG_{T'}$. Pour cela les symboles o_i , t_i ,x_i figurant dans EG_T

seront considérés comme des variables qu'on substituera par les symboles o_j , t_j , x_j de EG_T, de façon à obtenir

$$EG_T = EG_{T'}$$

une fois les substitutions effectuées .

les symboles FNn seront considérés comme des opérateurs d'arité n + 2 .

Algorithme 2

Définitions

- EG_T est la liste de formules obtenues par l'algorithme 1 pour la spécification T_s : <s,Σ,E> dans laquelle on a préfixé tous les symboles o_i , t_j , x_k par "v$"
- $EG_{T'}$ est la liste de formules obtenues par l'algorithme 1 pour la spécification $T_{s'}$: <s',Σ',E'>
- "renommages" est une liste de "resultats" , la liste vide est notée "Λ"
- chaque "resultat" est une liste de couples ($v\$o_i$,$o_j$) avec o_i dans Σ et o_j dans Σ' qui définit un renommage r de T dans T' par

$$r(o_i) = o_j \quad <=> \quad (v\$o_i, o_j) \text{ dans "resultat"}$$

- "hypotheses" , "hypotheses-suivantes" et "substitutions" sont des listes de couple de la forme ($v\$o_i$,$o_j$) ou ($v\$x_i$,$x_j$) ou ($v\$t_i$,$t_j$) , l'opérateur "+" note la concaténation de deux listes .
- "liste-de-formules" et "liste-suivante" sont deux listes de même type que EG_T
- premier(l) et reste(l) sont deux fonctions calculant respectivement la tête de la liste l , et la liste obtenue en supprimant sa tête
- substituer(liste,substitutions) est une fonction délivrant la liste obtenue en remplaçant dans "liste" toute occurrence des symboles x par y si et seulement si le couple (x,y) est dans "substitutions"
- filtrer(f1,f2) est une fonction dont le résultat est un couple

(succes,substitutions) ou "succes" est un booléen , vrai si le filtrage de f1 avec f2 a réussi ; la liste de couples (x,y) de "substitutions" définit alors une substitution σ des variables de f1 (les $v\$o_i$, $v\$t_i$,$v\$x_i$) par les constantes (les o_j , t_j , x_j) de f2 telle que :

$$\sigma.f1 = f2$$

- construire-renommage(liste-de-formules,hypotheses) est une procédure récursive définie par :

```
debut
si liste-de-formules = Λ
        alors resultat<- tous les (x,y) de hypotheses
                         tels que y dans  ' ;
              renommages <- ajouter(renommages,resultat) ;
sinon   formule <- premier(liste-de-formules) ;
        pour toute formule' dans EG_T, faire
                         (succes,substitutions)   <-
filtrer(formule,formule') ;
                         injective <- pourtout (x,y)
dans substitutions non il existe (z,y) dans hypotheses ;
                         si succes et injective
                         alors
                         liste-suivante           <-
substituer(reste(liste-de-formules),substitutions) ;
                         hypotheses-suivantes     <-
hypotheses + substitutions ;
                         construire-renommage
(liste-suivante,hypotheses-suivantes) ;
                         fsi
        fpour
```

fsi

fin

Algorithme

renommages <- Λ ;

hypotheses <- (v\$$t_0$,$t_0$) ; % t_0 désigne tout terme erreur dans toute spécification %

EG_T <- substituer(EG_T,hypotheses) ;

construire-renommage(EG_T,hypotheses) ;

si renommages = Λ alors "T et T' ne sont pas ρ-équivalentes "

sinon "renommages est une liste de renommages "

"tels que si card(Σ) = card(Σ') "

" alors l'hypothèse T $\overset{\rho}{\underset{\sigma}{\sim}}$ T' peut être "

" raisonnablement émise . "

fin

Lors du déroulement de l'algorithme 2 on essaie pour chaque formule de EG_T tous les choix compatibles avec la liste des substitutions déjà effectuées . Parmi ceux ci , la plupart conduiront à des échecs , mais on ne s'en apercevra qu'au cours des appels récursifs ultérieurs de la procédure "construire-renommage" lorsque , pour une formule donnée de EG_T , aucune formule de $EG_{T'}$ ne conviendra . Compte tenu du mode de construction par "couches" des listes de formules , dès que l'on a renommé toutes les opérations ce problème de choix ne se posent plus , les appels récursifs de "construire-renommage" servent alors à valider ou à invalider l'hypothèse retenue .

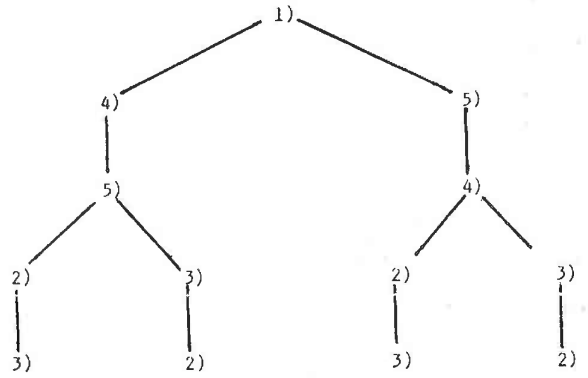
III.6.4. Problèmes de stratégie : ordonnancement du support.

Un pourrait croire que pour diminuer le nombre d'appels engendrés par un choix lors de la construction du renommage , il suffit de diminuer ce nombre "localement" en ne parcourant pas la liste EG_T dans l'ordre de sa construction , mais en choisissant comme prochaine formule à filtrer celle contenant le moins de "variables" . C'est déjà ce qui se passe dans la plupart des cas et ce critère est insuffisant , car en procédant ainsi on aggrandit "moins vite" l'hypothèse de renommage.

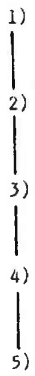
Considérons l'exemple de la figure 3.

On filtre l'ensemble de formules représenté par le graphe de gauche avec l'ensemble de formules représenté par le graphe de droite; la notation g/d indique que le symbole g a été substitué par le symbole d. Dans l'exemple v\$$tT_j$ et v\$$tT_i$ ont été substitués respectivement par tT'_j et tT'_i, et on cherche la prochaine formule (le prochain arc) à filtrer.

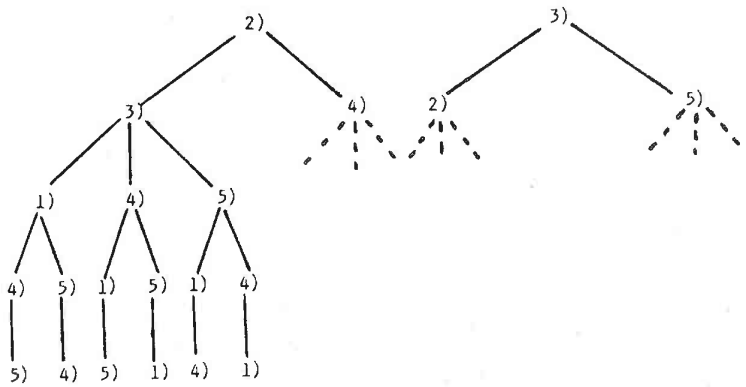
Il apparait que l'on a intérêt à traiter les formules en suivant l'ordre défini par une des branches de l'arbre



plutot que l'ordre initial de la liste



ou que l'ordre obtenu en prenant après chaque substitution la formule contenant le moins de variables



Le premier ordre est tel qu'à chaque appel de "construire-renommage" une seule formule de EG_T, convient , pour les autres , deux choix sont possibles pour les formules 2) ou 3) .

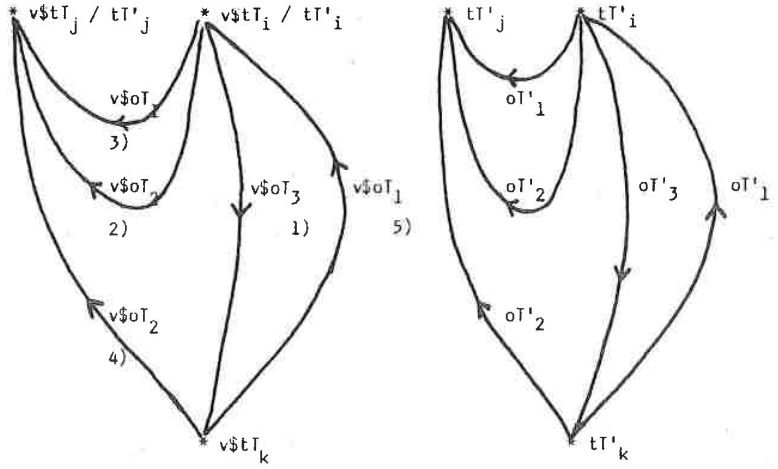


Figure 3.

Ce qui provoque l'explosion du nombre des appels est la présence dans la liste EG_T des deux formules 2) et 3) pouvant être filtrées avec les mêmes formules de EG_T, .Ce sera le cas lorsqu'on obtiendra à l'issue de l'algorithme 1 , une liste dans laquelle plusieurs formules ne diffèreront que par leur premier argument (nom d'opération) , on a alors intérêt à les filtrer le plus tard possible , les possibilités de renommage des symboles diminuant au fur et à mesure du déroulement de l'algorithme 2. On doit cependant conserver dans la mesure du possible l'ordre entre les "couches" qui fait que tout t_i est "défini" avant d'être "utilisé" , ainsi dans la plupart des cas lorsqu'on filtre une formule ,seul le dernier argument reste éventuellement à substituer .

Soit une formule $f = FNnos_1..s_n s_{n+1}$

soit rang(f) le rang de cette formule dans la liste EG_T construite par

l'algorithme 1 , et nombre_occurrences(f) le nombre de formules de EG_T ayant s_{n+1} comme dernier argument . On triera la liste EG_T par nombre_occurrences croissant et rang croissant avant d'appliquer l'algorithme 2 . Ceci permettra de diminuer le nombre d'appels à la procédure "construire-renommage" ; l'algorithme 1 devra être modifié afin d'associer à chaque formule qu'il construit les valeurs nécessaires au tri .

III.6.5. Conclusion.

La méthode présentée ne permet pas de traiter n'importe qu'elle spécification ; d'une part la spécification doit être canonique , d'autre part on doit être capable de décider si un terme avec variables est irréductible pour tout remplacement des variables par des termes (de types externes) irréductibles . Ne voulant pas obliger le spécifieur à définir pour chaque type un prédicat de forme normale , et n'étant pas capable de le construire automatiquement , nous avons utilisé un critère plus large : dès que lors du déroulement de l'algorithme 1 pour une présentation $\langle s,\Sigma_s,E_s\rangle$ on doit réduire un terme dont un sous terme est de la forme $f(x_1..x_n)$ ou f n'est pas dans Σ_s et ou les x_i sont des variables universellement quantifiées , la liste de formules obtenues n'est pas "sûre" .On aurait pu faire le choix de rejeter ,lors de la construction des supports , les spécifications présentant cette caractéristique . Il nous semble plus interessant de les conserver , tout en avertissant l'utilisateur du problème posé .L'algorithme 1 devra alors être modifié de façon à rejeter ces termes . Pour un nombre n de termes construits , le support sera d'autant meilleur que le rapport nombre de termes rejetés / n sera petit . Par ailleurs le principe de notre approche suppose que l'on construise un

nombre non négligeable de termes et de formules entre eux ; étant donné l'aspect combinatoire des algorithmes , on devra faire un compromis entre cette exigence et les nécessités d'économie en temps . En pratique , une profondeur de l'ordre de 3 ou 4 donne de bons résultats pour des spécifications contenant un nombre raisonnable d'opérations (inférieure à la dizaine) . Cette contrainte obligera à structurer d'autant les spécifications et de ce fait nous semble acceptable . On peut de plus étendre l'algorithme 2 de façon à prendre en compte les ϱ-extensions .En effet si $T \overset{\varrho}{\subset} T'$ alors $\text{card}(\Sigma) < \text{card}(\Sigma')$ .Il suffit donc de le dérouler en prenant comme support "cible" celui correspondant à la spécification contenant le plus d'opérations .

Examinons les réponses apportées en fonction des exemples présentés en III.3 :

- Les cas de ϱ-équivalence et de ϱ-extension (III.3.1. ,III.3.2.) sont ceux que nous venons d'examiner .

- Pour ce qui est du problème des permutations sur les profils (III.3.3.), nous avons supposé dans les algorithmes 1 et 2 que les profils des opérations étaient tels que les diverses occurrences du type d'intérêt figurent en tête des profils .Cette contrainte peut être garantie par la définition d'un ordre entre les noms de types .Toute spécification soumise au système ne pourra utiliser un type que si celui ci est présent dans la bibliothèque , on définit ainsi un préordre sur la "compilation" des spécifications , et on peut ordonner les noms de types suivant l'ordre dans lequel ils ont été introduits dans le système . Cette façon de procéder, simple à mettre en pratique, n'est pas la seule possible; des mécanismes plus agréables pour l'utilisateur pourront être retenus pourvu qu'il soit possible d'ordonner les noms des types. Une spécification pour laquelle les profils ne sont pas conformes à cet ordre pourra être transformée

automatiquement lors de sa soumission afin de satisfaire cette exigence .

Supposons une opération o de profil

$s_1^1 s_2^4 s_3^3 s_4^4 \rightarrow s_5^i$

ou les exposants correspondent aux numéros d'ordre des types dans l'environnement , une façon standard de répondre au problème est d'exiger que les types apparaissent dans l'ordre inverse de l'ordre défini sur les noms .

Une présentation standard de la spécification serait telle que

profil(o) = $s_2^4 s_4^4 s_3^3 s_1^1 \rightarrow s_5^i$

pour l'obtenir à partir de la précédente , il suffit de substituer dans la spécification toutes les occurrences d'un terme

$$o(t_1, t_2, t_3, t_4)$$

ou les t_i sont des symboles (de types , de variables) ou des termes par

$$o(t_2, t_4, t_3, t_1)$$

A l'issue de ce chapitre nous disposons donc de définitions pour prendre en compte les relations qui nous semblent importantes entre types abstraits , ainsi que des propositions permettant de les exploiter , et des algorithmes fournissant une aide à la consultation d'une bibliothèque de spécifications .

Nous avons peu insisté sur la conduite des preuves de ϱ-équivalence et de ϱ-extension , mais nous avons préféré chercher un moyen (automatisable) de proposer un renommage. Deux voies s'offraient pour atteindre cet objectif , celle que nous avons retenue peut se mettre en oeuvre de façon simple mais fournit des résultats dont le degré de confiance n'est pas indépendant des spécifications , cet inconvénient nous semble cependant faible au regard de l'aide fournie .

IV. PRESENTATION DU SYSTEME ORSEC

En nous appuyant sur les propositions du chapitre III , nous avons réalisé un prototype de bibliothécaire , baptisé ORSEC (Outil de Recherche de Spécifications Equivalentes par Comparaisons d'exemples). Il ne s'agit pas d'un système complet d'aide au développement de spécifications, on ne pourra donc pas spécifier une application en vue de l'"exécuter", mais uniquement conserver des spécifications de types abstraits et y accéder d'une part par leurs noms, d'autre part ,et c'est la principale fonction du système, par les "règles" en fournissant une spécification pour laquelle on désire savoir s'il en existe une dans la bibliothèque qui lui est ℓ-équivalente. Les spécifications fournies au système sont supposées former un système de réécriture canonique.

Nous présenterons dans ce chapitre les fonctions de l'outil , ainsi que quelques notes sur son implantation. Sa mise en oeuvre sur IRIS 80 , la syntaxe du langage de spécification utilisé et des exemples des résultats qu'il fournit sont donnés en annexe.

IV.1. FONCTIONS DU SYSTEME

ORSEC est un outil conversationnel ; lors de son lancement il engage le dialogue en demandant le nom de l'environnement de l'utilisateur, si celui ci n'existe pas il le crée. A partir de ce moment , toute spécification soumise au système pourra être conservée.

Lorsque le système est en attente de commande , il y a impression du message :

QUE VOULEZ VOUS FAIRE ?

l'utilisateur dispose alors de sept commandes permettant la consultation et la modification de la bibliothèque, le changement du support de sortie des résultats, l'arrêt du système. Une réponse ne correspondant à aucune des commandes existantes provoquera l'impression du "menu" et le retour à l'état "attente de commande".

IV.1.1. Commandes de consultation de la bibliothèque.

Trois commandes permettent la consultation de la bibliothèque en utilisant des accès par noms :

CATALOG : permet l'édition du contenu de la bibliothèque.

Chaque identificateur, prédéfini ou non, est imprimé précédé de 'TYPE' ou 'FONCTION' selon les cas.

Chaque type est suivi de la liste des types qu'il utilise entre les symboles "<" et ">'.

Chaque fonction est suivie de son profil , avec la même syntaxe que dans le langage de spécification.

TYPE : permet l'édition du texte d'un type abstrait (à l'exception des types prédéfinis "BOOLEEN" et "ENTIER"). Le nom du type à éditer est obtenu par dialogue; s'il n'y a pas de type de ce nom dans l'environnement il y impression du message :

TYPE <nom> PREDEFINI OU INEXISTANT

RELATION : permet l'édition de tous les types utilisés par ou utilisant un type donné (fermeture transitive).

Un dialogue conduit l'utilisateur à fournir le nom du type et à préciser s'il désire la fermeture transitive de la relation "UTILISE" (frappe du caractère "U") ou de son inverse (frappe du caractère "I").

IV.1.2. Commandes de consultation et de modification.

La commande principale du système ,"COMPARER" , permet à la fois la consultation de l'environnement en exhibant des spécifications supposées être ϱ-équivalentes, ϱ-inférieures ou ϱ-supérieures à une spécification donnée, et sa modification.

Par ailleurs une commande permet la suppression d'une spécification de l'environnement.

COMPARER : le lancement de cette commande provoque l'impression du message :

NOM DU FICHIER SOURCE ?

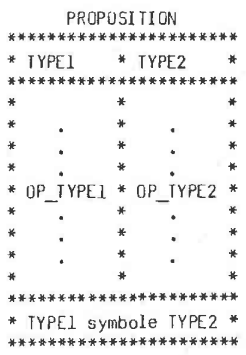
l'utilisateur doit fournir le nom d'un fichier (format éditeur de textes) contenant le texte source d'un type abstrait. Celui ci est alors analysé syntaxiquement; à l'issue de cette étape un message avertit l'utilisateur de la présence ou non d'erreurs

syntaxiques. En cas d'erreur , le système passe dans l'état "attente de commande", l'utilisateur doit sortir du système pour effectuer les corrections nécessaires du texte source.

Si le texte est correct , il y a construction du "support", puis comparaison de celui ci avec ceux associés aux spécifications présentes dans l'environnement. A l'issue de chaque comparaison s'affiche le message :

DUREE DE LA COMPARAISON n SECONDES

et si celle ci est positive un tableau de la forme suivante est imprimé :



ou "TYPE1", "TYPE2" sont des identificateurs de types (parmi lesquels figure celui du type source), "OP_TYPE1", "OP_TYPE2" des identificateurs de fonctions; ce tableau définit un renommage. Dans la dernière ligne de ce tableau, "symbole" est soit "<" qui indique que "TYPE2" est une ε-extension de "TYPE1", soit "<==>" qui indique que les deux types sont ε-équivalents.

Ces résultats sont des hypothèses .

A l'issue des différentes comparaisons l'utilisateur a la

possibilité d'inclure la spécification source dans la bibliothèque en répondant à la question :

FAUT-IL CONSERVER LE TYPE (OUI/NON) ?

DETRUIRE : permet la suppression d'une spécification de la bibliothèque. Le nom du type est obtenu par dialogue. Un type ne peut être supprimé que s'il n'est pas utilisé par d'autres spécifications, et s'il ne s'agit pas d'un type prédéfini.

Selon les cas on obtiendra l'impression de l'un des messages suivants :

TYPE :<nom> INEXISTANT OU PREDEFINI

TYPE UTILISE DANS D'AUTRES SPECIFICATIONS

TYPE DETRUIT

IV.1.3. Commandes annexes.

STOP : arrêt de la session.

TERMINAL : permet de changer le support de sortie des résultats.

Implicitement, les résultats des commandes CATALOG, TYPE, RELATION, COMPARER sont imprimés sur le terminal de l'utilisateur; cette commande permettra de conserver ces résultats, qui peuvent être longs, sur un fichier. Lors de son lancement, la commande imprime le texte suivant :

DONNEZ LE NOM DU FICHIER DE SORTIE DES RESULTATS

OU FRAPPEZ "TTY" SI VOUS LES VOULEZ A LA CONSOLE ?

IV.2. IMPLANTATION DU SYSTEME

L'objectif de cette partie n'est pas de fournir une documentation complète sur la réalisation du produit, mais de donner une idée de son architecture et de préciser les différences entre le produit et la présentation faite en III.6.

IV.2.1. Remarques générales.

La démarche qui a conduit à la réalisation d'ORSEC n'a pas été linéaire, en ce sens que le modèle du prototype n'a pas été figé avant que ne débute sa réalisation. De ce fait la version actuelle ne correspond pas exactement aux développements présentés en III.6.:

* il n'y a pas de "standardisation" des profils des opérations (cf III.6.3.)
* les arguments de types externes ne figurent pas dans la représentation des égalités sous la forme

FN n $s_1 s_2 \ldots s_n s$

mais bien évidemment figurent dans les termes développés.

Ainsi la représentation de la file de la figure 2 sera :

FN0 filevide $tfile_1$

FN1 enlever $tfile_2$ $tfile_{erreur}$

FN1 ajouter $tfile_1$ $tfile_2$

.

.

.

au lieu de :

FN0 filevide $tfile_1$

FN1 enlever $tfile_1$ $tfile_{erreur}$

FN2 ajouter $tfile_1$ j_1 $tfile_2$

.

.

.

Ces deux restrictions se neutralisent en quelque sorte l'une l'autre. La conséquence en est cependant un nombre plus important de choix possibles lors de la construction du renommage, en contrepartie des "ressemblances" plus larges seront détectées.

* Les formules constituant le support ne sont pas triées. Cette lacune conduit à des temps de réponses très longs dans certains cas, ainsi l'exemple des "listes pointées" et des "chenilles" de <CLR81> n'a pu être traité qu'après avoir effectué un tri manuel d'un des deux supports.

Nous terminerons en précisant que l'environnement est initialisé avec les types "ENTIER" et "BOOLEEN" et que l'évaluation des expressions de type "ENTIER" et "BOOLEEN" construites avec les opérateurs prédéfinis ("+","-","*","DIV","MOD","ET","OU","NON") est faite directement, sans utiliser de règles de réécriture. Il en est de même pour les conditionnel-

les, dans lesquelles on n'évalue les sous termes des parties "ALORS" et "SINON" que si c'est nécessaire.

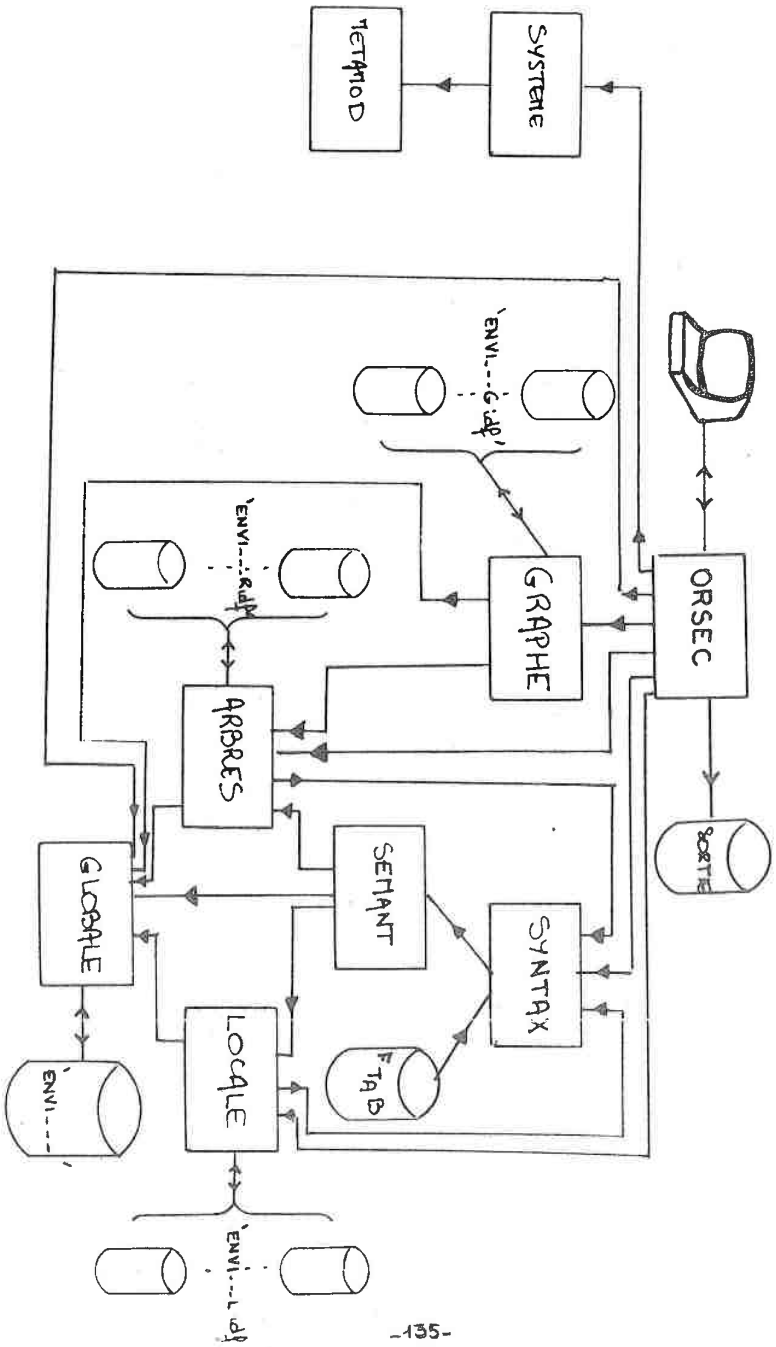
A l'exception de ce dernier point, nous n'avons pas cherché à optimiser les algorithmes mis en oeuvre, mais surtout à obtenir un produit pouvant évoluer relativement facilement.

IV.2.2. Architecture.

ORSEC est écrit pour l'essentiel en PASCAL 80 <MAU78> et pour le reste en METASYMBOL <CII77> , ce dernier langage permettant la définition de fonctions utilitaires telles que assignation dynamique, fermeture ou destruction de fichiers. Nous avons choisi PASCAL essentiellement parce que nous avons une bonne connaissance du langage et de son implantation sur IRIS 80.

La figure 4 donne l'architecture générale du système. Les modules qui le composent se partagent un ensemble de fichiers constituant une bibliothèque de spécifications. Un de ces fichiers contient le catalogue des types et des fonctions présents dans l'environnement et à chaque type sont associés trois fichiers contenant respectivement la table des symboles, les règles et le support associés à la spécification. Les noms de ces fichiers sont obtenus de façon standard à partir du nom de l'environnement et des noms des types.

ARCHITECTURE DU SYSTEME ORSEC



Les différents modules et leurs rôles respectifs sont :

ORSEC : C'est le module principal. Il est chargé du dialogue avec l'utilisateur et de l'appel des diverses fonctions du système.

SYNTAX : Module contenant l'analyseur lexical et syntaxique. Il a été construit en utilisant le générateur disponible sur le site grâce à P.Y. Cunin et M. Griffiths <C&G78>. Il interprète une table décrivant un automate, stockée sur mémoire secondaire.

SEMANT : Module contenant les fonctions sémantiques utilisées par l'analyseur syntaxique.

LOCALE : Module de gestion de la table des identificateurs locaux d'une spécification. Il est utilisé lors de l'analyse syntaxique qui précède la construction du support associé à la spécification source (commande COMPARER).

GLOBALE : Module de gestion de la table des identificateurs globaux de l'environnement.

ARBRES : Module de gestion des règles et des termes. Il contient en particulier les procédures effectuant la réécriture .

GRAPHE : Module de gestion des supports associés aux spécifications. Il contient les procédures de construction et de comparaison de ces supports.

SYSTEME : Ce module interface les fonctions de bas niveau permettant l'assignation dynamique des fichiers, leur fermeture et leur destruction. Ce module bien qu'écrit en PASCAL est particulièrement dépendant de l'implantation de PASCAL 80 et du système SIRIS 8.

METAMOD : Ce module ,écrit en METASYMBOL, implante les fonctions de bas niveau déjà citées. Celles ci s'utilisent à travers les fonctions du module SYSTEME. L'échange de données est fait par

l'intermediaire de variables externes.

En général les modules écrits en PASCAL n'exportent et n'importent que des procédures ou des fonctions.

Le transport du système ORSEC sur une autre installation nécessitera la réécriture du module principal ORSEC, les modules SYSTEME et METAMOD pourront éventuellement disparaitre, les fonctions qu'ils réalisent étant prédéfinies dans certains PASCAL (UCSD par exemple).

IV.3. UTILISATION DU SYSTEME

L'utilisation d'ORSEC ne pose aucun problème particulier; son lancement s'effectue à l'aide d'une "commande cataloguée" (le texte en est donné en annexe), l'utilisateur est alors guidé par le système. A l'issue d'une session, lors de la déconnexion, on aura soin de répondre affirmativement aux questions du système SIRIS 8 concernant la sauvegarde des fichiers créés au cours de l'utilisation d'ORSEC, sous peine de rendre incohérent l'environnement utilisé.

La construction des textes des spécifications s'effectue sous l'éditeur de textes SIRIS 8 et non sous le système ORSEC. En effet ,pour simplifier ce dernier, nous n'avons voulu ni interfacer l'éditeur ni en définir un spécifique.

La présence d'erreurs de syntaxe dans le texte d'un type abstrait soumis au système (commande COMPARER), est signalée par des messages d'erreurs et par le symbole "*" matérialisant l'emplacement de l'erreur dans la ligne source.

Les tailles des tables d'identificateurs sont fixes (200 pour la table

globale, 50 pour la table locale dans la version actuelle), en cas de débordement de ces tables des messages sont édités, mais l'environnement est préservé dans l'état précédent la dernière commande.

Les temps de réponses lors de l'accès "par les règles" dépendent bien évidemment de la charge de l'installation, de la complexité de la spécification source et de la taille de la bibliothèque; pour deux spécifications ε-équivalentes les durées de construction des supports affichées par ORSEC donnent une idée de leur "efficacités" relatives.

Pour conclure nous signalerons qu'ORSEC est "gourmand" en mémoire principale (recopie des termes lors de la réécriture) et en mémoire secondaire , chaque type de l'environnement nécessitant au moins trois quanta sur disque; de ce fait l'exécution peut avorter par manque d'espace mémoire principale ou secondaire.

V. CONCLUSIONS ET PERSPECTIVES

Arrivé au terme (momentané) de cette étude, nous allons, comme il se doit, faire un bilan des éléments qu'elle apporte. Nous le présenterons sous trois aspects différents :

comparaison avec des travaux voisins,

limites des concepts et de la réalisation et extensions possibles,

perspectives.

V.1. COMPARAISON AVEC DES TRAVAUX VOISINS

On pourra s'étonner du peu de références citées dans la partie essentielle de cette thèse (chapitre III). A notre connaissance, si l'équivalence de deux spécifications et les façons de la prouver ont été abordées par la plupart des auteurs s'intéressant aux types abstraits algébriques, l'utilisation d'une notion d'équivalence aux noms près pour l'accès à une bibliothèque de spécifications n'a pas été développée.

Le concept de ρ-équivalence ne nous semble pas avoir une importance théorique considérable, par contre il a, à notre avis, le mérite de souligner l'abstraction du concept de spécification algébrique de type, et d'être d'une utilité pratique non négligeable. De plus sa définition permet d'hériter des (de réutiliser les) travaux effectués dans le domaine.

Si sur le fond nous n'avons pu nous situer par rapport à "l'état de l'art", sur l'esprit on peut tenter quelques rapprochements.

Le principe sur lequel est fondée la réalisation est qu'on peut construire une hypothèse (universelle) par examen d'un nombre fini d'exemples

(particuliers) et généralisation. On retrouve là une analogie avec les travaux concernant la synthèse automatique de programmes à partir d'exemples menés en particulier par J. P. Jouannaud et Y. Kodratoff <KOD79>,<J&K80>. De la même façon les travaux de L. Bougé concernant le test de programmes et la production de jeux de test relèvent du même principe. On notera dans les chapitres 7 et 8 de sa thèse <BOU82> intitulés "Test et types abstraits algébriques" et "Validation de spécifications abstraites par test", d'une part que la construction des jeux de test (pour les spécifications algébriques) ressemble beaucoup à la construction de nos supports (à l'exception près que les termes instanciés sont clos), d'autre part que les spécifications qui ne se "testent pas bien" ne se "comparent pas bien" non plus (les ensembles par exemple).

V.2. LIMITES DE L'ETUDE ET EXTENSIONS POSSIBLES

Comme nous l'avons déjà noté, la ϱ-équivalence et la ϱ-inclusion ne couvrent pas l'ensemble des propriétés que l'on aimerait pouvoir établir entre deux spécifications.

Nous nous sommes placés dans le cadre des spécifications pour lesquelles les axiomes peuvent être orientés de façon à former un système de réécriture canonique, choix qui a des avantages évidents. La proposition 4 restreint encore le domaine d'application aux spécifications pour lesquelles on peut réduire un terme contenant des variables de types externes sans que l'affectation de valeurs à ces variables ne permette d'autre réduction. On s'interdit ainsi des spécifications de types tels que ensemble, multi-ensemble, liste triée... Ces limitations ne sont pas liées à la

ϱ-équivalence en soi mais à la volonté de pouvoir travailler sur des représentants des classes d'équivalence (et ainsi utiliser l'égalité syntaxique) plutôt que sur les classes elles-mêmes.

Cette même proposition et la définition qui l'introduit insistent sur la notion d'environnement au risque de perdre un peu de la généralité des propositions précédentes. Ceci nous semble toutefois être une approche raisonnable du problème et qui a le mérite de la simplicité. Une autre façon de procéder pourrait peut être consister à structurer de façon plus fine l'environnement et à s'autoriser le remplacement , dans une présentation I_s, d'un type s_i par un type s'_i dont on sait qu'il lui est ϱ-équivalent.

La ϱ-équivalence de deux "sous-spécifications" (suffisamment complète et consistante) de deux spécifications n'a pas été envisagée, essentiellement pour des raisons d'ordre pratique, afin de ne pas faire "exploser" la complexité des algorithmes mis en oeuvre; de plus il n'est pas simple de définir quand une telle "intersection" commune (aux noms près) est interessante. L'idée évoquée précédemment de partitionner la signature, d'une part en un "noyau" (suffisamment complet et consistant), d'autre part en un ensemble de fonctions définies par composition des précédentes, n'est pas satisfaisante , rien ne garantit en effet que l'on sache définir à priori un tel noyau.

On pourrait souhaiter mettre en évidence qu'une opération d'une présentation source peut être obtenue par la composition de plusieurs opérations d'une présentation cible. On aborde là, en partie, le problème de la construction automatique de la représentation d'un type abstrait par un autre qui englobe la remarque précédente. Il est clair que la ϱ-équivalence est un cas particulier dans lequel un type "s'implante" par un autre avec une correspondance opération à opération.

On peut envisager, en pratique, d'effectuer ces manipulations (composition

d'opérations, permutations sur l'ordre des arguments de même type) dans des limites raisonnables (compositions et permutations en nombre petit) ; dans ce cas on devra réfléchir à des moyens de diminuer le nombre de cas à examiner, de la même façon que nous éliminons les renommages non pertinents. Ce travail reste à faire.

Enfin nous n'avons pas réfléchi à l'extension de nos définitions et propositions au cadre des types abstraits paramétrés; on ne sait pas exprimer la ϱ-équivalence de deux spécifications paramétrées de types abstraits. On notera qu'un autre problème lié à la paramétrisation, celui de l'instanciation d'un type abstrait à partir d'une spécification paramétrée, fait intervenir un renommage des symboles paramètres formels par ceux des paramètres effectifs et une "équivalence" des contraintes portant sur les paramètres <EHR81>. Ce problème nous semble être différent du concept de ϱ-équivalence pour lequel on exige :

1) que les classes d'équivalence de termes engendrées par les congruences définies dans chaque présentation soient identiques après renommage,

2) que les opérations définies sur ces classes soient les mêmes, toujours après renommage.

Le problème de la validité d'un passage de paramètre semble porter plus sur le point 2), en effet pour les types paramètres formels on ne se donne généralement pas la signature complète, de ce fait on ne connait ni l'ensemble des termes clos, ni à fortiori l'ensemble quotient.

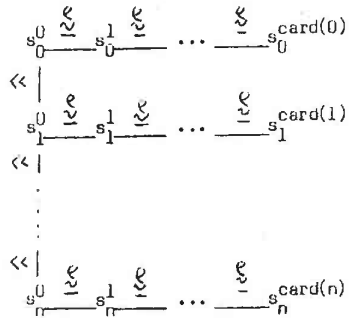
Comme nous l'avons précisé au chapitre IV, le prototype ORSEC n'est pas totalement conforme au point de vue adopté en III. Les modifications à y apporter pour remédier à cet état de fait ne présentent aucune difficulté nouvelle. De la même façon les performances du prototype (temps de réponse, taille mémoire utilisée) pourraient être notablement améliorées.

La réalisation et l'utilisation du produit nous ont permis de préciser la

vision intuitive que nous avions de l'équivalence de deux spécifications, et de valider l'hypothèse selon laquelle on peut inférer un renommage à partir d'exemples. Elles ont aussi fixé, de fait, les limites mentionnées précédemment.

ORSEC est un système qui émet des hypothèses sans efectuer de preuve, et il les émet par généralisation. Intuitivement cette généralisation sera fondée si la spécification vérifie une sorte de "continuité", ce qui ne sera pas le cas par exemple pour une pile de taille bornée. Comme nous l'avons vu en III.5.1. il semble qu'une partie des preuves puisse être effectuée avec des mécanismes relativement simples et apparemment automatisables; on peut donc envisager un étage supplémentaire dans le système, en aval du module de comparaison, effectuant ces preuves, éventuellement de façon interactive. Le premier étage effectuera dans ce cas un premier "tri" de façon à ne retenir que les renommages potentiellement fondés, le deuxième se chargeant de la preuve.

Une gestion un peu plus sophistiquée de l'environnement est aussi possible et nécessaire. Une idée simple consiste à partitionner l'environnement en "tranches" regroupant les types ϱ-équivalents, tout en privilégiant un représentant dans la tranche. On peut alors faire en sorte que la présentation de ce représentant n'utilise que les représentants des autres tranches. L'ensemble ordonné des représentants constituera alors l'environnement "standard". Lors de l'examen d'une nouvelle présentation, on commencera par la "standardiser" par renommage des symboles de façon à opérer sur l'environnement standard, et par ordonnancement des profils conformément à l'ordre défini sur les sortes. On peut représenter une telle structure par un "peigne"



l'hypothèse que :

$$\langle U_{i=0}^{n} s_i^{j_i}, U_{i=0}^{n} \underset{\sim}{\overset{\rho}{\Sigma}} s_{i_\iota}, U_{i=0}^{n} E_{s_{i_\iota}} \rangle$$

$$\langle U_{i=0}^{n} s_i^{0}, U_{i=0}^{n} \underset{\sim}{\overset{\rho}{\Sigma}} s_{i_\iota}, U_{i=0}^{n} E_{s_{i_\iota}} \rangle$$

pour tout j_0 dans $[0..card(0)]$,..., j_n dans $[0..card(n)]$

devra être prouvée.

V.3. PERSPECTIVES

Les perspectives ouvertes par ce travail ont déjà été évoquées dans les lignes qui précèdent, nous n'y reviendrons pas dans la suite de ce texte, mais nous serons amené à les préciser dans le cadre d'un contrat passé avec l'ADI pour la réalisation d'un prototype intégrant le système d'aide à la spécification VEGA, développé par J. J. et J. Chabrier, et ORSEC. Nous examinerons alors les modifications et les extensions réalisables dans les délais qui nous serons impartis.

Ce futur couple (VEGA,ORSEC) nous semble être un début prometteur de système

intégré de conception de spécifications. Pour aboutir à un système intégré de production de logiciel, il resterait à assurer la continuité des étapes allant des spécifications aux programmes en passant par les algorithmes, et ce autant que possible de façon assistée. L'aide fournie par ORSEC dans un tel système allégerait la tâche de développement en permettant l'accès à des programmes par la donnée d'une spécification. Des interrogations plus précises prenant en compte des critères d'implantation, tels que complexité des algorithmes et taille mémoire nécessaire, deviennent alors possibles.

Nous n'en sommes pas encore là, nous ne pouvons traiter n'importe quelle spécification, et le passage automatique d'une spécification à un algorithme, et à fortiori à un programme nécessite encore de nombreuses recherches.

D'ailleurs , même si ces problèmes étaient résolus, il resterait à fournir les outils permettant de passer de l'énoncé d'un problème à l'énoncé d'une solution.

Les sujets de réflexion ne manquent donc pas...

BIBLIOGRAPHIE

<ADA82> Reference Manual for the ADA Programming Language
Draft proposed ANSI standard document for editorial review.
July 1982

<ADAm78> A. Adam
"Utilisation des transformations sémantiques pour la correction automatique de programmes"
Thèse d'état. Paris VI Novembre 1978

<ADJ78> 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 methodologie. Vol 3. 1978

<B&D81> M. Bergman, P. Deransart
"Abstract data types and rewriting systems : application to the programming of algebraic abstract data types in PROLOG"
6th CAAP (Gènes) LNCS 112. 1981

<B&G77> R. M. Burstall, J. A. Goguen
"Putting theories together to make specifications"
Procedings 5th IJCAI. 1977

<B&W80> M. Broy, M. Wirsing
"Initial versus terminal algebra semantics for partially defined abstract data types"
Inst. fur Informatik, TU Munchen, Rep. TUM-I-8018. 1980

<BID81> M. Bidoit
"Une méthode de présentation des types abstraits : applications"
Thèse de IIIème cycle. Paris Sud Juin 1981

<BJM82> G. Barberye, T. Joubert, M. Martin

"OASIS: Un Outil d'Aide à la Spécification Interactive et Structurée"
Acte de journées BIGRE. Grenoble Janvier 1982

<BOU82> L. Bougé
"Modélisation de la notion de test de programme . Application à la production de jeux de test"
Thèse de III^{ème} cycle. Paris VI Octobre 1982

<CAL80> "Cohérence Assistée de Logiciel Documenté"
Rapport de fin de contrat 80/211
Agence de l'informatique 1980

<C&G78> P. Y. Cunin, M. Griffiths
"Générateur d'analyseur LL(1) sur IRIS 80"
CRIN 78-R-008 Février 1978

<CHA82> J. J. Chabrier
"Spécification et construction de systèmes orientés bases de données :
Techniques et langages basés sur le concept de type abstrait."
Thèse d'état. Nancy I Octobre 1982

<CII77> CII-HB
"METASYMBOL sous SIRIS 7/8. Manuel d'utilisation et d'opérations"
Ref 58F23753REV1 1977

<CLR81> C. Choppy, P. Lescanne, J. L. Rémy
"Improving abstract data type specification by an appropriate choice of constructors"
in Automatic program construction techniques
Ed A. Bierman, G. Guiho, Y. Kodratoff
Mac Millan Pub. Comp. 1981

<CUN82> P. Y. Cunin

"Système de construction modulaire de programmes"
Acte des journées BIGRE. Grenoble Janvier 1982

<DAH68> O. J. Dahl
"The SIMULA 67 common base language"
Norwegian Computing Center. Oslo 1968

<DERa82> P. Deransart
"Dérivation de programmes PROLOG à partir de spécifications algébriques"
INRIA Version provisoire. Mars 1982

<DERn74> J. C. Derniame
"Le projet CIVA un système de programmation modulaire"
Thèse d'état. Nancy I Janvier 1974

<DERs82> N. Dershowitz
"Orderings for term-rewriting systems"
Theoritical computer science 17. 1982

<D&F79> J. C. Derniame, J. P. Finance
"types abstraits de données : Spécification, utilisation et réalisation"
Cours de l'école d'été de l'AFCET. Monastir Juillet 1979

<DOD80> "Requirements for ADA Programming Support Environnements"
D. O. D. Février 1980

<EHR81> H. Ehrig
"Algebraic theorie of parameterized specifications with requirements"
6th CAAP (Gènes) LNCS 112 1981

<EHR82> H. Ehrig, H. J. Kreowski, B. Mahr, P. Padawitz
"Algebraic implementation of abstract data types"
Theoritical computer sciences. July 1982

<F&W80> P. Freeman, A. I. Wasserman

"Tutorial on software design techniques"

IEEE Catalog No EHO EMO 161-10. 1980

<FIN79> J. P. Finance

"Etude de la construction des programmes"

Thèse d'état. Nancy I Octobre 1979

<FOI82> J. Foisseau

"Assistance à la spécification des fonctions et des types de données dans SPRAC"

Actes des journées BIGRE. Grenoble Janvier 1982

<GAU78> M. C. Gaudel

"Specifications incomplètes mais suffisantes de la représentation des types abstraits"

Rapport Laboria n°320 . IRIA 1978

<GAU80> M. C. Gaudel

"Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation"

Thèse d'état. Paris VI 1980

<G&H78> J. V. Guttag, J. J. Horning

"The algebraic specification of abstract data types"

Acta Informatica 10. 1978

<GGM76> V. Giarratana, F. Gimona, U. Montanari

"Observability concepts in abstract data type specification"

Proc. 5th M.F.C.S. LNCS 45 1976

<GHM78> J. V. Guttag, E. H. Horowitz, D. R. Musser

"abstract data types and software validation"

Communication of ACM. Vol 21 Decembre 1978

<GJQ80> M. Grandbastien, J. Jaray, A. Quéré

"Présentation d'un projet de système interactif de spécification de problèmes"

Actes des journées Génie logiciel de l'AFCET. Toulouse Octobre 1980

<GOG78> J. A. Goguen

"Abstract errors for abstract data types"

Formal description of programming concepts.

North-Holland Pub. Comp. 1978

<GOG80> J. A. Goguen

"How to prove algebraic inductive hypotheses without induction, with applications to the correctness of data type implementation"

5th Conference on automated deduction (Les Arcs) LNCS 87 1980

<GUT80> J. V. Guttag

"Notes on type abstraction"

IEEE trans. on Soft. Eng. Vol 6 1980

<HEN80> P. Henry

"La connexion dans le projet TYP"

Thèse de IIIème cycle. Nancy I 1980

<H&O80> G. Huet, D. C. Oppen

"Equations and rewrite rules : a survey"

Formal languages : Perspectives and open problems

R. Book Ed. Academic Press 1980

<HOA72> C.A.R. Hoare

"Proof of correctness of data representations"

Acta Informatica Vol 1. 1972

<HUE77> G. Huet

"Confluent reductions : abstract propertises and applications to term-rewriting systems"

Rapport Laboria n° 250. IRIA Août 1977

<J&K80> J. P. Jouannaud, Y. Kodratoff

"An automatic construction of LISP programs by transformations of functions synthesized from their input-output behavior"

International Journal of Policy Analysis and Information Systems. Vol 4, n°4 1980

<JLR82> J. P. Jouannaud, P. Lescanne, F. Reinig

"Recursive decomposition ordering"

Formal description of programming concepts

North-Holland . Ed D. Bjorner. 1982

<KAM80> S. Kamin

"Final data type specification : a new data type specification method"

JACM n° 7. 1980

<K&B70> D. E. Knuth, P. B. Bendix

"Simple word problems in universal algebras"

Computational problems in abstract algebra

J. Leech Ed. Pergammon press 1970

<K&K82> C. et H. Kirchner

"Contribution à la résolution d'équations dans les algèbres libres et les variétés équationnelles d'algèbres"

Thèse de III ème cycle. Nancy I Mars 1982

<KOD79> Y. Kodratoff

"A class of functions synthesized from a finite number of examples and a LISP program scheme"

International Journal of Computer and Information Sciences. Vol 8, n°6 1979

<KRA82> S. Krakowiak

"Systèmes intégrés de production de logiciel : concepts et réalisations"

Technique et Science Informatiques. Vol 1, n°3 1982

<LAU78> J. P. Laurent

"un système qui met en évidence des erreurs sémantiques dans les programmes"

Thèse d'état. Paris VI Novembre 1978

<LEG80> J.L. Bouchenez, M. Loyer, L. Lucrece, P. Maurice, F. Prusker, J.C. Sogno, A.M.Vercoustre

"Le système LEGOS, environnement de programmation sur MITRA 125"

Acte de journées BIGRE. Rennes Decembre 1980

<LES79> P. Lescanne

"Etude algébrique et relationnelle des types abstraits et de leurs représentations"

Thèse d'état. INPL Septembre 1979

<LIS74> B. H. Liskov

"A note on CLU"

Comp. Structures Groupe. Memo 112. MIT project MAC

Cambridge Massachusset 1974

<MAU78> P. Maurice

"PASCAL 80. Manuel d'utilisation"

Service de synthèse et d'orientation de la recherche en informatique. Decembre 1978

<MIN79> R. Minot

"ATM un système de fabrication de programmes basé sur les concepts de modularité et de types abstraits"

Thèse de III ème cycle. Nancy I 1979

<MUS78> D. R. Musser

"A Data Type Verification System based on rewrite rules"

6th texas conf. on computing system structure 1978

<MUS80> D. R. Musser

"Abstract data type specification in the AFFIRM system"

IEEE trans. on Soft. Eng. Vol 6 n°1 1980

<PAI79> C. Pair

"La construction des programmes"

RAIRO informatique Vol 13, n° 2 1979

<PAI80> C. Pair

"Sur les modèles des types abstraits algébriques"

Séminaire du LITP. Mai 1980

CRIN 80-P-052 1980

<PRO79> K. Proch

"Paramétrisation des unités de compilation en ATM"

Rapport de DEA CRIN 79-R-058

<REM82a> J. L. Rémy

"Etude des systèmes de réécriture conditionnels et applications aux types abstraits algébriques"

Thèse d'état. INPL Juillet 1982

<REM82b> J. L. Rémy

"Fonction de normalisation, fonctions de transfert"

Communication privée. Octobre 1982

<R&V81> J. L. Rémy, P. A. S. Veloso

"An economical method for comparing data type specifications"

Sigplan Notices May 1981

<ROY83> A. Royer

"Développement et suivi de programmes"

Thèse de IIIème cycle. Nancy I à paraitre début 1983

<SOK80> Sok S. Chak

"Evolutivité du logiciel"

Thèse de IIIème cycle. Nancy I 1980

<SPR81> J. Foisseau, R. Jacquart, M. Lemaitre, M. Lemoine, G. Zanon

"Recherche sur la conception des programmes assistée par ordinateur"

Rapport détaillé n° 3/3155/DERI

CERT-DERI Octobre 1981

<WAN79> M. Wand

"Final algebra semantics and data type extension"

J. Comp. System Sciences. Vol 19. 1979

<WIR76> N. Wirth

"MODULA, a language for modular programming"

ETH. Zurich 1976

<WIR80> N. Wirth

"MODULA 2" (Manuel de definition)

ETH. Zurich 1980

<WLS76> W. A.Wulf, R. L. London, M. Shaw

"An introduction to the construction and verification of Alphard programs"

IEEE Trans. on Soft. Eng. Decembre 1976

ANNEXE I

SYNTAXE DU LANGAGE UTILISE

Jeu de caractères.

Les caractères utilisables sont :

- les lettres majuscules
- les chiffres
- les symboles '+', '-', '=', '(', ')', ',', ';', '>', '<', '*', ':'

Grammaire

La grammaire est présentée en utilisant les conventions suivantes :

- ':=:' est le symbole de dérivation
- '.' est le symbole de fin de règle
- ' | ' indique l'alternative dans les règles
- les non-terminaux sont désignés par des chaines de caractères majuscules
- les terminaux sont désignés soit par la chaine de caractères minuscules soulignée correspondante, soit par une suite de symboles entre '"', soit par 'identificateur' qui désigne toute chaine alphanumérique débutant par une lettre et différente d'un mot réservé, soit par 'entier' qui désigne une suite de chiffres.
- 'Λ' désigne le mot vide

Remarque :

Les mots-clés sont réservés ainsi que les mots 'BOOLEEN' et 'ENTIER'qui sont des identificateurs de types prédéfinis.

PRESENTATION ::= type identificateur
PARTIE_PROFILS
PARTIE_DECLARATIONS
PARTIE_REGLES
fin .

PARTIE_PROFILS ::= profils PROFIL LISTE_DE_PROFILS .

LISTE_DE_PROFILS ::= ^ | PROFIL LISTE_DE_PROFILS .

PROFIL ::= identificateur PARAMETRES ":" identificateur ";" .

PARAMETRES ::= ^ | "(" identificateur LISTE_IDENTIFICATEURS ")" .

LISTE_IDENTIFICATEURS ::= ^ | "," identificateur LISTE_IDENTIFICATEURS .

PARTIE_DECLARATIONS ::= ^ | var DECLARATION LISTE_DE_DECLARATIONS .

DECLARATION ::= identificateur LISTE_IDENTIFICATEURS ":" identificateur ";" .

LISTE_DE_DECLARATIONS ::= ^ | DECLARATION LISTE_DE_DECLARATIONS .

PARTIE_REGLES ::= REGLE LISTE_DE_REGLES .

LISTE_DE_REGLES ::= ^ | REGLE LISTE_DE_REGLES .

REGLE ::= identificateur SUITE_MEMBRE_GAUCHE "->" MEMBRE_DROIT ";" .

SUITE_MEMBRE_GAUCHE ::= ^ | "(" EXPRESSION LISTE_EXPRESSIONS ")" .

MEMBRE_DROIT ::= EXPRESSION |
si EXPRESSION alors MEMBRE_DROIT
sinon MEMBRE_DROIT |
erreur .

LISTE_EXPRESSIONS ::= ^ | "," EXPRESSION LISTE_EXPRESSIONS .

EXPRESSION ::= SOUS_EXPRESSION_SANS_OU DISJONCTION .

DISJONCTION ::= ^ | ou SOUS_EXPRESSION_SANS_OU DISJONCTION .

SOUS_EXPRESSION_SANS_OU ::= SOUS_EXPRESSION_SANS_ET CONJONCTION .

CONJONCTION ::= ^ | et SOUS_EXPRESSION_SANS_ET CONJONCTION .

SOUS_EXPRESSION_SANS_ET ::= SOUS_EXPRESSION_SANS_NON
| non SOUS_EXPRESSION_SANS_ET .

SOUS_EXPRESSION_SANS_NON ::= EXPRESSION_SIMPLE COMPARAISON .

COMPARAISON ::= ^ | OPERATEUR_DE_RELATION EXPRESSION_SIMPLE .

EXPRESSION_SIMPLE ::= OPERANDE SOMME .

SOMME ::= ^ | OPERATEUR_PLUS_MOINS OPERANDE SOMME .

OPERANDE ::= FACTEUR PRODUIT .

PRODUIT ::= ^ | OPERATEUR_MULT_DIV_MOD FACTEUR PRODUIT .

FACTEUR ::= OPERATEUR_UNAIRE TERME | TERME .

TERME ::= CONSTANTE | "(" EXPRESSION ")"
| identificateur ARGUMENTS .

ARGUMENTS ::= ^ | "(" EXPRESSION LISTE_EXPRESSIONS ")" .

OPERATEUR_RELATION ::= ">" | ">=" | "<" | "<=" | "<>" | "=" .

OPERATEUR_MULT_DIV_MOD ::= "*" | div | mod .

OPERATEUR_UNAIRE ::= "+" | "-" .

OPERATEUR_PLUS_MOINS ::= "+" | "-" .

CONSTANTE ::= entier | vrai | faux .

Signification des constructions syntaxiques

L'identificateur situé après le mot 'TYPE' est le nom du type spécifié.

La PARTIE_PROFILS est la liste des profils des opérations de la

présentation, chaque profil se compose du nom de l'opération suivie, s'il y a lieu, de la liste des identificateurs des types des arguments; le symbole ':' introduit le nom du type du résultat.

La PARTIE_DECLARATION peut être vide, sinon chaque DECLARATION se compose d'une liste d'identificateurs de variables; le symbole ':' précède l'identificateur du type des variables de la liste.

La PARTIE_REGLES contient une liste de règles. Chaque règle s'interprète comme une règle de réécriture. Les variables y apparaissant sont considérées comme universellement quantifiées et doivent être déclarées dans la PARTIE_DECLARATION. Les membres droits et gauches de ces règles peuvent contenir des identificateurs d'opérations définies dans d'autres présentations, si celles ci sont présentes dans l'environnement. Les deux membres d'une règle doivent être de même type.

Tout type de l'environnement peut être utilisé dans une présentation; réciproquement les seuls identificateurs utilisables dans une présentation sont les identificateurs de l'environnement et ceux déclarés dans la présentation. On rappelle que le système ne s'assure pas de la confluence de l'ensemble des règles.

ANNEXE II

CARACTERISTIQUES TECHNIQUES D'ORSEC SUR IRIS 80

Taille du module de chargement.

Le module de chargement occupe 14 quanta sur disque, soit 112 K octets.

Commande de mise en oeuvre : ORSEC.

Le texte de la commande ORSEC est le suivant :

```
!LIMIT (NDP),(CORE,40),(PAGES,50),(SPDISC,50),(TIME,2)
!SLIMIT (DP),(CORE,220,40)
!ASSIGN LM,FIL,(STS,OLD),(NAM,ORSECLM)
!ASSIGN SI,FIL,(STS,OLD),(NAM,TREEORSEC)
!ASSIGN LO,FIL,(STS,MOD),(NAM,FLYTOX)
!ASSIGN TTYI,DEV,IN
!ASSIGN TTYO,DEV,OUT
!ASSIGN FTAB,MTN,FIL,(STS,OLD),(NAM,SYNTAXE)
!ASSIGN PDP,MTN,FIL,(STS,OLD),(NAM,BIDON)
!RUN ,OPTION,F=10
```

Afin de ne pas avoir de problème d'espace mémoire (allocation dynamique), nous utilisons 220 pages de mémoire en pagination (le module de chargement en nécessite 66).

L'étiquette logique "SI" doit être assignée à un fichier existant quelconque, ceci à cause de l'ouverture implicite du fichier "INPUT" effectuée par le "moniteur" PASCAL.

L'assignation de "LO" peut être remplacée par

!ASSIGN LO,DUM

L'assignation au fichier "FLYTOX" permettra de récupérer les messages d'erreur édités par le "moniteur" PASCAL en cas de vice caché du produit.

"TTYI" et "TTYO" sont assignées au périphérique utilisateur.

L'étiquette logique "FTAB" est assignée au fichier "SYNTAXE" contenant la table générée par le générateur d'analyseur syntaxique et interprétée par le module "SYNTAX".

L'assignation de "PDP" à un fichier quelconque (et non obligatoirement existant) a pour rôle de référencer le support des fichiers (PDP dans la terminologie CII) et est utilisée dans le module "METAMOD".

L'option F=10 figurant dans l'appel du chargeur fournit le nombre de pages à réserver pour la gestion des divers fichiers.

La commande "ORSEC" s'utilise en temps partagé uniquement, par la simple frappe du nom de la commande lorsque SIRIS 8 émet un "!".

Fichiers construits par ORSEC.

L'environnement de l'utilisateur est conservé dans un certain nombre de fichiers :

* un fichier "catalogue" dont le nom physique est une chaine de 17 caractères constituée par le nom de l"environnement suivi de caractères "_" .

Ce fichier contient la table de tous les symboles ou identificateurs (de types et de fonctions), prédéfinis ou définis par l'utilisateur.
A chaque symbole est associée sa nature (T(ype) ou F(onction), les opérateurs sont assimilés à des fonctions), un entier qui est la représentation interne du symbole, et une liste d'entiers représentant soit le profil pour les symboles de fonctions, soit la liste des types utilisés pour les symboles de types.

* à chaque type défini par l'utilisateur sont associés trois fichiers dont les noms sont constitués du nom de l'environnement suivi de,
(8 - longueur du nom de l'environnement) caractères "-" ,
suivis d'une lettre ("L", "R", "G") ,
suivie de l'identificateur du type.

  ** le fichier de nom <environnement> ("-")* "L" <nom du type>
  contient la table des identificateurs définis dans la présentation du type (identificateurs de type, de fonction et de variables). Ce fichier est utilisé lorsqu'on désire imprimer le texte d'une présentation (commande "TYPE").

  ** un fichier de nom <environnement> ("-")* "R" <nom du type>
  contient la partie axiomes de la présentation. Les termes sont représentés sous forme préfixée, chaque symbole de fonction ou de variable, chaque opérateur, chaque valeur est représenté par un couple

  (no_du_symbole, type_du_symbole)

  ou no_du_symbole est la codification interne du symbole
  et type_du_symbole est en entier valant
  1 pour les variables
  2 pour les valeurs (notations de constantes entières ou valeurs booléennes)

3 pour les fonctions (définies par l'utilisateur)

4 pour erreur

5 pour les opérateurs prédéfinis .

Les symboles "(", ")", "->", ";" apparaissant dans les axiomes de la présentation sont représentés respectivement par les couples (-3.0), (-4,0), (-5,0), (-6,0). Le couple (-7,0) termine le fichier.

Ce fichier est utilisé lorsqu'on désire imprimer le texte de la présentation et lorsqu'on construit le support d'un type nécessitant l'utilisation de ces règles.

** le fichier de nom <environnement> ("-")* "G" <nom du type>

contient le support associé au type.

Chaque symbole d'une formule

$$FNn\ o\ t_1..t_n\ t_{n+1}$$

est représenté par une chaine de 6 caractères (3_lettres,3_chiffres) ou 3_chiffres est pour les symboles d'opération , la codification interne du symbole (figurant dans le fichier "catalogue").

Tous ces fichiers sont accessibles par l'éditeur de texte SIRIS 8, et sont "lisibles" avec un peu d'habitude.

Remarques.

Lors d'une session ORSEC on va être amené à créer de tels fichiers. A l'issue de la session temps partagé le système SIRIS 8 pose pour chaque fichier nouveau la question

SAVE .... /0001 (Y/N) ?

on aura soin de répondre affirmativement sous peine de rendre l'environnement incohérent.

ANNEXE III

EXEMPLES D'UTILISATION

Les exemples suivants sont des listings d'éxécution d'ORSEC. Nous les avons complétés avec des remarques manuscrites afin de guider le lecteur. Les textes en caractères majuscules sont imprimés par le système, ceux en minuscules sont tapés à la console par l'utilisateur.

```
lorsec
- SFER/PASCAL-SYSTEM,VERS. 1/11/79-U4
                    O R S E C   A   V O T R E   S E R V I C E

VERSION  1.2 -OCT  82
16*11*82

NOM DE L'ENVIRONNEMENT ?demo
ENVIRONNEMENT CREE.

QUE VOULEZ-VOUS FAIRE ?catalog
LISTE DES TYPES ET DES OPERATIONS

 TYPE  ENTIER    <>
 TYPE  BOOLEEN   <>
 FONCTION  SI      :?
 FONCTION  PRE     :BOOLEEN
 FONCTION  &       (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  DIV     (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  =       (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  ET      (BOOLEEN ,BOOLEEN ):BOOLEEN
 FONCTION  <=      (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  <       (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  MOD     (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  -       (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  *       (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  NON     (BOOLEEN ):BOOLEEN
 FONCTION  OU      (BOOLEEN ,BOOLEEN ):BOOLEEN
 FONCTION  +       (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  >=      (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  >       (ENTIER  ,ENTIER  ):BOOLEEN

FIN DE LISTE
QUE VOULEZ-VOUS FAIRE ?terminal

DONNEZ LE NOM DU FICHIER DE SORTIE DES RESULTATS
OU FRAPPEZ "TTY" SI VOUS LES VOULEZ A LA CONSOLE ?listing
QUE VOULEZ-VOUS FAIRE ?catalog

QUE VOULEZ-VOUS FAIRE ?tecccc

                LISTE DES COMMANDES DISPONIBLES

   CATALOG  : PERMET D 'OBTENIR LA LISTE DES TYPES ET DES FONCTIONS
              PRESENTS DANS L'ENVIRONNEMENT.
              CHAQUE TYPE EST SUIVI DE SA LISTE D'IMPORTATION,
              CHAQUE FONCTION DE SON PROFIL.
   COMPARER : COMPARAISON D'UNE UNITE AVEC CELLES
              DE L'ENVIRONNEMENT.
   TYPE     : PERMET D'OBTENIR LE TEXTE D'UN TYPE.
   DETRUIRE : DESTRUCTION D'UNE UNITE.
   RELATION : PERMET D'OBTENIR LES TYPES UTILISES PAR OU
              UTILISANT UN TYPE DONT ON DONNERA LE NOM
```

Impression des types et des fonctions de l'environnement (types et opérateurs prédéfinis)

l'environnement "demo" n'existait pas ; il est créé.

inutilisé dans la version actuelle

symbole "#"

A partir de ce moment, les impressions sont faites dans le fichier "listing" ...

... sauf le dialogue...

... et le "menu".

```
STOP     : ARRET DE LA SESSION.
TERMINAL : PERMET DE CHOISIR LE FICHIER DE SORTIE DES
           RESULTATS (FICHIER SUR DISQUE OU TERMINAL).

                    ******************************

QUE VOULEZ-VOUS FAIRE ?terminal

DONNEZ LE NOM DU FICHIER DE SORTIE DES RESULTATS
OU FRAPPEZ "TTY" SI VOUS LES VOULEZ A LA CONSOLE ?tty
```
retour au périphérique standard.
```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex21
TYPE BOOLEAN
  PROFILS
    TRUE  :BOOLEAN;
    FALSE :BOOLEAN;
    NOT(BOOLEAN)  :BOOLEAN;
    TAUT(BOOLEAN) :BOOLEAN;
    CONTRA(BOOLEAN):BOOLEAN;
    AND(BOOLEAN,BOOLEAN)  :BOOLEAN;
    OR(BOOLEAN,BOOLEAN)   :BOOLEAN;
    XOR(BOOLEAN,BOOLEAN)  :BOOLEAN;
    NOR(BOOLEAN,BOOLEAN)  :BOOLEAN;
    IF(BOOLEAN,BOOLEAN)   :BOOLEAN;
    IFF(BOOLEAN,BOOLEAN)  :BOOLEAN;
    NAND(BOOLEAN,BOOLEAN) :BOOLEAN;
  VAR X,Y:BOOLEAN;
  AXIOMES
    NOT(TRUE)    -> FALSE;
    NOT(FALSE)   -> TRUE;
    TAUT(X)      -> TRUE;
    CONTRA(X)    -> FALSE;
    AND(TRUE,TRUE)  -> TRUE;
    AND(FALSE,X)    -> FALSE;
    AND(X,FALSE)    -> FALSE;
    OR (X,Y)        -> NOT(AND(NOT(X),NOT(Y)));
    XOR(X,Y)        -> OR(AND(X,NOT(Y)),AND(Y,NOT(Y)));
    NOR(X,Y)        -> AND(NOT(X),NOT(Y));
    IF(X,Y)         -> NOT(AND(X,NOT(Y)));
    IFF(X,Y)        -> AND(IF(X,Y),IF(Y,X));
    NAND(X,Y)       -> NOT(AND(X,Y));
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:
FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui
```
présentation du type BOOLEAN ...

NOT, AND et OR servent à définir les autres opérations ...

... sauf ici, pour simplifier l'écriture

On remarquera qu'on n'effectue pas de comparaison avec les types prédéfinis.

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex22
TYPE LOGIC
  PROFILS
    WAR  :LOGIC;
    FALSCH:LOGIC;
    NICHT(LOGIC)  :LOGIC;
    UND(LOGIC,LOGIC)  :LOGIC;
    ODER(LOGIC,LOGIC) :LOGIC;
    OB(LOGIC,LOGIC)   :LOGIC;
    OBB(LOGIC,LOGIC)  :LOGIC;
    BASIC(LOGIC,LOGIC):LOGIC;
  VAR X,Y:LOGIC;
  AXIOMES
    BASIC(WAR,WAR)    ->FALSCH;
    BASIC(X,FALSCH)   ->WAR;
    BASIC(FALSCH,X)   ->WAR;
    NICHT(X)          ->BASIC(X,X);
    ODER(X,Y)         ->BASIC(BASIC(X,X),BASIC(Y,Y));
    UND(X,Y)          ->BASIC(BASIC(X,Y),BASIC(Y,X));
    OB(X,Y)           ->BASIC(X,BASIC(X,Y));
    OBB(X,Y)          ->UND(OB(X,Y),OB(Y,X));
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:        2 SECONDES
DUREE DE LA COMPARAISON:                    5 SECONDES

                PROPOSITION
        ***************************
        *  LOGIC     *  BOOLEAN   *
        ***************************
        *  UND       *  AND       *
        *  ODER      *  OR        *
        *  OB        *  IF        *
        *  OBB       *  IFF       *
        *  BASIC     *  NAND      *
        *  NICHT     *  NOT       *
        *  WAR       *  TRUE      *
        *  FALSCH    *  FALSE     *
        ***************************
        *  LOGIC      < BOOLEAN ? *
        ***************************

FAUT-IL CONSERVER LE TYPE (OUI/NON)?non
```
présentation du type LOGIC

les opérations sont définies en utilisant BASIC ...

le système propose l'hypothèse de p-inclusion et le renommage correspondant.

On remarquera que pour ce cas d'espèce la proposition est prouvée exhaustivement.

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex14
TYPE INT1
 PROFILS
    ZERO1          :INT1;
    SUCC1(INT1)    :INT1;
    PRED1(INT1)    :INT1;
    OPP1(INT1)     :INT1;
    ADD1(INT1,INT1):INT1;
 VAR I1,I2:INT1;
 AXIOMES
   SUCC1(PRED1(I1)) ->I1;
   PRED1(SUCC1(I1)) ->I1;
   OPP1(ZERO1)      ->ZERO1;
   OPP1(SUCC1(I1))  ->PRED1(OPP1(I1));
   OPP1(PRED1(I1))  ->SUCC1(OPP1(I1));
   ADD1(ZERO1,I1)   ->I1;
   ADD1(SUCC1(I1),I2)->SUCC1(ADD1(I1,I2));
   ADD1(PRED1(I1),I2)->PRED1(ADD1(I1,I2));
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          2 SECONDES
DUREE DE LA COMPARAISON:          1 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui
```

Une présentation des entiers relatifs avec les constructeurs ZERO1, PRED1, SUCC1

On la conserve dans l'environnement.

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex15
TYPE INT2
 PROFILS
    ZERO2          :INT2;
    SUCC2(INT2)    :INT2;
    PRED2(INT2)    :INT2;
    OPP2(INT2)     :INT2;
    ADD2(INT2,INT2):INT2;
 VAR I1,I2:INT2;
 AXIOMES
    SUCC2(OPP2(SUCC2(I2)))->OPP2(I2);
    OPP2(ZERO2)           ->ZERO2;
    OPP2(OPP2(I2))        ->I2;
    PRED2(ZERO2)          ->OPP2(SUCC2(ZERO2));
    PRED2(SUCC2(I2))      ->I2;
    PRED2(OPP2(I2))       ->OPP2(SUCC2(I2));
    ADD2(ZERO2,I1)        ->I1;
    ADD2(SUCC2(I1),I2)    ->SUCC2(ADD2(I1,I2));
    ADD2(OPP2(I1),I2)     ->OPP2(ADD2(I1,OPP2(I2)));
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          3 SECONDES
DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:          3 SECONDES

        PROPOSITION
 ***************************
 *  INT2       *  INT1      *
 ***************************
 *  ADD2       *  ADD1      *
 *  SUCC2      *  SUCC1     *
 *  PRED2      *  PRED1     *
 *  OPP2       *  OPP1      *
 *  ZERO2      *  ZERO1     *
 ***************************
 *  INT2     <==>INT1    ? *
 ***************************


        PROPOSITION
 ***************************
 *  INT2       *  INT1      *
 ***************************
 *  ADD2       *  ADD1      *
 *  SUCC2      *  PRED1     *
 *  PRED2      *  SUCC1     *
 *  OPP2       *  OPP1      *
 *  ZERO2      *  ZERO1     *
 ***************************
 *  INT2     <==>INT1    ? *
 ***************************

FAUT-IL CONSERVER LE TYPE (OUI/NON)?non
```

Une autre présentation dont on s'apercevra qu'elle est, à peu d'équivalente à la précédente. On utilise cette fois ZERO2, SUCC2, OPP2 comme constructeurs.

Les deux renommages possibles sont mis en évidence ...
Dans cet exemple les types "externes" n'interviennent pas

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex3
TYPE NAT
  PROFILS
  ZNAT    :NAT;
  SNAT  (NAT):NAT;
  PNAT  (NAT):NAT;
  ADDNAT  (NAT,NAT):NAT;
  MOINAT  (NAT,NAT):NAT;
  INFNAT (NAT,NAT):BOOLEEN;
  VAR X,N: NAT;
  AXIOMES
  PNAT(ZNAT) -> ZNAT;
  PNAT(SNAT(X)) -> X;
  MOINAT(X,ZNAT) -> X;
  MOINAT(X,SNAT(N)) -> PNAT(MOINAT(X,N));
  ADDNAT(X,ZNAT) -> X;
  ADDNAT(X,SNAT(N)) -> SNAT(ADDNAT(X,N));
  INFNAT(ZNAT,SNAT(X)) ->VRAI;
  INFNAT(SNAT(X),ZNAT) ->FAUX;
  INFNAT(SNAT(X),SNAT(N)) ->INFNAT(X,N);
  INFNAT(ZNAT,ZNAT) ->FAUX;
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          1 SECONDES
DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui
```

Une présentation des entiers naturels ...

On ne les confond pas avec les entiers relatifs "INT1"

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex23
TYPE REL
  PROFILS
  ZEREL :REL;
  C (BOOLEEN,NAT):REL;
  ADDREL (REL,REL):REL;
  OPPREL (REL):REL;
  SUCCREL (REL):REL;
  PREDREL (REL):REL;

  VAR R1,R2:REL;
      B1:BOOLEEN; N,N2:NAT;
  AXIOMES
   ZEREL ->C(VRAI,ZNAT);
   C(FAUX,ZNAT) ->C(VRAI,ZNAT);
   SUCCREL(C(VRAI,N)) -> C(VRAI,SNAT(N));
   SUCCREL(C(FAUX,ZNAT)) -> C(VRAI,SNAT(ZNAT));
   SUCCREL(C(FAUX,SNAT(N))) ->C(FAUX,N);
   PREDREL(C(FAUX,N)) ->C(FAUX,SNAT(N));
   PREDREL(C(VRAI,ZNAT)) ->C(FAUX,SNAT(ZNAT));
   PREDREL(C(VRAI,SNAT(N))) ->C(VRAI,N);
   OPPREL(C(VRAI,N)) ->C(FAUX,N);
   OPPREL(C(FAUX,N)) ->C(VRAI,N);
   ADDREL(C(B1,N),C(B1,N2)) ->C(B1,ADDNAT(N,N2));
   ADDREL(C(VRAI,N),C(FAUX,N2)) ->SI INFNAT(N,N2) ALORS C(FAUX,MOINAT(N2,N))
                                  SINON C(VRAI,MOINAT(N,N2));
   ADDREL(C(FAUX,N),C(VRAI,N2)) ->SI INFNAT(N,N2) ALORS C(VRAI,MOINAT(N2,N))
                                  SINON C(FAUX,MOINAT(N,N2));
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          44 SECONDES
DUREE DE LA COMPARAISON:          3 SECONDES

DUREE DE LA COMPARAISON:          12 SECONDES

          PROPOSITION
 ****************************
 *  REL       *  INT1       *
 ****************************
 *  ADDREL    *  ADD1       *
 *  SUCCREL   *  SUCC1      *
 *  PREDREL   *  PRED1      *
 *  OPPREL    *  OPP1       *
 *  ZEREL     *  ZERO1      *
 ****************************
 *  REL          > INT1   ? *
 ****************************

          PROPOSITION
 ****************************
 *  REL       *  INT1       *
 ****************************
 *  ADDREL    *  ADD1       *
 *  PREDREL   *  SUCC1      *
 *  SUCCREL   *  PRED1      *
 *  OPPREL    *  OPP1       *
 *  ZEREL     *  ZERO1      *
 ****************************
 *  REL          > INT1   ? *
 ****************************
```

Une présentation des entiers relatifs sous forme de couples (signe, valeur absolue) utilisant les types BOOLEEN et NAT

Le système met en évidence les deux renommages possibles avec INT1 (INT2 n'a pas été envisagé). La proposition est moins "sûre", on a introduit un terme avec variables C(x1, x2)

```
DUREE DE LA COMPARAISON:          2 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?non


QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex20
 TYPE FILE                                  Une file ...
   PROFILS
     FILEVIDE : FILE;
     ENFILER (FILE,ENTIER):FILE;
     DEFILER (FILE):FILL;
     PREMIER(FILE):ENTIER;
   VAR P1:FILE; E1,E2:ENTIER;
  AXIOMES
     DEFILER(FILEVIDE)->ERREUR;
     DEFILER(ENFILER(FILEVIDE,E1))->FILEVIDE;
     DEFILER(ENFILER(ENFILER(P1,E1),E2))->ENFILER(DEFILER(ENFILER(P1,E1)),E2);
     PREMIER(FILEVIDE)->ERREUR;
     PREMIER(ENFILER(FILEVIDE,E1))->E1;
     PREMIER(ENFILER(ENFILER(P1,E1),E2))->PREMIER(ENFILER(P1,E1));
 FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          1 SECONDES
DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui
```

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex4
 TYPE FILE2                                 ... une autre file, les
   PROFILS                                  axiomes sont différents...
     VIDEFIL2 : FILE2;
     ENFILER2 (FILE2,ENTIER):FILE2;
     DEFILER2 (FILE2):FILE2;
     PREMIER2(FILE2):ENTIER;
     EGALVID2 (FILE2):BOOLEEN;
   VAR P1,P2:FILE2; E1,E2:ENTIER;
  AXIOMES
     DEFILER2(VIDEFIL2)->ERREUR;
     DEFILER2(ENFILER2(P1,E1)) -> SI EGALVID2(P1) ALORS VIDEFIL2
                                  SINON ENFILER2(DEFILER2(P1),E1);
     PREMIER2(ENFILER2(P1,E1)) ->SI EGALVID2(P1) ALORS E1
                                    SINON PREMIER2(P1);

     PREMIER2(VIDEFIL2)->ERREUR;
     EGALVID2(VIDEFIL2) -> VRAI;
     EGALVID2(ENFILER2(P1,E1))->FAUX;
 FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          2 SECONDES
DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES


          PROPOSITION
     ****************************
     *  FILE2     *  FILE      *
     ****************************
     *  ENFILER2  *  ENFILER   *
     *  DEFILER2  *  DEFILER   *
     *  PREMIER2  *  PREMIER   *
     *  VIDEFIL2  *  FILEVIDE  *
     ****************************
     *  FILE2      > FILE    ? *
     ****************************

FAUT-IL CONSERVER LE TYPE (OUI/NON)?non
```

... on obtient le résultat attendu

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex14
TYPE INT1
 PROFILS
    ZERO1          :INT1;
    SUCC1(INT1)    :INT1;
    PRED1(INT1)    :INT1;
    OPP1(INT1)     :INT1;
    ADD1(INT1,INT1):INT1;
 VAR I1,I2:INT1;
 AXIOMES
   SUCC1(PRED1(I1)) ->I1;
   PRED1(SUCC1(I1)) ->I1;
   OPP1(ZERO1)      ->ZERO1;
   OPP1(SUCC1(I1))  ->PRED1(OPP1(I1));
   OPP1(PRED1(I1))  ->SUCC1(OPP1(I1));
   ADD1(ZERO1,I1)   ->I1;
   ADD1(SUCC1(I1),I2)->SUCC1(ADD1(I1,I2));
   ADD1(PRED1(I1),I2)->PRED1(ADD1(I1,I2));
FIN
```

On soumet au système la même présentation INT1 que précédemment ...

```
 TYPE SYNTAXIQUEMENT CORRECT
 ***************************
IDENTIFICATEUR <INT1    > DEJA PRESENT
RENOMMEZ-LE ?i

IDENTIFICATEUR <ZERO1   > DEJA PRESENT
RENOMMEZ-LE ?z

IDENTIFICATEUR <SUCC1   > DEJA PRESENT
RENOMMEZ-LE ?s

IDENTIFICATEUR <PRED1   > DEJA PRESENT
RENOMMEZ-LE ?p

IDENTIFICATEUR <OPP1    > DEJA PRESENT
RENOMMEZ-LE ?o

IDENTIFICATEUR <ADD1    > DEJA PRESENT
RENOMMEZ-LE ?a
```

... nous n'acceptons pas la surcharge des identificateurs, ORSEC demande de nouveaux noms ...

```
DUREE DE LA CONSTRUCTION DU SUPPORT:        5 SECONDES
DUREE DE LA COMPARAISON:          1 SECONDES
DUREE DE LA COMPARAISON:          3 SECONDES
```

... mais il n'y a pas "recompilation" ...

```
        PROPOSITION
 ***************************
 *  I         *  INT1       *
 ***************************
 *  A         *  ADD1       *
 *  S         *  SUCC1      *
 *  P         *  PRED1      *
 *  O         *  OPP1       *
 *  Z         *  ZERO1      *
 ***************************
 *  I       <==>INT1     ? *
 ***************************
```

... on retrouve bien sûr les hypothèses de renommage ...

```
        PROPOSITION
 ***************************
 *  I         *  INT1       *
 ***************************
 *  A         *  ADD1       *
 *  S         *  SUCC1      *
 *  P         *  PRED1      *
 *  O         *  OPP1       *
 *  Z         *  ZERO1      *
 ***************************
 *  I       <==>INT1     ? *
 ***************************

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?non

QUE VOULEZ-VOUS FAIRE ?comparer
```

Une pile maintenant ...

```
NOM DU FICHIER SOURCE  ?ex2
 TYPE PILE
   PROFILS
     VIDE : PILE;
     EMPILER (PILE,ENTIER):PILE;
     DEPILER (PILE):PILE;
     SOMMET(PILE):ENTIER;
   VAR P1,P2:PILE; E1,E2:ENTIER;
  AXIOMES
     DEPILER(VIDE)->ERREUR;
     DEPILER(EMPILER(P1,E1)) -> P1;
     SOMMET(EMPILER(P1,E1)) ->E1;
     SOMMET(VIDE)->ERREUR;
 FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:        1 SECONDES
DUREE DE LA COMPARAISON:          0 SECONDES
DUREE DE LA COMPARAISON:          0 SECONDES
DUREE DE LA COMPARAISON:          0 SECONDES
DUREE DE LA COMPARAISON:          0 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui

QUE VOULEZ-VOUS FAIRE ?stop

    AU REVOIR...
- END PASCAL
!
```

... ce qui est différent d'une pile, des autres relatifs et naturels.

arrêt de la session.

```
1
SAVE DEMO-------------/0001 (Y/N)? y
SAVE LISTING          /0001 (Y/N)? n
SAVE DEMO----LBOOLEAN /0001 (Y/N)? y
SAVE DEMO----RBOOLEAN /0001 (Y/N)? y
SAVE DEMO----GBOOLEAN /0001 (Y/N)? y
SAVE DEMO----LINTI    /0001 (Y/N)? y
SAVE DEMO----RINTI    /0001 (Y/N)? y
SAVE DEMO----GINTI    /0001 (Y/N)? y
SAVE DEMO----LNAT     /0001 (Y/N)? y
SAVE DEMO----RNAT     /0001 (Y/N)? y
SAVE DEMO----GNAT     /0001 (Y/N)? y
SAVE DEMO----LFILE    /0001 (Y/N)? y
SAVE DEMO----RFILE    /0001 (Y/N)? y
SAVE DEMO----GFILE    /0001 (Y/N)? y
SAVE DEMO----LPILE    /0001 (Y/N)? y
SAVE DEMO----RPILE    /0001 (Y/N)? y
SAVE DEMO----GPILE    /0001 (Y/N)? y
LOGOUT DONE AT  19*15*02

?? IUCASSTS    IS DISCONNECTED 00 DAY:0320,HOUR:0019,MIN:0014

? en

?? BYE DAY:0320,HOUR:0019,MIN:0014
```

lors de la déconnexion on prend la garde de conserver les fichiers constituant l'environnement.

```
orsec
- SFER/PASCAL-SYSTEM,VERS. 1/11/79-U4
                    O R S E C   A   V O T R E   S E R V I C E

VERSION  1.2 -OCT  82
17*11*82

NOM DE L'ENVIRONNEMENT ?demo

QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex1
TYPE LISTE
  PROFILS
  NIL:LISTE;
  MAKE(ENTIER):LISTE;
  APPEND(LISTE,LISTE):LISTE;
  CONS(LISTE,ENTIER):LISTE;
  DEL(LISTE):LISTE;
  VAR
          ENT:ENTIER;
          L1,L2:LISTE;
  AXIOMES
        APPEND(NIL,L1)->L1;
        APPEND(MAKE(ENT),NIL)->MAKE(ENT);
        APPEND(APPEND(MAKE(ENT),L1),L2)->APPEND(MAKE(ENT),APPEND(L1,L2));
        CONS(L1,ENT)->APPEND(L1,MAKE(ENT));
        DEL(NIL)->ERREUR;
        DEL(APPEND(MAKE(ENT),L1))->L1;
        DEL(MAKE(ENT))->NIL;
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          2 SECONDES
DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui


QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex8
TYPE LISTBIS
   PROFILS
      NILBIS                    :LISTBIS;
      CONSBIS(LISTBIS,ENTIER)   :LISTBIS;
      APPBIS(LISTBIS,LISTBIS)   :LISTBIS;
      MAKEBIS(ENTIER)           :LISTBIS;
      DELBIS(LISTBIS)           :LISTBIS;
   VAR
     ENT         :ENTIER;
     L1,L2       :LISTBIS;
   AXIOMES
```

Cette fois l'environnement DEMO existe

Une liste...

... on remarque que CONS ajoute "à droite"

```
    MAKEBIS(ENT)                    ->CONSBIS(NILBIS,ENT);
    APPBIS(NILBIS,L1)               ->L1;
    APPBIS(CONSBIS(L1,ENT),L2)      ->CONSBIS(APPBIS(L1,L2),ENT);
    DELBIS(CONSBIS(L1,ENT))         ->L1;
    DELBIS(NILBIS)                  ->ERREUR;
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          1 SECONDES
DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          1 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?non


QUE VOULEZ-VOUS FAIRE ?stop

     AU REVOIR...
- END PASCAL
!
```

LISTE et LISTBIS ne sont pas p.équivalents

On va modifier sous éditeur le type LISTE.

```
!orsec
- SFER/PASCAL-SYSTEM,VERS. 1/11/79-U4
                    O R S E C    A    V O T R E    S E R V I C E

VERSION  1.2 -OCT  82
17*11*82

NOM DE L'ENVIRONNEMENT ?demo


QUE VOULEZ-VOUS FAIRE ?detruire

NOM DU TYPE A SUPPRIMER DE L'ENVIRONNEMENT ?liste

TYPE DETRUIT

QUE VOULEZ-VOUS FAIRE ?comaprer


                    LISTE DES COMMANDES DISPONIBLES


  CATALOG  : PERMET D 'OBTENIR LA LISTE DES TYPES ET DES FONCTIONS
             PRESENTS DANS L'ENVIRONNEMENT.
             CHAQUE TYPE EST SUIVI DE SA LISTE D'IMPORTATION,
             CHAQUE FONCTION DE SON PROFIL.
  COMPARER : COMPARAISON D'UNE UNITE AVEC CELLES
             DE L'ENVIRONNEMENT.
  TYPE     : PERMET D'OBTENIR LE TEXTE D'UN TYPE.
  DETRUIRE : DESTRUCTION D'UNE UNITE.
  RELATION : PERMET D'OBTENIR LES TYPES UTILISES PAR OU
             UTILISANT UN TYPE DONT ON DONNERA LE NOM.
  STOP     : ARRET DE LA SESSION.
  TERMINAL : PERMET DE CHOISIR LE FICHIER DE SORTIE DES
             RESULTATS (FICHIER SUR DISQUE OU TERMINAL).


                    ******************************


QUE VOULEZ-VOUS FAIRE ?
```

Suppression du type LISTE de l'environnement.

```
                    comparer

NOM DU FICHIER SOURCE  ?ex1
TYPE LISTE
  PROFILS
  NIL:LISTE;
  MAKE(ENTIER):LISTE;
  APPEND(LISTE,LISTE):LISTE;
  CONS(LISTE,ENTIER):LISTE;
  DEL(LISTE):LISTE;
  VAR
          ENT:ENTIER;
          L1,L2:LISTE;
  AXIOMES
        APPEND(NIL,L1)->L1;
        APPEND(MAKE(ENT),NIL)->MAKE(ENT);
        APPEND(APPEND(MAKE(ENT),L1),L2)->APPEND(MAKE(ENT),APPEND(L1,L2));
        CONS(L1,ENT)->APPEND(MAKE(ENT),L1);
        DEL(NIL)->ERREUR;
        DEL(APPEND(MAKE(ENT),L1))->L1;
        DEL(MAKE(ENT))->NIL;
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          2 SECONDES
DUREE DE LA COMPARAISON:           1 SECONDES

DUREE DE LA COMPARAISON:           0 SECONDES

DUREE DE LA COMPARAISON:           0 SECONDES

DUREE DE LA COMPARAISON:           0 SECONDES

DUREE DE LA COMPARAISON:           0 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui


QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex8
TYPE LISTBIS
   PROFILS
      NILBIS                  :LISTBIS;
      CONSBIS(LISTBIS,ENTIER) :LISTBIS;
      APPBIS(LISTBIS,LISTBIS) :LISTBIS;
      MAKEBIS(ENTIER)         :LISTBIS;
      DELBIS(LISTBIS)         :LISTBIS;
   VAR
     ENT         :ENTIER;
     L1,L2       :LISTBIS;
   AXIOMES
     MAKEBIS(ENT)                      ->CONSBIS(NILBIS,ENT);
     APPBIS(NILBIS,L1)                 ->L1;
     APPBIS(CONSBIS(L1,ENT),L2)        ->CONSBIS(APPBIS(L1,L2),ENT);
     DELBIS(CONSBIS(L1,ENT))           ->L1;
     DELBIS(NILBIS)                    ->ERREUR;
 FIN

  TYPE SYNTAXIQUEMENT CORRECT
  ***************************
```

présentation du type LISTE après correction

```
DUREE DE LA CONSTRUCTION DU SUPPORT:          1 SECONDES
DUREE DE LA COMPARAISON:           1 SECONDES

DUREE DE LA COMPARAISON:           0 SECONDES

DUREE DE LA COMPARAISON:           0 SECONDES

DUREE DE LA COMPARAISON:           0 SECONDES

DUREE DE LA COMPARAISON:           0 SECONDES

DUREE DE LA COMPARAISON:           3 SECONDES

          PROPOSITION
     ***************************
     *  LISTBIS   *  LISTE     *
     ***************************
     *  APPBIS    *  APPEND    *
     *  CONSBIS   *  CONS      *
     *  DELBIS    *  DEL       *
     *  NILBIS    *  NIL       *
     *  MAKEBIS   *  MAKE      *
     ***************************
     *  LISTBIS <==>LISTE   ? *
     ***************************

FAUT-IL CONSERVER LE TYPE (OUI/NON)?non
```

Cette fois, LISTE et LISTBIS sont équivalents.

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex11
TYPE CIRCULIS
     PROFILS
          CREATE                        :CIRCULIS;
          INSERT(CIRCULIS,ENTIER)       :CIRCULIS;
          DELETE(CIRCULIS)              :CIRCULIS;
          RIGHT(CIRCULIS)               :CIRCULIS;
          HEAD (CIRCULIS)               :ENTIER;
     VAR
       C,C1:CIRCULIS;
       E,E1,E2:ENTIER;
     AXIOMES
        DELETE(CREATE)                  ->ERREUR;
        DELETE(INSERT(CREATE,E))            ->CREATE;
        DELETE(INSERT(INSERT(C,E1),E2)) -> INSERT(DELETE(INSERT(C,E1)),E2);
        RIGHT(CREATE)                   ->CREATE;
        RIGHT(INSERT(CREATE,E))         ->INSERT(CREATE,E);
        RIGHT(INSERT(INSERT(C,E1),E)) ->INSERT(RIGHT(INSERT(C,E)),E1);
        HEAD(CREATE)                    -> ERREUR;
        HEAD(INSERT(CREATE,E))          -> E;
        HEAD(INSERT(INSERT(C,E1),E2)) -> HEAD(INSERT(C,E1));
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

 DUREE DE LA CONSTRUCTION DU SUPPORT:          0 SECONDES
 DUREE DE LA COMPARAISON:           0 SECONDES

 DUREE DE LA COMPARAISON:           0 SECONDES

 DUREE DE LA COMPARAISON:           0 SECONDES

 DUREE DE LA COMPARAISON:           0 SECONDES


          PROPOSITION
       ***************************
       *  CIRCULIS  *  FILE      *
       ***************************
       *  INSERT    *  ENFILER   *
       *  DELETE    *  DEFILER   *
       *  HEAD      *  PREMIER   *
       *  CREATE    *  FILEVIDE  *
       ***************************
       *  CIRCULIS  > FILE    ? *
       ***************************

 DUREE DE LA COMPARAISON:           0 SECONDES

 DUREE DE LA COMPARAISON:           0 SECONDES

 FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui
```

Une liste circulaire...

... dans laquelle on retrouve une file

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex25
TYPE LISTEJL1
  PROFILS
    NULL1       :LISTEJL1;
    AJOU1(LISTEJL1,ENTIER):LISTEJL1;
    INS1(LISTEJL1,ENTIER):LISTEJL1;
    INSG1(LISTEJL1,ENTIER):LISTEJL1;
    VOIRG1(LISTEJL1) :ENTIER;
    VOIR1(LISTEJL1)   :ENTIER;
    ESTNULL1(LISTEJL1) :BOOLEEN;
    ENL1(LISTEJL1) :LISTEJL1;
    ENLG1(LISTEJL1) :LISTEJL1;
  VAR
    LP :LISTEJL1;
    X,Y :ENTIER;
  AXIOMES
    AJOU1(INS1(LP,X),Y) -> INS1(AJOU1(LP,Y),X);
    INSG1(NULL1,X)      -> AJOU1(NULL1,X);
    INSG1(INS1(LP,Y),X) -> INS1(INSG1(LP,X),Y);
    INSG1(AJOU1(LP,Y),X)-> AJOU1(INSG1(LP,X),Y);
    VOIR1(NULL1)        -> ERREUR;
    VOIR(INS1(LP,X))    -> X;
       *
 * FONCTION <VOIR    > INEXISTANTE
    VOIR(AJOU1(LP,X))   -> VOIR1(LP);
       *
 * FONCTION <VOIR    > INEXISTANTE
    VOIRG1(NULL1)       -> ERREUR;
    VOIRG1(INS1(LP,X))  -> VOIRG1(LP);
    VOIRG1(AJOU1(LP,X)) -> SI ESTNULL1(LP) ALORS X
                                           SINON VOIRG1(LP);

    ESTNULL1(NULL1)     -> VRAI;
    ESTNULL1(AJOU1(LP,X))->FAUX;
    ESTNULL1(INS1(LP,X))->FAUX;
    ENL1(NULL1)         -> ERREUR;
    ENL1(INS1(LP,X))    -> LP;
    ENL1(AJOU1(LP,X))   -> AJOU1(ENL1(LP),X);
    ENLG1(NULL1)        -> ERREUR;
    ENLG1(INS1(LP,X))   -> INS1(ENLG1(LP),X);
    ENLG1(AJOU1(LP,X))  -> SI ESTNULL1(LP) ALORS LP
                                           SINON AJOU1(ENLG1(LP),X);
FIN

TYPE SYNTAXIQUEMENT ERRONE
**************************

QUE VOULEZ-VOUS FAIRE ?stop


     AU REVOIR...
- END PASCAL
!edit
V1503 D60P00 00/000/102
```

Corrections sous éditeur

```
!orsec
- SFER/PASCAL-SYSTEM,VERS. 1/11/79-04
                    O R S E C   A   V O T R E   S E R V I C E

VERSION  1.2 -OCT  82
17*11*82

NOM DE L'ENVIRONNEMENT ?demo


QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex25
TYPE LISTEJL1
  PROFILS
    NULL1       :LISTEJL1;
    AJOU1(LISTEJL1,ENTIER):LISTEJL1;
    INS1(LISTEJL1,ENTIER):LISTEJL1;
    INSG1(LISTEJL1,ENTIER):LISTEJL1;
    VOIRG1(LISTEJL1) :ENTIER;
    VOIR1(LISTEJL1)  :ENTIER;
    ESTNULL1(LISTEJL1) :BOOLEEN;
    ENL1(LISTEJL1) :LISTEJL1;
    ENLG1(LISTEJL1) :LISTEJL1;
  VAR
    LP :LISTEJL1;
    X,Y :ENTIER;
  AXIOMES
    AJOU1(INS1(LP,X),Y) -> INS1(AJOU1(LP,Y),X);
    INSG1(NULL1,X)      -> AJOU1(NULL1,X);
    INSG1(INS1(LP,Y),X) -> INS1(INSG1(LP,X),Y);
    INSG1(AJOU1(LP,Y),X)-> AJOU1(INSG1(LP,X),Y);
    VOIR1(NULL1)        -> ERREUR;
    VOIR1(INS1(LP,X))   -> X;
    VOIR1(AJOU1(LP,X))  -> VOIR1(LP);
    VOIRG1(NULL1)       -> ERREUR;
    VOIRG1(INS1(LP,X))  -> VOIRG1(LP);
    VOIRG1(AJOU1(LP,X)) -> SI ESTNULL1(LP) ALORS X
                           SINON VOIRG1(LP);

    ESTNULL1(NULL1)     -> VRAI;
    ESTNULL1(AJOU1(LP,X))->FAUX;
    ESTNULL1(INS1(LP,X))->FAUX;
    ENL1(NULL1)         -> ERREUR;
    ENL1(INS1(LP,X))    -> LP;
    ENL1(AJOU1(LP,X))   -> AJOU1(ENL1(LP),X);
    ENLG1(NULL1)        -> ERREUR;
    ENLG1(INS1(LP,X))   -> INS1(ENLG1(LP),X);
    ENLG1(AJOU1(LP,X))  -> SI ESTNULL1(LP) ALORS LP
                           SINON AJOU1(ENLG1(LP),X);
  FIN

  TYPE SYNTAXIQUEMENT CORRECT
  **************************

DUREE DE LA CONSTRUCTION DU SUPPORT:        20 SECONDES
DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:         11 SECONDES

          PROPOSITION
     **************************
     *  LISTEJL1  *  FILE     *
     **************************
     *  AJOU1     *  ENFILER  *
     *  ENLG1     *  DEFILER  *
     *  VOIRG1    *  PREMIER  *
     *  NULL1     *  FILEVIDE *
     **************************
     *  LISTEJL1  > FILE    ? *
     **************************

DUREE DE LA COMPARAISON:          12 SECONDES

          PROPOSITION
     **************************
     *  LISTEJL1  *  PILE     *
     **************************
     *  INS1      *  EMPILER  *
     *  ENL1      *  DEPILER  *
     *  VOIR1     *  SOMMET   *
     *  NULL1     *  VIDE     *
     **************************
     *  LISTEJL1  > PILE    ? *
     **************************

          PROPOSITION
     **************************
     *  LISTEJL1  *  PILE     *
     **************************
     *  INSG1     *  EMPILER  *
     *  ENLG1     *  DEPILER  *
     *  VOIRG1    *  SOMMET   *
     *  NULL1     *  VIDE     *
     **************************
     *  LISTEJL1  > PILE    ? *
     **************************

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          2 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui
```

après correction

```
QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex26
TYPE LISTEJL2
  PROFILS
    NULL2                    :LISTEJL2;
    AJOU2(LISTEJL2,ENTIER):LISTEJL2;
    INS2(LISTEJL2,ENTIER) :LISTEJL2;
    INSG2(LISTEJL2,ENTIER):LISTEJL2;
    VOIR2(LISTEJL2)        :ENTIER;
    VOIRG2(LISTEJL2)       :ENTIER;
    ESTNULL2(LISTEJL2)     :BOOLEEN;
    ENLG2(LISTEJL2)        :LISTEJL2;
    ENL2(LISTEJL2)         :LISTEJL2;
  VAR
    LP :LISTEJL2;
    X,Y :ENTIER;
  AXIOMES
    INSG2(INS2(LP,Y),X)          -> INS2(INSG2(LP,X),Y);
    AJOU2(NULL2,X)               -> INSG2(NULL2,X);
    AJOU2(INSG2(LP,X),Y)         -> INSG2(AJOU2(LP,Y),X);
    AJOU2(INS2(LP,X),Y)          -> INS2(AJOU2(LP,Y),X);
    ESTNULL2(NULL2)              -> VRAI;
    ESTNULL2(INSG2(LP,X))        -> FAUX;
    ESTNULL2(INS2(LP,X))          -> FAUX;
    ENL2(NULL2)                  -> ERREUR;
    ENL2(INS2(LP,X))             -> LP;
    ENL2(INSG2(LP,X))            ->INSG2(ENL2(LP),X);
    ENLG2(NULL2)                 -> ERREUR;
    ENLG2(INS2(LP,X))            -> INS2(ENLG2(LP),X);
    ENLG2(INSG2(LP,X))           -> LP;
    VOIR2(NULL2)                 -> ERREUR;
    VOIR2(INS2(LP,X))            -> X;
    VOIR2(INSG2(LP,X))           -> VOIR2(LP);
    VOIRG2(NULL2)                -> ERREUR;
    VOIRG2(INS2(LP,X))           -> VOIRG2(LP);
    VOIRG2(INSG2(LP,X))          -> X;
  FIN

  TYPE SYNTAXIQUEMENT CORRECT
  ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:        19 SECONDES
DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:         12 SECONDES

          PROPOSITION
      ***************************
      *  LISTEJL2  *  FILE      *
      ***************************
      *  AJOU2     *  ENFILER   *
      *  ENLG2     *  DEFILER   *
      *  VOIRG2    *  PREMIER   *
      *  NULL2     *  FILEVIDE  *
      ***************************
      *  LISTEJL2   > FILE    ? *
      ***************************
```

Une autre présentation ...

```
      ***************************

DUREE DE LA COMPARAISON:         13 SECONDES

          PROPOSITION
      ***************************
      *  LISTEJL2  *  PILE      *
      ***************************
      *  INSG2     *  EMPILER   *
      *  ENLG2     *  DEPILER   *
      *  VOIRG2    *  SOMMET    *
      *  NULL2     *  VIDE      *
      ***************************
      *  LISTEJL2   > PILE    ? *
      ***************************


          PROPOSITION
      ***************************
      *  LISTEJL2  *  PILL      *
      ***************************
      *  INS2      *  EMPILER   *
      *  ENL2      *  DEPILER   *
      *  VOIR2     *  SOMMET    *
      *  NULL2     *  VIDE      *
      ***************************
      *  LISTEJL2   > PILE    ? *
      ***************************

DUREE DE LA COMPARAISON:          0 SECONDES

DUREE DE LA COMPARAISON:          2 SECONDES

DUREE DE LA COMPARAISON:        133 SECONDES

          PROPOSITION
      ***************************
      *  LISTEJL2  *  LISTEJL1  *
      ***************************
      *  AJOU2     *  AJOU1     *
      *  INS2      *  INS1      *
      *  INSG2     *  INSG1     *
      *  VOIR2     *  VOIR1     *
      *  VOIRG2    *  VOIRG1    *
      *  ESTNULL2  *  ESTNULL1  *
      *  ENLG2     *  ENLG1     *
      *  ENL2      *  ENL1      *
      *  NULL2     *  NULL1     *
      ***************************
      *  LISTEJL2 <==>LISTEJL1? *
      ***************************

FAUT-IL CONSERVER LE TYPE (OUI/NON)?non

QUE VOULEZ-VOUS FAIRE ?catalog
```

... équivalente à celle de LISTEJL1

état de l'environnement après toutes ces manipulations

```
LISTE DES TYPES ET DES OPERATIONS

 TYPE  ENTIER    <>
 TYPE  BOOLEEN   <>
 FONCTION  SI      :?
 FONCTION  PRE     :BOOLEEN
 FONCTION  &       (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  DIV     (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  =       (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  ET      (BOOLEEN ,BOOLEEN ):BOOLEEN
 FONCTION  <=      (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  <       (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  MOD     (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  -       (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  *       (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  NON     (BOOLEEN ):BOOLEEN
 FONCTION  OU      (BOOLEEN ,BOOLEEN ):BOOLEEN
 FONCTION  +       (ENTIER  ,ENTIER  ):ENTIER
 FONCTION  >=      (ENTIER  ,ENTIER  ):BOOLEEN
 FONCTION  >       (ENTIER  ,ENTIER  ):BOOLEEN
 TYPE  BOOLEAN   <ENTIER  ,BOOLEEN ,BOOLEAN >
 FONCTION  TRUE    :BOOLEAN
 FONCTION  FALSE   :BOOLEAN
 FONCTION  NOT     (BOOLEAN ):BOOLEAN
 FONCTION  TAUT    (BOOLEAN ):BOOLEAN
 FONCTION  CONTRA  (BOOLEAN ):BOOLEAN
 FONCTION  AND     (BOOLEAN ,BOOLEAN ):BOOLEAN
 FONCTION  OR      (BOOLEAN ,BOOLEAN ):BOOLEAN
 FONCTION  XOR     (BOOLEAN ,BOOLEAN ):BOOLEAN
 FONCTION  NOR     (BOOLEAN ,BOOLEAN ):BOOLEAN
 FONCTION  IF      (BOOLEAN ,BOOLEAN ):BOOLEAN
 FONCTION  IFF     (BOOLEAN ,BOOLEAN ):BOOLEAN
 FONCTION  NAND    (BOOLEAN ,BOOLEAN ):BOOLEAN
 TYPE  INT1      <ENTIER  ,BOOLEEN ,INT1    >
 FONCTION  ZERO1   :INT1
 FONCTION  SUCC1   (INT1    ):INT1
 FONCTION  PRED1   (INT1    ):INT1
 FONCTION  OPP1    (INT1    ):INT1
 FONCTION  ADD1    (INT1    ,INT1    ):INT1
 TYPE  NAT       <ENTIER  ,BOOLEEN ,NAT     >
 FONCTION  ZNAT    :NAT
 FONCTION  SNAT    (NAT     ):NAT
 FONCTION  PNAT    (NAT     ):NAT
 FONCTION  ADDNAT  (NAT     ,NAT     ):NAT
 FONCTION  MOINAT  (NAT     ,NAT     ):NAT
 FONCTION  INFNAT  (NAT     ,NAT     ):BOOLLEN
 TYPE  FILE      <ENTIER  ,BOOLEEN ,FILE    >
 FONCTION  FILEVIDE:FILE
 FONCTION  ENFILER (FILE    ,ENTIER  ):FILE
 FONCTION  DEFILER (FILE    ):FILE
 FONCTION  PREMIER (FILE    ):ENTIER
 TYPE  PILE      <ENTIER  ,BOOLLEN ,PILE    >
 FONCTION  VIDE    :PILE
 FONCTION  EMPILER (PILE    ,ENTIER  ):PILE
 FONCTION  DEPILER (PILE    ):PILE
 FONCTION  SOMMET  (PILE    ):ENTIER
 TYPE  LISTE     <ENTIER  ,BOOLEEN ,LISTE   >
 FONCTION  NIL     :LISTE
 FONCTION  MAKE    (ENTIER  ):LISTE
 FONCTION  APPEND  (LISTE   ,LISTE   ):LISTE
 FONCTION  CONS    (LISTE   ,ENTIER  ):LISTE
 FONCTION  DEL     (LISTE   ):LISTE
 TYPE  CIRCULIS  <ENTIER  ,BOOLEEN ,CIRCULIS>
 FONCTION  CREATE  :CIRCULIS
 FONCTION  INSERT  (CIRCULIS,ENTIER  ):CIRCULIS
 FONCTION  DELETE  (CIRCULIS):CIRCULIS
 FONCTION  RIGHT   (CIRCULIS):CIRCULIS
 FONCTION  HEAD    (CIRCULIS):ENTIER
 TYPE  LISTEJL1  <ENTIER  ,BOOLLEN ,LISTEJL1>
 FONCTION  NULL1   :LISTEJL1
 FONCTION  AJOU1   (LISTEJL1,ENTIER  ):LISTEJL1
 FONCTION  INS1    (LISTEJL1,ENTIER  ):LISTEJL1
 FONCTION  INSG1   (LISTEJL1,ENTIER  ):LISTEJL1
 FONCTION  VOIRG1  (LISTEJL1):ENTIER
 FONCTION  VOIR1   (LISTEJL1):ENTIER
 FONCTION  ESTNULL1(LISTEJL1):BOOLEEN
 FONCTION  ENL1    (LISTEJL1):LISTEJL1
 FONCTION  ENLG1   (LISTEJL1):LISTEJL1


FIN DE LISTE
QUE VOULEZ-VOUS FAIRE ?
```

comparer

```
NOM DU FICHIER SOURCE  ?ex27
TYPE CHAINE
  PROFILS
    CHVIDE                          :CHAINE;
    CHAJD (CHAINE,ENTIER)           :CHAINE;
    CHAJG(CHAINE,ENTIER)            :CHAINE;
    CHEND(CHAINE)                   :CHAINE;
    CHENG(CHAINE)                   :CHAINE;
    CHESTVID(CHAINE)                :BOULEEN;
    CHVOIRG(CHAINE)                 :ENTIER;
    CHVOIRD(CHAINE)                 :ENTIER;
    CHD(CHAINE)                     :CHAINE;
    CHG(CHAINE)                     :CHAINE;
  VAR
    CH :CHAINE;
    X,Y :ENTIER;
  AXIOMES
    CHAJG(CHVIDE,X)          -> CHAJD(CHVIDE,X);
    CHAJG(CHAJD(CH,X),Y)       -> CHAJD(CHAJG(CH,Y),X);
    CHENG(CHVIDE)            -> ERREUR;
    CHENG(CHAJD(CH,X))       -> SI CHESTVID(CH) ALORS CH
                                                SINON CHAJD(CHENG(CH),X);

    CHEND(CHVIDE)             -> ERREUR;
    CHEND(CHAJD(CH,X))        -> CH;
    CHESTVID(CHVIDE)         -> VRAI;
    CHESTVID(CHAJD(CH,X))    -> FAUX;
    CHVOIRD(CHVIDE)           -> ERREUR;
    CHVOIRD(CHAJD(CH,X))      -> X;
    CHVOIRG(CHVIDE)           -> ERREUR;
    CHVOIRG(CHAJD(CH,X))      -> SI CHESTVID(CH) ALORS X
                                                 SINON CHVOIRG(CH);

    CHD(CHVIDE)               -> CHVIDE;
    CHD(CHAJD(CH,X))          -> CHAJG(CH,X);
    CHG(CHVIDE)               -> CHVIDE;
    CHG(CHAJD(CH,X))          -> CHAJD(CHAJD(CHENG(CH),X),CHVOIRG(CH));
FIN
```

Un type où on peut ajouter un élément à droite, à gauche, en supprimer à droite ou à gauche, voir l'élément à droite ou à gauche, faire passer l'élément de droite à gauche et réciproquement...

TYPE SYNTAXIQUEMENT CORRECT

DUREE DE LA CONSTRUCTION DU SUPPORT: 6 SECONDES
DUREE DE LA COMPARAISON: 0 SECONDES

DUREE DE LA COMPARAISON: 5 SECONDES

DUREE DE LA COMPARAISON: 2 SECONDES

DUREE DE LA COMPARAISON: 9 SECONDES

PROPOSITION

CHAINE	FILE
CHAJG	ENFILER
CHEND	DEFILER
CHVOIRD	PREMIER
CHVIDE	FILEVIDE
CHAINE > FILE ?	

CHAINE	FILE
CHAJD	ENFILER
CHENG	DEFILER
CHVOIRG	PREMIER
CHVIDE	FILEVIDE
CHAINE > FILE ?	

DUREE DE LA COMPARAISON: 9 SECONDES

PROPOSITION

CHAINE	PILE
CHAJD	EMPILER
CHEND	DEPILER
CHVOIRD	SOMMET
CHVIDE	VIDE
CHAINE > PILE ?	

PROPOSITION

CHAINE	PILE
CHAJG	EMPILER
CHENG	DEPILER
CHVOIRG	SOMMET
CHVIDE	VIDE
CHAINE > PILE ?	

DUREE DE LA COMPARAISON: 0 SECONDES

DUREE DE LA COMPARAISON: 17 SECONDES

PROPOSITION

CHAINE	CIRCULIS
CHAJG	INSERT
CHEND	DELETE
CHD	RIGHT
CHVOIRD	HEAD
CHVIDE	CREATE
CHAINE > CIRCULIS?	

PROPOSITION

CHAINE	CIRCULIS
CHAJD	INSERT
CHENG	DELETE
CHD	RIGHT

```
    * CHVOIRG    * HEAD     *
    * CHVIDE     * CREATE   *
    ***************************
    * CHAINE     > CIRCULIS? *
    ***************************

DUREE DE LA COMPARAISON:          1 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui


QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex24
TYPE IMPLCIRC
  PROFILS
    IMPLVIDE : IMPLCIRC;
    IMPLAJ (IMPLCIRC,ENTIER) : IMPLCIRC;
    IMPLROT (IMPLCIRC) : IMPLCIRC;
    IMPLTETE (IMPLCIRC) :ENTIER;
    IMPLENL (IMPLCIRC) : IMPLCIRC;
    REP  (CHAINE) : IMPLCIRC;
  VAR
    T :CHAINE;
    X1,X2 :ENTIER;
  AXIOMES
    IMPLVIDE  -> REP(CHVIDE);
    IMPLAJ(REP(T),X1) -> REP(CHAJD(T,X1));
    IMPLTETE(REP(T)) -> CHVOIRG(T);
    IMPLENL(REP(T)) -> SI CHESTVID(T) ALORS ERREUR
                                      SINON REP(CHENG(T));
    IMPLROT(REP(T)) -> SI CHESTVID(T) ALORS REP(T)
                       SINON REP(CHAJG(CHEND(T),CHVOIRD(T)));
FIN


 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:        15 SECONDES
DUREE DE LA COMPARAISON:          2 SECONDES

DUREE DE LA COMPARAISON:          2 SECONDES

DUREE DE LA COMPARAISON:          2 SECONDES

DUREE DE LA COMPARAISON:          3 SECONDES


          PROPOSITION
     ***************************
     *  IMPLCIRC  *  FILE      *
     ***************************
     *  IMPLAJ    *  ENFILER   *
     *  IMPLENL   *  DEFILER   *
     *  IMPLTETE  *  PREMIER   *
     *  IMPLVIDE  *  FILEVIDE  *
     ***************************
     *  IMPLCIRC   > FILE    ? *
     ***************************

DUREE DE LA COMPARAISON:          2 SECONDES

DUREE DE LA COMPARAISON:          2 SECONDES
```

On utilise le type CHAINE pour construire ...

```
DUREE DE LA COMPARAISON:          4 SECONDES

          PROPOSITION
     ***************************
     *  IMPLCIRC  *  CIRCULIS  *
     ***************************
     *  IMPLAJ    *  INSERT    *
     *  IMPLENL   *  DELETE    *
     *  IMPLROT   *  RIGHT     *
     *  IMPLTETE  *  HEAD      *
     *  IMPLVIDE  *  CREATE    *
     ***************************
     *  IMPLCIRC   > CIRCULIS? *
     ***************************

DUREE DE LA COMPARAISON:          1 SECONDES

DUREE DE LA COMPARAISON:          3 SECONDES

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui
```

... une liste circulaire.

```
QUE VOULEZ-VOUS FAIRE ?detruire

NOM DU TYPE A SUPPRIMER DE L'ENVIRONNEMENT ?circulis

TYPE DETRUIT

QUE VOULEZ-VOUS FAIRE ?comparer

NOM DU FICHIER SOURCE  ?ex11
TYPE CIRCULIS
     PROFILS                                  :CIRCULIS;
          CREATE                              :CIRCULIS;
          INSERT(CIRCULIS,ENTIER)             :CIRCULIS;
          DELETE(CIRCULIS)                    :CIRCULIS;
          RIGHT(CIRCULIS)                     :CIRCULIS;
          HEAD (CIRCULIS)                     :ENTIER;
     VAR
       C,C1:CIRCULIS;
       E,E1,E2:ENTIER;
     AXIOMES
        DELETE(CREATE)                        ->ERREUR;
        DELETE(INSERT(CREATE,E))                  ->CREATE;
        DELETE(INSERT(INSERT(C,E1),E2)) -> INSERT(DELETE(INSERT(C,E1)),E2);
        RIGHT(CREATE)                         ->CREATE;
        RIGHT(INSERT(CREATE,E))               ->INSERT(CREATE,E);
        RIGHT(INSERT(INSERT(C,E1),E)) ->INSERT(RIGHT(INSERT(C,E)),E1);
        HEAD(CREATE)                          -> ERREUR;
        HEAD(INSERT(CREATE,E))                -> E;
        HEAD(INSERT(INSERT(C,E1),E2)) -> HEAD(INSERT(C,E1));
FIN

 TYPE SYNTAXIQUEMENT CORRECT
 ***************************

DUREE DE LA CONSTRUCTION DU SUPPORT:          1 SECONDES
DUREE DE LA COMPARAISON:            0 SECONDES

DUREE DE LA COMPARAISON:            0 SECONDES

DUREE DE LA COMPARAISON:            0 SECONDES

DUREE DE LA COMPARAISON:            0 SECONDES

          PROPOSITION
      ***************************
      *  CIRCULIS  *  FILE      *
      ***************************
      *  INSERT    *  ENFILER   *
      *  DELETE    *  DEFILER   *
      *  HEAD      *  PREMIER   *
      *  CREATE    *  FILEVIDE  *
      ***************************
      *  CIRCULIS   > FILE    ? *
      ***************************

DUREE DE LA COMPARAISON:            0 SECONDES

DUREE DE LA COMPARAISON:            0 SECONDES

DUREE DE LA COMPARAISON:            2 SECONDES
```

On supprime CIRCULIS ...

... et on le remet à nouveau au système ...

... pour montrer qu'on retrouve les mêmes relations.

```
DUREE DE LA COMPARAISON:           10 SECONDES

          PROPOSITION
      ***************************
      *  CIRCULIS  *  CHAINE    *
      ***************************
      *  INSERT    *  CHAJC     *
      *  DELETE    *  CHENC     *
      *  RIGHT     *  CHD       *
      *  HEAD      *  CIVOIRC   *
      *  CREATE    *  CIVIDE    *
      ***************************
      *  CIRCULIS   < CHAINE  ? *
      ***************************


          PROPOSITION
      ***************************
      *  CIRCULIS  *  CHAINE    *
      ***************************
      *  INSERT    *  CHAJD     *
      *  DELETE    *  CHENG     *
      *  RIGHT     *  CHD       *
      *  HEAD      *  CIVOIRG   *
      *  CREATE    *  CIVIDE    *
      ***************************
      *  CIRCULIS   < CHAINE  ? *
      ***************************

DUREE DE LA COMPARAISON:            4 SECONDES


          PROPOSITION
      ***************************
      *  CIRCULIS  *  IMPLCIRC  *
      ***************************
      *  INSERT    *  IMPLAJ    *
      *  DELETE    *  IMPLENL   *
      *  RIGHT     *  IMPLROT   *
      *  HEAD      *  IMPLTETE  *
      *  CREATE    *  IMPLVIDE  *
      ***************************
      *  CIRCULIS   < IMPLCIRC? *
      ***************************

FAUT-IL CONSERVER LE TYPE (OUI/NON)?oui
```

```
QUE VOULEZ-VOUS FAIRE ?detruire

NOM DU TYPE A SUPPRIMER DE L'ENVIRONNEMENT ?chaine

TYPE UTILISE DANS D'AUTRES SPECIFICATIONS

QUE VOULEZ-VOUS FAIRE ?relation

POUR LA FERMETURE TRANSITIVE DE "UTILISE" , TAPEZ "U",
POUR SON INVERSE , TAPEZ "I" ?u

NOM DU TYPE ?implcirc

LISTE DES TYPES UTILISES PAR IMPLCIRC
ENTIER
BOOLEEN
CHAINE
FIN DE LISTE

QUE VOULEZ-VOUS FAIRE ?relation

POUR LA FERMETURE TRANSITIVE DE "UTILISE" , TAPEZ "U",
POUR SON INVERSE , TAPEZ "I" ?i

NOM DU TYPE ?chaine

LISTE DES TYPES UTILISANT CHAINE
IMPLCIRC
FIN DE LISTE

QUE VOULEZ-VOUS FAIRE ?detruire

NOM DU TYPE A SUPPRIMER DE L'ENVIRONNEMENT ?implcirc

TYPE DETRUIT

QUE VOULEZ-VOUS FAIRE ?stop


    AU REVOIR...
- END PASCAL
!1
SAVE DEMO----LCHAINE  /0001 (Y/N)? y
SAVE DEMO----RCHAINE  /0001 (Y/N)? y
SAVE DEMO----GCHAINE  /0001 (Y/N)? y
SAVE EX24             /0001 (Y/N)? y
SAVE DEMO----LCIRCULIS/0001 (Y/N)? y
SAVE DEMO----RCIRCULIS/0001 (Y/N)? y
SAVE DEMO----GCIRCULIS/0001 (Y/N)? y
LOGOUT DONE AT  15*19*33

?? IUCASSTS    IS DISCONNECTED ON DAY:0322,HOUR:0015,MIN:0019
```

On ne peut détruire le type CHAINE car il est utilisé dans d'autres précentations, le qui est mis en evidence par la commande RELATION

NOM DE L'ETUDIANT : PROCH Karol

NATURE DE LA THESE : Doctorat 3ème cycle Informatique

VU, APPROUVE

ET PERMIS D'IMPRIMER

NANCY, le [illegible] 1982 3925

LE PRESIDENT DE L'UNIVERSITE DE NANCY I,

UNIVERSITE NANCY I
LE PRESIDENT
R. MAINARD