Les Types génériques : propositions pour un mécanisme d'abstraction dans les langages de programmation

Paul Jacquet

THESE

présentée à

Université Scientifique et Médicale de Grenoble

Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR DE 3ème CYCLE

Informatique

par

Paul JACQUET

LES TYPES GENERIQUES

PROPOSITIONS POUR UN MECANISME D'ABSTRACTION DANS LES LANGAGES DE PROGRAMMATION.

Thèse soutenue le 25 septembre 1978 devant la Commission d'Examen :

Président : L. BOLLIET

Examinateurs : J.C. BOUSSARD
P. JORRAND
G. VEILLON

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Monsieur Gabriel CAU : Président
Monsieur Joseph KLEIN : Vice-Président

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

PROFESSEURS TITULAIRES

MM.	AMBLARD Pierre	Clinique de dermatologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	I.S.N.
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale A
	BEAUDOING André	Clinique de pédiatrie et puériculture
	BELORIZKY Elie	Physique
	BARNARD Alain	Mathématiques pures
Mme	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZES Henri	Clinique chirurgicale et traumatologie
	BLAMBERT Maurice	Mathématiques pures
	BOLLIET Louis	Informatique (I.U.T. B)
	BONNET Jean-Louis	Clinique ophtalmologie
	BONNET-EYMARD Joseph	Clinique hépato-gastro-entérologie
Mme	BONNIER Marie-Jeanne	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BOUTET DE MONVEL Louis	Mathématiques pures
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie

.../...

MM.	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques pures
	CHARACHON Robert	Clinique ot-rhino-laryngologique
	CHATEAU Robert	Clinique de neurologie
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	COUDERC Pierre	Anatomie pathologique
	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumophtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DODU Jacques	Mécanique appliquée (I.U.T. I)
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	FONTAINE Jean-Marc	Mathématiques pures
	GAGNAIRE Didier	Chimie physique
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Analyse numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique générale
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
Mme	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LE ROY Philippe	Mécanique (I.U.T. I)

.../...

MM.	LLIBOUTRY Louis	Géophysique
	LOISEAUX Jean-Marie	Sciences nucléaires
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mlle	LUTZ Elisabeth	Mathématiques pures
MM.	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Clinique cardiologique
	MAYNARD Roger	Physique du solide
	MAZARE Yves	Clinique Médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEGRE Robert	Mécanique
	NOZIERES Philippe	Spectrométrie physique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET Jean	Séméiologie médicale (neurologie)
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REVOL Michel	Urologie
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-Chirurgie
	SARRAZIN Roger	Clinique chirurgicale B
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique (I.U.T. I)
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique biophysique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale

.../...

PROFESSEURS ASSOCIES

MM.	CRABBE Pierre	CERMO
	SUNIER Jules	Physique

PROFESSEURS SANS CHAIRE

Mlle	AGNIUS-DELORS Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Gilbert	Géographie
	BENZAKEN Claude	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique (I.U.T. I)
	BUISSON René	Physique (I.U.T. I)
	BUTEL Jean	Orthopédie
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique (I.U.T. I)
	DELOBEL Claude	M.I.A.G.
	DEPASSEL Roger	Mécanique des fluides
	GAUTRON René	Chimie
	GIDON Paul	Géologie et minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biochimie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et médecine préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme	KAHANE Josette	Physique
MM.	KRAKOWIACK Sacha	Mathématiques appliquées
	KUHN Gerard	Physique (I.U.T. I)
	LUU DUC Cuong	Chimie organique - pharmacie
	MICHOULIER Jean	Physique (I.U.T. I)
Mme	MINIER Colette	Physique (I.U.T. I)

.../...

MM.	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
MM.	STIEGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	ARMAND Yves	Chimie (I.U.T. I)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro-chirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (I.U.T. B) (Personne étrangère habilitée à être directeur de thèse)
	BERNARD Pierre	Gynécologie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COLIN DE VERDIERE Yves	Mathématiques pures
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	CORDONNER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie

.../...

MM.	CYROT Michel	Physique du solide
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FAURE Gilbert	Urologie
	GAUTIER Robert	Chirurgie générale
	GIDON Maurice	Géologie
	GROS Yves	Physique (I.U.T. I)
	GUIGNIER Michel	Thérapeutique
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	JALBERT Pierre	Histologie
	JUNIEN-LAVILLAVROY Claude	O.R.L.
	KOLODIE Lucien	Hématologie
	LE NOC Pierre	Bactériologie-virologie
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MALLION Jean-Michel	Médecine du travail
	MARECHAL Jean	Mécanique (I.U.T. I)
	MARTIN-BOUYER Michel	Chimie (CUS)
	MASSOT Christian	Médecine interne
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (I.U.T. I)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	PEFFEN René	Métallurgie (I.U.T. I)
	PERRIER Guy	Géophysique-glaciologie
	PHELIP Xavier	Rhumatologie
	RACHALL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
Mme	RENAUDET Jacqueline	Bactériologie (pharmacie)
MM.	ROBERT Jean-Bernard	Chimie-physique
	ROMIER Guy	Mathématiques (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	SAKAROVITCH Michel	Mathématiques appliquées

.../...

MM.	SCHAERER René	Cancérologie
Mme	SEIGLE-MURANDI Françoise	Crytogamie
MM.	STOEBNER Pierre	Anatomie pathologie
	STUTZ Pierre	Mécanique
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	DEVINE Roderick	Spectro Physique
	KANEKO Akira	Mathématiques pures
	JOHNSON Thomas	Mathématiques appliquées
	RAY Tuhina	Physique

MAITRE DE CONFERENCES DELEGUE

M.	ROCHAT Jacques	Hygiène et hydrologie (pharmacie)

Fait à Saint Martin d'Hères, novembre 1977

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1978-1977

Président : M. Philippe TRAYNARD

Vice-présidents : M. René PAUTHENET

M. Georges LESPINARD

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Electronique - automatique
	BESSON Jean	Chimie minérale
	BLOCH Daniel	Physique du solide - cristallographie
	BONNETAIN Lucien	Génie chimique
	BONNIER Etienne	Métallurgie
*	BOUDOURIS Georges	Electronique - automatique
	BRISSONNEAU Pierre	Physique du solide - cristallographie
	BUYLE-BODIN Maurice	Electronique - automatique
	COUMES André	Electronique - automatique
	DURAND Francis	Métallurgie
	FELICI Noël	Electronique - automatique
	FOULARD Claude	Electronique - automatique
	LANCIA Roland	Electronique - automatique
	LONGEQUEUE Jean-Pierre	Physique nucléaire corpusculaire
	LESPINARD Georges	Mécanique
	MOREAU René	Mécanique
	PARIAUD Jean-Charles	Chimie - physique
	PAUTHENET René	Electronique - automatique
	PERRET René	Electronique - automatique
	POLOUJADOFF Michel	Electronique - automatique
	TRAYNARD Philippe	Chimie - physique
	VEILLON Gérard	Informatique fondamentale et appliquée

* en congé pour études

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuël	Electronique - automatique
	BOUVARD Maurice	Génie mécanique
	COHEN Joseph	Electronique - automatique
	GUYOT Pierre	Métallurgie physique
	LACOUME Jean-Louis	Electronique - automatique
	JOUBERT Jean-Claude	Physique du solide - cristallographie

.../...

MM.	ROBERT André	Chimie appliquée et des matériaux
	ROBERT François	Analyse numérique
	ZADWORNY François	Electronique - automatique

MAITRES DE CONFERENCES

MM.	ANCEAU François	Informatique fondamentale et appliquée
	CHARTIER Germain	Electronique - automatique
	CHIAVERINA Jean	Biologie, biochimie, agronomie
	IVANES Marcel	Electronique - automatique
	LESIEUR Marcel	Mécanique
	MORET Roger	Physique nucléaire - corpusculaire
	PIAU Jean-Michel	Mécanique
	PIERRARD Jean-Marie	Mécanique
	SABONNADIERE Jean-Claude	Informatique fondamentale et appliquée
Mme	SAUCIER Gabrielle	Informatique fondamentale et appliquée
M.	SOHM Jean-Claude	Chimie Physique

CHERCHEURS DU C.N.R.S. (Directeur et Maîtres de Recherche)

M.	FRUCHART Robert	Directeur de Recherche
MM.	ANSARA Ibrahim	Maître de Recherche
	BRONOEL Guy	Maître de Recherche
	CARRE René	Maître de Recherche
	DAVID René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	KLEITZ Michel	Maître de Recherche
	LANDAU Ioan-Doré	Maître de Recherche
	MATHIEU Jean-Claude	Maître de Recherche
	MERMET Jean	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Personnalités habilitées à diriger des travaux de recherche (décision du Conseil Scientifique)

E.N.S.E.E.G.

MM.	BISCONDI Michel	Ecole des Mines St. Etienne (dépt. Métallurgie)
	BOOS Jean-Yves	Ecole des Mines St. Etienne (Métallurgie)
	DRIVER Julian	Ecole des Mines St. Etienne (Métallurgie)

.../...

MM.	KOBYLANSKI André	Ecole des Mines St. Etienne (Métallurgie)
	LE COZE Jean	Ecole des Mines St. Etienne (Métallurgie)
	LESBATS Pierre	Ecole des Mines St. Etienne (Métallurgie)
	LEVY Jacques	Ecole des Mines St. Etienne (Métallurgie)
	RIEU Jean	Ecole des Mines St. Etienne (Métallurgie)
	SAINFORT	C.E.N. Grenoble (Métallurgie)
	SOUQUET	U.S.M.G.
	CAILLET Marcel	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	COULON Michel	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	GUILHOT Bernard	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	LALAUZE René	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	LANCELOT Francis	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	SARRAZIN Pierre	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	SOUSTELLE Michel	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	THEVENOT François	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	THOMAS Gérard	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	TOUZAIN Philippe	Ecole des Mines St. Etienne (Chim. Min. Ph.)
	TRAN MINH Canh	Ecole des Mines St. Etienne (Chim. Min. Ph.)

E.N.S.E.R.G.

MM.	BOREL	Centre d'études nucléaires de Grenoble
	KAMARINOS	Centre national recherche scientifique

E.N.S.E.G.P.

M.	BORNARD	Centre national recherche scientifique
Mme	CHERUY	Centre national recherche scientifique
MM.	DAVID	Centre national recherche scientifique
	DESCHIZEAUX	Centre national recherche scientifique

Je tiens à remercier Monsieur le Professeur BOLLIET, directeur du département Informatique de l'I.U.T. de Grenoble, qui a bien voulu s'intéresser à mon travail. Je suis particulièrement sensible à l'honneur qu'il m'a fait en acceptant de présider le jury.

Je remercie Monsieur Jean-Claude BOUSSARD, professeur à l'Université de Nice, d'avoir bien voulu accepter de faire partie du jury de cette thèse. Il sut me communiquer, alors qu'il enseignait à Grenoble, son intérêt pour tout ce qui touche aux langages de programmation.

Je remercie tout particulièrement Philippe JORRAND, maître de recherches au C.N.R.S., pour la confiance qu'il m'a témoignée en m'acceptant dans son équipe. L'ambiance amicale qu'il a su créer, alliée à sa compétence dans le domaine des langages de programmation, sont pour une large part responsables de l'aboutissement de ce travail.

Je remercie également Monsieur Gérard VEILLON, professeur à l'Institut National Polytechnique de Grenoble, qui a bien voulu s'intéresser à mon travail et a accepté de le juger.

Les principaux résultats de cette thèse sont le fruit d'une collaboration étroite avec Didier BERT. Il a su me faire profiter de sa grande expérience, et son appui, autant moral que technique, a toujours été précieux, qu'il en soit ici très sincèrement remercié.

Je ne voudrais pas dissocier de ces remerciements tous mes collègues du Laboratoire avec qui j'ai souvent eu de fructueuses discussions.

Madame DIAZ a dactylographié ce manuscrit avec patience et compétence, je la remercie ainsi que tout le personnel du Service de reproduction qui a assuré, dans un temps record, la réalisation matérielle de cette thèse.

Paul JACQUET

LES TYPES GENERIQUES

PROPOSITIONS POUR UN MECANISME D'ABSTRACTION DANS LES LANGAGES DE PROGRAMMATION

Chapitre I

LES TYPES

Chapitre II

LES TYPES ABSTRAITS

Chapitre III
LES TYPES ABSTRAITS GENERIQUES

CHAPITRE I

LES TYPES

I.1. INTRODUCTION

La notion de type a été introduite dans les langages de programmation pour représenter *les propriétés fondamentales associées aux ensembles de valeurs manipulées* [NAU 60]. Cette définition, bien qu'assez vague, a néanmoins l'avantage de la concision, ce chapitre va tenter d'en développer les différents aspects.

L'énoncé précédent associe à un type un ensemble de valeurs qu'il représente, cette approche discriminatoire trouve ses racines dans la théorie logique des types. Cette théorie a été élaborée par Russel [RUS 08] pour éviter la formation de certains paradoxes, c'est-à-dire des énoncés de propositions qui, quelle que soit la valeur qu'on leur attribue, conduisent à une antinomie.

L'exemple célèbre d'un tel paradoxe est le suivant:
Considérons la classe R de toutes les classes qui ne sont pas membres d'elles-mêmes, pour une classe x quelconque on peut énoncer l'équivalence suivante:

$$x \in R \equiv \neg(x \in x)$$

et donc $R \in R \equiv \neg(R \in R)$

Si bien que la proposition $R \in R$ est équivalente à son propre contraire, donc si elle est vraie alors elle est également fausse et vice versa.

Le principe de base de la théorie des types est que les notions logiques (individus, propositions, fonctions propositionnelles) sont classifiées en une hiérarchie de *types* et qu'une fonction ne peut prendre comme arguments que des notions qui la précèdent dans la hiérarchie. Un énoncé du genre $x \in x$ devient donc incorrect (les deux arguments du symbole d'appartenance étant du même type) ce qui du même coup interdit la formation du paradoxe précédent.

Ceci suggère qu'en logique la notion de type est essentiellement syntaxique, c'est-à-dire que pour toute proposition on est capable d'associer un type et un seul à toutes les notions qui la composent et de vérifier la cohérence de l'ensemble, ceci indépendamment de toute interprétation ultérieure.

L'analogie avec nos préoccupations s'arrête là, en effet le caractère dynamique d'un programme et le fait qu'il doit s'exécuter sur une machine introduisent de nouvelles contraintes que nous allons examiner dans le paragraphe suivant.

I.2. QUELQUES NOTIONS DE BASE

L'utilisation des ressources d'un ordinateur se fait presque toujours via un langage de programmation contenant les primitives nécessaires à la description des objets manipulés et des solutions élaborées pour un programme donné. La notion de type est fortement reliée au premier aspect ; son insertion dans les langages de haut niveau répond essentiellement à deux besoins. D'une part, fournir au programmeur un niveau de machine abstraite supérieur à celui de la machine réelle en ce qui concerne les objets et les opérations qui leur sont applicables. D'autre part, assurer à ce niveau d'abstraction les vérifications de cohérence dans leur utilisation. Nous allons essayer, dans ce qui suit, de fournir une justification "a contrario" de la notion de type en utilisant le langage machine qui est très pauvre relativement aux deux aspects évoqués plus haut.

Dans la mémoire d'une machine rien ne permet d'attacher, à une séquence de bits quelconque, une signification particulière. Seules certaines instructions appliquées à cette séquence permettront de la considérer comme: un nombre entier une suite de caractères, une adresse etc... En fait, le seul moyen dont dispose le programmeur, travaillant au niveau de la machine, pour décrire les objets qu'il manipule est leur représentation exprimée en terme d'unité de mémoire adressable (octet, mot, etc...).

Ainsi, s'il veut décrire une structure comportant: une chaîne de caractères (32 octets), un entier (4 octets), deux réels (8 octets) et une adresse (4 octets) il devra réserver une zone mémoire de 48 octets indépendamment de toute signification particulière (la réservation serait la même pour une séquence de 12 entiers) et toutes les références à la structure et aux éléments qui la composent se feront par une adresse et éventuellement des déplacements. Ce dernier point prend toute son importance si le programmeur veut écrire un sous-programme opérant un traitement particulier sur une telle structure. Il n'aura en fait aucun moyen de spécifier que l'adresse transmise au sous-programme doit nécessairement être celle d'une structure comportant les champs chaîne de caractère, entier, réel, réel, adresse et ce sous-programme s'exécutera quelle que soit l'adresse qu'on lui fournira en interprétant les 48 octets comme une structure.

Ceci peut être à l'origine d'erreurs, détectées à l'exécution, particulièrement difficiles à corriger.

Les difficultés précédentes résultent de la pauvreté des moyens de spécification (limités à la taille de la mémoire utilisée) du langage machine. Examinons sur un nouvel exemple les conséquences que cela entraîne quant aux possibilités de vérifications de cohérence.

La machine que nous considérons a comme plus petite unité de mémoire adressable l'octet. Nous supposons d'autre part qu'il existe une opération (câblée ou non) notée SUCC permettant d'ajouter 1 à un octet donné.

Considérons la valeur hexadécimale particulière D7 d'un octet, cette valeur peut représenter:

- l'entier 215,
- le caractère "P" en code EBCDIC,
- le code de l'opération "ou exclusif" du système 360.

Supposons maintenant que nous appliquions l'opération SUCC à cet octet, la nouvelle valeur représente:

- l'entier 216,
- le caractère "Q" en code EBCDIC,
- ce n'est la représentation d'aucun code opération existant.

Les deux premiers exemples donnent bien pour l'entier 215 son *successeur* 216 et pour le caractère "P" son *successeur* "Q" dans l'ordre alphabétique. Bien qu'on puisse toujours trouver un ordre arbitraire sur les opérations, l'application de l'opération SUCC à la représentation d'un code opération n'a pas de sens, il manque en fait un moyen de restreindre l'ensemble des opérations applicables à une représentation de code opération.

Poursuivant cet exemple, on peut appliquer deux fois de suite l'opération SUCC à l'octet D8. La nouvelle valeur DA représente l'entier 218 qui est bien le successeur du successeur de 216, cette valeur n'est, par contre plus le code d'un caractère, en particulier ce n'est pas le code de "S". On voit donc qu'on aura à redéfinir l'opération SUCC sur les représentations de caractères et que, pour un octet donné, on aura

à choisir éventuellement parmi deux interprétations possibles de cette opération.

Les deux exemples que l'on vient d'évoquer vont nous permettre d'esquisser une première approche de la notion de type, cette approche volontairement incomplète permettra de dégager les points importants que l'on retrouvera tout au long des chapitres suivants.

L'objectif de tout programme est de s'exécuter sur une machine, par là même toutes les valeurs manipulées par ce programme devront donc être représentées dans la mémoire de cette machine. Comme on l'a vu précédemment, une représentation peut être la même pour des valeurs de nature tout à fait différentes, elle ne peut donc constituer un critère permettant de distinguer les différentes sortes de valeurs. La donnée d'une représentation ne peut donc constituer,en aucun cas,la donnée d'un type au sens où on l'a vu dans l'introduction. Par contre, comme on le verra par la suite, la donnée d'un type doit être, entre autres, celle d'une représentation qui reste une propriété fondamentale attachée à l'ensemble des valeurs.

La caractéristique la plus importante d'un type est l'ensemble des fonctions applicables à ses objets ; ce sont elles qui permettent, à des degrés divers, de dégager le programmeur des contraintes de la machine. C'est à travers ces fonctions que le programmeur va manipuler les objets du type ; c'est à partir d'elles qu'il va donc élaborer ses solutions. Le niveau de machine abstraite défini par l'ensemble des fonctions est directement lié à la difficulté d'élaboration des programmes, c'est pourquoi les langages de programmation offrent des mécanismes de définition de types de plus en plus sophistiqués.

Cette facilité d'expression ne serait rien sans le mécanisme de vérification qui la sous-tend. Il est en effet essentiel de s'assurer, à la compilation, du bon usage de ces fonctions. A cette fin, il faut pouvoir répondre aux deux questions:

. la fonction est-elle définie sur les objets du type?

. si oui, quelle est son interprétation?

Ces deux questions recouvrent, comme nous le verrons dans les chapitres II et III plusieurs aspects que nous détaillerons.

Il ressort de ces quelques remarques que l'ensemble des fonctions définies sur les objets d'un type est la propriété la plus fondamentale, c'est elle qui, du point de vue de l'utilisateur, caractérise complètement le type.

I.3. LANGAGES AUX TYPES FINIS

I.3.1. Ensemble de types fini + Vérifications statiques = Algol 60, Algol W (limité au sous-ensemble équivalent à Algol 60), Fortran ...

Ces langages proposent, en ce qui concerne les valeurs, un niveau d'abstraction relativement rudimentaire. Il comprend essentiellement trois types de bases qui sont: les entiers, les réels et les booléens auxquels sont associés implicitement, d'une part une représentation et d'autre part, un ensemble d'opérateurs (arithmétiques et logiques) permettant de construire des expressions à partir d'objets simples (variables, constantes). La possibilité de structurer un ensemble de valeurs d'un même type est également fournie par le constructeur tableau qui permet de regrouper sous un même nom cet ensemble et qui fournit, par son mécanisme d'indexation, le moyen d'accéder séparément à ces valeurs.

L'association d'un type à un objet simple se fait au moyen d'une déclaration pour les variables et d'une notation pour les constantes. Contrairement à leur homologue Fortran, Algol 60 et Algol W imposent que toutes les variables soient déclarées en évitant ainsi certaines erreurs dues aux déclarations implicites qui attribuent un type à une variable en fonction de la première lettre de son identification. En ce qui concerne les constantes, la notation étant unique pour un type de base, il n'y a pas d'ambiguité possible.

L'association d'un type à une expression se fait au moyen d'une évaluation abstraite, c'est-à-dire une évaluation où les opérandes ne sont pas les valeurs d'un type (comme dans une exécution) mais les types eux-mêmes. On dispose, pour chaque opérateur, du type du résultat en fonction du type des opérandes (les types ainsi que les opérateurs étant en nombre fini et prédéfini cette association est elle-même prédéfinie).

exemple: après les déclarations

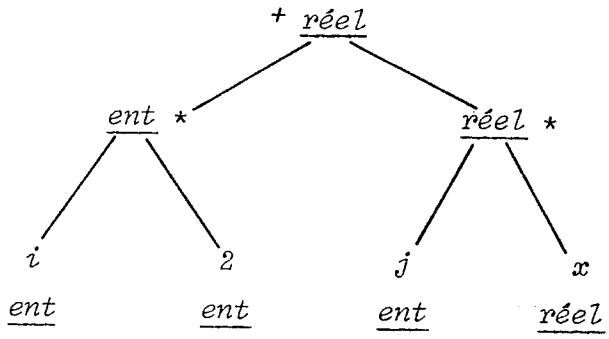
entier i,j ; réel x ;

le type de l'expression: $i*2+j*x$ sera déterminé en propageant dans l'arbre de l'expression type (connu grâce aux déclarations et aux dénotations) des feuilles, ceci au moyen des règles d'évaluation abstraite:

*ent*ent→ent*

*ent*réel→réel*

ent+réel→réel



Notons que le + entre entier et réel, délivrant un résultat réel, nécessitera la conversion de la représentation de l'entier (résultat de la multiplication de i par 2) en la représentation réelle correspondante pour pouvoir exécuter l'instruction machine correspondante (à moins évidemment que cette dernière ne soit câblée).

Toutefois, cette évaluation statique des types n'est pas toujours possible dans certains cas pathologiques ; ainsi l'expression: $i \uparrow n$ où i et n sont entiers, est de type réel si n est négatif et de type entier sinon.

Dans ce cas là,une erreur de type pourra être détectée à l'exécution si on utilise cette expression pour indexer un tableau et si n est négatif.

Les choses deviennent plus graves en ce qui concerne la correspondance entre paramètres formels et paramètres effectifs des procédures, cet aspect est fondamental pour valider les appels de procédures. En ALGOL 60 et en ALGOL W lors d'une déclaration de procédure, les paramètres ne sont pas eux-mêmes déclarés mais spécifiés (c'est-à-dire qu'il y manque certaines propriétés additionnelles comme les dimensions (en ALGOL 60), les bornes (en ALGOL W), de tableaux où la description complète (comprenant celle des paramètres) du type des procédures formelles. Ceci accroît évidemment la souplesse du langage mais au détriment des possibilités de vérifications comme nous allons le voir dans l'exemple suivant:

Cet exemple a été écrit en ALGOL W.

```
begin
  integer array a,b (1::5);
  integer l1, l2 ;

  integer procédure car (integer i) ; a(i) ;
  integer procédure cdr (integer i) ; b(i) ;
  integer procédure applique (integer procédure f, arg) ; f(arg) ;

  a(1):=30 ; a(2):=10 ; a(3):=2 ; a(4):=-1 ; a(5):=20 ;
  b(1):=0 ;  b(2):=5  ; b(3):=0 ; b(4):=3 ;  b(5):=1 ;
  l1:=2 ; l2:=4 ;

  write (applique(car,P1)) ;
  write (applique(cdr, applique(cdr,P2))) ;
  write (applique(car,cdr)) ;
  write (applique(applique,car)) ;
end.
```

On peut, sur ce petit exemple, faire un certain nombre de remarques pour le

moins curieuses. Tout d'abord,il a été compilé sans qu'aucune erreur n'ait été détectée. Or les deux premiers appels sont, syntaxiquement, incorrects (le deuxième paramètre de *applique* n'est pas du type spécifié, *integer procédure)* pour les admettre le compilateur a dû convertir la variable entière *P1* d'une par et l'appel de la procédure entière *applique* d'autre part, en une procédure entière. La première de ces conversions, bien "qu'allant de soit", n'est nulle part décrite dans le rapport de définition du langage (en l'occurrence sa validité ne tient qu'au fait que *arg* est employé sans paramètre dans le corps de *applique*). Ces appels s'exécutent tout à fait correctement et délivrent bien les valeurs attendues, 10 et 0, respectivement.

Les deux appels suivants, syntaxiquement corrects (les deux paramètres de *applique* sont dans les deux cas du type spécifié *integer procédure*, vont eux par contre, à l'exécution, provoquer un message d'erreur pour des raisons évidentes.

Nous verrons donc dans un prochain paragraphe comment ALGOL 68 remédie à ce problème.

I.3.2. Ensemble de types fini + vérification dynamique = LISP

L'apparition de LISP dans ce chapitre consacré aux types peut sembler hors de propos puisque ses auteurs assurent qu'une de ses caractéristiques essentielles est justement qu'il n'est pas typé. Le but de ce paragraphe est de montrer que, tout au contraire, la notion de type est sous-jacente dans ce langage, du moins sous certains de ses aspects.

Une des originalités de LISP est que données et programmes ont la même structure externe et interne, c'est celle des S-expressions que l'on peut assimiler à des listes construites à partir d'atomes. Un symbole atomique étant lui-même un identificateur ou un nombre. Certaines fonctions du système LISP(comme PLUS) n'admettent que des arguments atomiques et, plus précisément, des nombres.
Il est donc nécessaire de tester, préalablement à l'interprétation de la fonctior le type de ses paramètres ; ceci se fait au moyen d'un prédicat (NUMBER) qui

délivre la valeur vraie si son argument est un nombre. Ce genre de validation se retrouve au niveau de EVAL, fonction de base de l'interpréteur LISP, puisque l'évaluation d'un atome est différente selon qu'il s'agit d'un nombre ou pas.

Le caractère dynamique des vérifications est lié au fait que LISP n'est pas un langage déclaratif et qu'un même identificateur peut, au cours de l'interprétation d'un programme, être lié à une valeur atomique, à un nombre ou à une S-expression plus élaborée, voire à une fonction. Cette souplesse de fonctionnement permet, entre autres, de générer dynamiquement des fonctions directement interprétables ; cette facilité n'est évidemment possible qu'au détriment de toute vérification statique.

I.4. EXTENSIBILITE DES OBJETS

Une des caractéristiques des langages que nous venons de voir est qu'ils offrent un répertoire fini de types prédéfinis. Le programmeur ne pourra donc décrire des objets complexes qu'à partir de ces types primitifs. De la même façon que, dans une langue naturelle pauvre en vocabulaire, on est obligé d'avoir recours à la périphrase pour décrire, souvent partiellement, de nouveaux concepts. C'est pour pallier cette difficulté ainsi que la prolifération de langages ad'hoc qu'ont été introduits les langages extensibles.

L'extensibilité a porté, dans un premier temps, essentiellement sur la syntaxe du langage, [JOR 72], [BER 73], nous ne nous attarderons pas sur cet aspect qui a donné lieu à de nombreux développements [CHE 66], [JOR 75], [VID 74]. Très vite la nécessité d'appliquer les mécanismes d'extension aux objets manipulés dans le langage est apparue [STA 67], c'est ce qu'on désigne sous le terme général d'extension sémantique.

Il est maintenant classique de distinguer la sémantique statique de la sémantique dynamique [GRI 73] (bien que la frontière entre les deux devienne moins nette [COU 75]). La première traite des caractéristiques contextuelles du langage (essentiellement vérification des types), elle permet de passer de l'arbre d'analyse d'un programme à une nouvelle représentation, l'arbre abstrait, débarrassé

des contraintes, de la syntaxe originale et des propriétés contextuelles.

Il reste maintenant à passer de l'arbre abstrait du programme à un programme exécutable sur une machine, c'est la sémantique dynamique qui permet cette transformation en définissant le répertoire d'opérations de cette machine.

L'extension des objets que nous allons voir sur les deux exemples suivants concerne la sémantique statique, c'est-à-dire la possibilité de définir de nouveaux objets en spécifiant leur configuration et éventuellement certaines règles de leur utilisation. L'extension de la sémantique dynamique sera étudiée plus en détail dans le second chapitre.

I.4.1. Extension des objets + vérifications dynamiques = EL/1

EL/1 est un langage extensible [WEG 70] dont une des originalités est le traitement dynamique des modes. Dans ce langage, de la même façon que 28 est de mode int, TRUE de mode bool, le mode d'un type est mode. C'est dire qu'en EL/1 il est possible de manipuler des valeurs de mode.

exemple:

```
DECL m1:MODE; ...m1←INT...BEGIN DECL J:m1 ; ...END
```

la déclaration de m1 indique que cette variable peut prendre n'importe quelle valeur de mode et être utilisée partout où un mode peut l'être. Dans le bloc interne J est une variable entière puisque m1 a la valeur INT à l'entrée de ce bloc.

La détermination du type d'une variable n'est cependant pas toujours aussi simple, ainsi dans l'exemple:

```
DECL m1:MODE;...[I<0 ⇒ m1←INT ; m1←BOOL] ...BEGIN DECL J:m1;...END
```

Tout ce qu'on peut dire à la compilation c'est que J est soit de mode INT soit de mode BOOL ; le choix entre les deux, dépendant de la valeur de I, ne pourra se faire qu'à l'exécution.

Cet aspect dynamique de la notion de mode rend impossible toute vérification statique. Cependant en EL/1 la séparation classique: compilation, chargement et exécution n'est pas clairement établie, un programme est d'abord interprété afin de déterminer les valeurs constantes, en particulier les valeurs de modes, leur connaissance permet alors une compilation partielle du programme, qui sera de nouveau interprété avec les tests dynamiques qui subsistent.

Une des particularités de EL/1 est de permettre la définition de fonctions génériques, c'est-à-dire de fonctions qui plutôt que de convertir leurs arguments vers un type fixe, exécutent des actions différentes selon le type de leurs arguments, l'opérateur + d'ALGOL 60 est bien générique dans ce sens.

Les appels à de telles fonctions peuvent être:
- soit compilés: le choix de l'alternative se faisant sur la base du type des arguments connus statiquement ;
- soit interprétés: le choix de l'alternative se faisant sur la même base mais dynamiquement.

I.4.2. Extension des objets + vérifications statiques = ALGOL 68

ALGOL 68 est un langage qui a généralisé et systématisé un grand nombre d'idées existant dans les langages de programmation [vWI 77] . Le but de ce paragraphe n'étant pas de faire une revue détaillée de ce travail de synthèse, nous nous cantonnerons aux aspects qui nous intéressent, à savoir le traitement des types, rebaptisés *modes* en ALGOL 68.

Il existe dans ce langage (comme en EL/1) un certain nombre de modes de base: *int*, *real*, *bool*, *char*, *format*... auxquels sont associés une représentation, une notation et un ensemble de fonctions qui leur sont applicables. Les facilités d'extension sont fournies par les constructeurs qui, prenant un ou plusieurs modes en paramètre, délivrent un nouveau mode auquel sont associés une représentation, une notation et un ensemble de fonctions d'accès.

Les constructeurs sont au nombre de cinq: *row* ; *struct*, *proc*, *union* et *ref*.

Les nouveaux modes, définis à partir de ces constructeurs, se voient attribuer un ensemble de fonctions caractéristiques du constructeur ou de la composition des constructeurs (dans ce cas les fonctions associées sont elles-mêmes obtenues par composition des fonctions de base) . Par exemple, tous les modes construits avec *row* disposent des fonctions d'accès utilisant le mécanisme d'indexation et ce indépendamment du type des éléments.

ALGOL 68 étant un langage d'expression, chaque construction possède une valeur donc un mode. Nous prendrons l'exemple des procédures pour illustrer cet aspect.

En ALGOL 68 une routine est une valeur dont le mode est déduit de celui de ses paramètres et de son résultat. Si μ_1, μ_2,...μ_n et μ sont les modes, le mode d'une routine aura l'une des quatre formes suivantes:

proc(μ_1,μ_2,...μ_n)μ	les paramètres sont de mode μ_1,μ_2,...μ_n et le résultat de mode μ.
proc(μ_1,μ_2,...μ_n)*neutre*	idem et la procédure ne délivre pas de résultat.
proc μ	procédure sans paramètre à résultat de mode μ.
proc neutre	procédure sans paramètre ne délivrant pas de résultat.

Une des raisons de considérer les procédures comme des valeurs est que, par exemple, elles peuvent être utilisées comme paramètres d'autres procédures et bénéficier de ce fait d'une vérification complète des types à la compilation. Nous allons illustrer ce point à l'aide du petit exemple déjà écrit en ALGOL W et dont voici une version en ALGOL 68.

```
begin
  [1:5] int a:=(30,10,-2,-1,20), b:=(0,5,0,3,1);
  int l1:=2, l2:=4 ;
  proc car = (int i) int:a[i];
  proc cdr = (int i) int:b[i];

  proc applique = (proc(int)int f, proc int  arg)int:f(arg) ;
```

```
  print(applique(car,int:l1) ;
  print(applique(cdr,applique(cdr,int:l2))) ;

  print(applique(car,cdr)) ;
  print(applique(applique,car)) ;
end
```

Les quatre premières lignes diffèrent assez peu de leurs homologues du programme ALGOL w (aux initialisations près) ainsi d'ailleurs que les quatre dernières. Par contre en ce qui concerne la déclaration de la procédure *applique* on peut noter un certain nombre de changements. En ALGOL W la spécification du premier paramètre f précisait uniquement qu'il s'agissait d'une fonction procédure délivrant un résultat entier et ce, indépendamment du nombre et du type de ses paramètres. En ALGOL 68 on précise que la procédure *f* délivre un résultat entier et qu'elle n'a qu'un paramètre lui-même de type entier. Cette information permettra, en particulier, d'invalider à la compilation le quatrième appel de *applique*.

En ce qui concerne le deuxième paramètre, nous avions vu que le compilateur ALGOL w prenait la liberté de considérer de la même façon, du point de vue syntaxique, un entier et une procédure délivrant un résultat entier. En ALGOL 68, par contre, il n'existe pas de conversion implicite d'un objet de mode m en un objet de mode proc m, il conviendra donc dans les deux premiers appels de applique de convertir explicitement les objets *l1* et *l2* de mode int en objets de mode proc int au moyen de la notation de routine (int:P1).

Le corps de la fonction quant à lui reste inchangé. Il est clair maintenant que les deux premiers appels de *applique* sont syntaxiquement corrects et que le troisième est lui incorrect, *cdr* étant une procédure à un paramètre. L'écriture des paramètres de *applique* ainsi que des deux premiers appels peut paraître un peu lourde, cet inconvénient est cependant largement compensé par un gain de rigueur, donc de sûreté, dans la programmation.

D'une façon générale ALGOL 68 se distingue par la formalisation très poussée des

vérifications statiques de compatibilités des modes. Comme dans les langages du type ALGOL 60 ou ALGOL W, l'association d'un mode à un objet se fait au moyen d'une déclaration pour les variables et d'une notation pour les constantes (possibilité étendue aux tableaux comme le montre l'exemple précédent). En ce qui concerne les expressions,nous avons vu qu'en ALGOL 60 leur type est le résultat de ce que nous avions appelé une évaluation abstraite. Les règles de cette évaluation sont décrites en anglais dans le rapport de définition, ceci parce qu'elles traduisent des propriétés contextuelles du langage. Le mécanisme des W-grammaires adopté pour la définition d'ALGOL 68 permet de prendre en compte syntaxiquement ces propriétés contextuelles. De plus, toutes les constructions du langage possédant un mode, ces propriétés seront souvent présentes dans la grammaire. Ainsi la règle concernant l'affectation:

REF to MODE assignation: REF to MODE destination, becomes token, MODE source.

traduit le fait que dans une affectation le mode de la partie gauche doit être le même (règle de substitution uniforme de la méta-notion *MODE*) que celui de la partie droite à un niveau de référence près. Ce mode est lui-même celui de l'instruction d'affectation , afin de permettre des affectations multiples. Les conversions, aspect sur lequel nous ne nous étendrons pas, sont également traitées au niveau syntaxique par le formalisme des W-grammaires.

Nous conclurons ce chapitre d'introduction en même temps que le paragraphe consacré à ALGOL 68. Ce langage a poussé très loin la formalisation et le champ d'application des vérifications statiques de cohérence, cependant il est également clair que les possibilités de définitions de nouveaux modes restent très liées à la représentation des objets dans la mémoire de la machine. Le second chapitre étudie plus particulièrement ce dernier point et décrit quelques propositions de langages postérieurs à ALGOL 68.

CHAPITRE II

LES TYPES ABSTRAITS

II.1. INTRODUCTION

Comme c'est souvent le cas en informatique, une idée, déjà existante, peut être réactualisée tout en étant approfondie et généralisée. Il en est ainsi de la notion de type abstrait qui se trouve au carrefour des préoccupations linguistiques que nous avons évoquées dans le premier chapitre, mais aussi des problèmes de protection dans les systèmes informatiques ainsi que des courants actuels portant sur la méthodologie de la programmation. L'originalité réside donc essentiellement dans l'unification des concepts.

II.1.1. Les motivations

Le courant méthodologique en programmation s'est développé vers la fin des années 60, il traduisait l'effort d'un ensemble de chercheurs (pour la plupart enseignants) vers une tentative de définition de ce que devrait être "la bonne programmation". Le but visé était double, d'une part il s'agissait de trouver les règles à respecter pour faciliter les preuves de programmes, et d'autre part on essayait de dégager les concepts qui deviendraient la base d'un enseignement cohérent de la programmation. Cet effort de synthèse s'élevait, entre autres, contre la prolifération des langages (cause de la dispersion et quelquefois même de la dissimulation des idées fondamentales) et le fait que, très souvent, l'enseignement de la programmation se résumait à l'apprentissage d'un langage particulier.

Paradoxalement, l'idée de s'abstraire d'un langage donné conduisit à dégager les composants de ce que devrait être un langage d'expression d'algorithme. C'est-à-dire un langage définissant un niveau de machine qui permet de s'abstraire de la plupart des contraintes inhérentes à une machine réelle, pour ne s'intéresser qu'à la résolution algorithmique du problème que l'on s'est posé.

L'effort porta donc, dans un premier temps, sur l'expression du contrôle de cette machine abstraite. La spécification des structures de contrôle fondamentales [BOH 66] conduisit, en outre, à la maintenant fameuse querelle du "goto"

sur laquelle je ne m'attarderai pas en renvoyant le lecteur à l'abondante littérature qui la documente [DIJ 68], [KNU 71], [KNU 74]. Il ne faudrait cependant pas croire que l'apport de cette réflexion se limita à ce débat quelque peu dogmatique.

Un des points importants, reflété d'ailleurs dans la plupart des langages de programmation, est de disposer d'un mécanisme d'abstraction fonctionnelle: c'est l'idée de sous-programme, procédure... A partir d'un ensemble d'actions élémentaires et des primitives de composition (enchaînement séquentiel, conditionnel, itératif) on peut nommer et paramétrer un groupe d'actions pour en disposer par la suite comme d'une action élémentaire. Une partie de la méthodologie réside alors dans la reconnaissance, a priori (et non pas a postériori comme c'était souvent le cas) des abstractions fonctionnelles utiles. On débouche donc sur une approche descendante de la solution qui peut, dans un premier temps, être décrite en termes de ces abstractions dont l'élaboration est alors repoussée Le processus s'applique récursivement jusqu'à l'obtention de l'algorithme final. L'idée n'est évidemment pas neuve (DES 37), de plus, la quasi totalité des langages de programmation disposant d'un tel mécanisme, on peut penser qu'il suffisait de le dire. Les choses ne sont cependant pas aussi simples si on considère maintenant les objets sur lesquels s'appliquent les abstractions fonctionnelles.

En effet, bon nombre de langages n'offrent qu'un répertoire figé d'objets de types différents, si bien que les contraintes de représentation peuvent intervenir très tôt dans le processus de décomposition introduisant ainsi un déséquilibre entre l'abstraction réalisée au niveau fonctionnel et le caractère éminemment concret (souvent très proche de la machine) des objets.

En fait, il manquait à la plupart des langages d'étendre, au niveau des objets, les facilités offertes au niveau fonctionnel. C'est-à-dire la possibilité de définir complètement (fonctions + objets) une machine abstraite adaptée à la résolution d'un problème particulier.

Des langages comme FORTRAN, COBOL,... conçus il y a maintenant une vingtaine d'années se prêtent difficilement à ce genre de définition. Il est cependant clair que, pour des raisons diverses, le nombre de leurs utilisateurs est de plus en plus important. Il devient alors nécessaire de disposer d'outils supplémentaires (dissociés du langage) qui permettent d'exprimer clairement les algorithmes [COU 74], [HUC 77].

L'approche complémentaire tente d'intégrer aux langages ces possibilités, on peut citer à ce sujet PASCAL et ALGOL 68 [WIR 71], [vWI 77]. L'outil qu'est le langage de programmation s'adapte de plus en plus à la méthodologie développée. La frontière entre les deux devenant de moins en moins nette, on peut raisonnablement penser qu'à la limite ils se confondront, l'apprentissage du langage supportant alors l'acquisition de la méthodologie.

II.1.2. L'évolution

De la discussion du chapitre I et du paragraphe précédent, il ressort que la notion de type, en ce qui concerne l'utilisateur, se dégage de plus en plus de la représentation des objets pour s'intéresser au répertoire de fonctions (ou opérations) primitives. Ceci traduit le fait qu 'en programmation il est plus important de savoir ce que l'on peut faire des objets d'un type plutôt que de connaître la manière dont ils sont construits. Ce point prend toute son importance si l'on veut qu'un changement dans la représentation des objets d'un type n'entraîne pas la modification de tous les programmes utilisant ces objets [GOL 73].

De ce point de vue, il est clair qu'ALGOL 68 ne correspond pas entièrement à cette approche dans la mesure où la construction de nouveaux types se fait sur la base de leur représentation en mémoire, représentation qui est toujours accessible au programmeur par le biais des fonctions d'accès associées aux constructeurs utilisés. En fait, les seuls moyens d'abstraction dont on dispose sont la procédure et la possibilité de créer de nouveaux opérateurs.

Les approches des langages extensibles postérieures à ALGOL 68, ainsi que les études portant sur les mécanismes de protection dans les systèmes ont apporté un début de réponse à ces problèmes. Les solutions encore incomplètes suggèrent une séparation marquée entre la représentation des objets et les fonctions (ou opérations) qui leur sont applicables [PAR 71]. Ainsi dans le mécanisme des classes proposé par Jorrand et Bert [JOR 72], [BER 73], [JOR 75] on distingue le langage de base, dans lequel sont exprimées les représentations, du mécanisme d'extension proprement dit qui permet d'ajouter une classe (un nouveau type) dans le graphe des classes en la reliant aux classes existantes par des arcs de fonctions ou de conversions. L'idée de voir un type comme un ensemble de fonctions était donc bien isolée et mise en évidence, toutefois un problème subsistait dans l'association d'une classe à sa représentation. Cette liaison s'opérait implicitement par le biais des fonctions qui manipulaient la représentation. Mais, le langage de base n'étant pas typé, il devenait impossible d'assurer la cohérence de l'ensemble.

Une ébauche de solution a été proposée dans ALEPH [FIS73], l'idée est de disposer d'un niveau de représentation typé, un lien explicite (les fonctions REP et MAKE) existant entre un nouveau type T et le type t qui le représente. La fonction REP permet de considérer un objet de type T comme un objet de type t (concrétisation), la fonction MAKE assurant elle, le passage d'un objet de type t à un objet de type T (abstraction). Toutes les fonctions d'accès et de construction de T sont décrites par l'utilisateur au moyen de REP, MAKE et des fonctions associées au type t, si bien que toutes les manipulations d'objets de T se feront en ignorant complètement la représentation sous-jacente, tout en garantissant la cohérence d'utilisation des fonctions opérant sur cette représentation.

Cependant,une critique subsiste à cette approche. En effet, bien que rendant explicite tout accès à la représentation (fonction REP),elle n'assure pas l'intégrité de cette dernière. La protection contre des accès non prévus par le programmeur n'est pas garantie, il fallut attendre les travaux de Liskov pour avoir une première idée (encore incomplète) de ce que devait être un langage de types abstraits.

II.2. Présentation de quelques langages de types abstraits

Avant toute chose, il convient de rappeler que SIMULA [DAH68], langage conçu en 1967, contenait déjà les prémices des idées que l'on retrouve actuellement. Il admet comme langage noyau ALGOL 60 et fournit, grâce au mécanisme des classes, une approche que tous les nouveaux langages ont plus ou moins reprise. L'idée est de regrouper dans une même unité syntaxique, baptisée *classe*, une collection d'objets avec les fonctions qui les manipulent. La classe deviendra le *cluster* de CLU [LIS 75], la *form* d'ALPHARD [WUL75] et les objets auront en fait la représentation associée au type que l'on définit. Notons cependant que, contrairement à SIMULA, tous les langages de types abstraits assurent la protection de la représentation: on entend par là que toute manipulation de cette représentation ne peut se faire qu'à travers les fonctions du type.

Les paragraphes suivants présentent trois approches différentes de ce que peut être un langage supportant un mécanisme d'abstraction au niveau des objets. Un même exemple sera traité dans les trois cas, celui de la définition du type *ensemble d'entiers* pour lequel on aura, entre autre, à définir les fonctions:

- insert : qui ajoute, s'il n'y est pas déjà, un élément à un ensemble,
- remove : qui enlève un élément à un ensemble,
- has : qui teste si un élément appartient ou non à un ensemble.

Nous ne considérerons pas les opérations entre ensembles (union, intersection, ...) qui compliqueraient inutilement l'exemple.

II.2.1. CLU

CLU est un langage développé au MIT par B.LISKOV et son équipe. C'est la première tentative de définition et d'implantation d'un langage fournissant un mécanisme d'abstraction au niveau des objets. Il admet PASCAL comme langage de représentation, les abstractions sont implantées par des modules appelés *clusters*

pour les objets et procédure pour les fonctions. La séparation entre les différents modules illustre le fait qu'ils supportent différentes abstractions

Le langage est fortement typé afin de vérifier les interfaces entre modules et d'assurer la vérification des types à la compilation (ceci n'est cependant pas toujours le cas comme nous le verrons dans la seconde partie).

Il n'y a , de ce fait,que trois façons d'utiliser un objet abstrait:

1/ il peut être opérande des opérations qui définissent son type abstrait,
2/ il peut être passé en paramètre d'une procédure (il doit alors y avoir identité entre le type du paramètre formel et celui du paramètre effectif),
3/ il peut être affecté à une variable du même type (ce cas est à rapprocher du premier dans la mesure où l'affectation est une opération, définie implicitement sur tous les types, mais qui n'a cependant pas le même statut que les autres, nous le verrons en abordant le problème des paramètres).

L'appel à une fonction définie dans un type se fait en utilisant un nom composé: la première partie est constituée par le nom du type auquel appartient la fonction, la seconde identifie la fonction (les deux sont séparées par le symbole $).

Exemple ensemble d'entiers:

```
(1)   intset = cluster is create, insert, remove,has,equal,copy;

(2)          rep = array of int;

(3)          create = oper( )returns cvt;
(4)                 r:rep:=rep $ create(0);
(5)                 return r
(6)          end create;

(7)          insert = oper(s:cvt,i:int);
(8)                 if search(s,i) ≤ rep $ high(s)then return;
(9)                 rep $ extendh(s,i);
(10)                return
(11)         end insert,

(12)         search = oper(s:rep,i:int)returns int;
(13)                for j:int:=rep $ low(s) to rep $ high(s)by 1 do
(14)                  if i=s[j] then return j;
(15)                return rep $ high(s)+1
(16)         end search;

(17)         remove = oper(s:cvt,i:int);
(18)                j:int:=search(s,i);
(19)                  if j ≤ rep $ high(s)then
(20)                       begin
(21)                            s[j]:=s[rep $ high(s)];
(22)                            rep $  retrachth(s)
(23)                       end;
(24)                     return
(25)         end remove;

(26)         has = oper(s:cvt,i:int) returns boolean;
(27)                if search(s,i) ≤ rep $ high(s)
(28)                     then return true
(29)                     else return false
(30)         end has;

(31)         equal=oper(s,t:cvt)returns boolean
(32)                if rep $ size(s)≠rep $ size(t)then return false;
(33)                     for i:int:=rep $ low(s)to rep $ high(s) by 1 do
(34)                        if search (t,s[i]) ≥rep $ high(t) then return fals
(35)                        return true
(36)         end equal;

(37)         copy = oper(s:cvt) returns cvt;
(38)                return rep $ copy(s)
(39)         end copy

(40)  end intset
```

Une définition de *cluster* comprend essentiellement trois parties. Tout d'abord (ligne 1) la donnée du nom du cluster (intset) associée à celle de l'interface elle-même constituée de la liste des fonctions (ou opérateurs) qui définissent le type. Seules ces fonctions seront, dans un programme donné, applicables aux objets du type. Cette propriété sera vérifiée statiquement par le compilateur.

La deuxième partie (ligne 2) fournit la représentation des objets, il s'agit dans l'exemple du type tableau d'entier qui est prédéfini dans le langage de représentation. Afin de raccourcir l'écriture, le type de représentation (qui peut être complexe) sera dénoté *rep* dans le module (c'est ce que traduit la déclaration d'identité *rep = array of int)*. La représentation n'étant pas accessible de l'extérieur du module, c'est seulement à l'intérieur de ce dernier que l'on pourra utiliser les fonctions définies sur les objets du type *rep*. Par exemple (ligne 9) l'appel *rep $ extendh(s,i)*, qui ajoute un élément au tableau s en y rangeant l'entier i.

La troisième partie comprend les définitions de fonctions (celles de l'interface plus celles locales au module). Ces fonctions reçoivent comme paramètres, entre autre, des objets du type *intset*, mais opèrent, telles qu'elles sont définies dans le type, sur des tableaux d'entiers ; ce fait se traduit par la déclaration du paramètre *s:cvt* ligne 7) qui indique que l'objet de type intset, passé en paramètre, devra être *converti* vers le type de la représentation. Cette spécification de conversion est utilisée également pour les fonctions retournant un objet du type intset (ligne 37). On peut noter que la fonction *search*, locale à la définition (son nom n'apparaît pas dans l'interface) possède un paramètre de type *rep*. La plupart des langages déclaratifs fournissent un exemplaire de l'objet au moment de la déclaration, ce n'est pas le cas en CLU, où les deux fonctions sont obligatoirement dissociées ; on peut déclarer une variable:

si : intset ;

et créer ensuite un objet de type intset qu'on lui affecte:

si := intset $ create( );

On peut alors appliquer à *si* les fonctions du type *intset:*

intset $ insert(si,1);

Après ces quelques considérations sur la forme externe du langage, nous allons essayer d'aborder quelques problèmes touchant plus particulièrement la sémantique.

La dernière remarque indique clairement la séparation entre le concept de variable et celui d'objet. En CLU, les objets existent dans un univers comparable au tas d'ALGOL 68. Les variables quant à elles, dénotent ces objets: concrètement ce sont des pointeurs vers les objets du tas. Dans cette optique, l'affectation entre variables devient une affectation de pointeurs, les deux variables partagent alors le même objet. L'obtention d'un nouvel exemplaire de l'objet ne peut se faire que par l'invocation de la fonction *copy* qui doit donc être fournie dans toute déclaration de type.

Poursuivant la même idée, le passage de paramètre se fait par référence, le paramètre effectif est affecté au paramètre formel. Une différence fondamentale existe cependant avec le passage par référence classique, car toute affectation au paramètre formel ne modifie pas le paramètre effectif (le paramètre formel *pointe* simplement vers un nouvel objet, le paramètre effectif n'est alors plus accessible). Les seuls effets de bords possibles le sont via les fonctions d'accès définies dans le type du paramètre.

II.2.2. ALPHARD

ALPHARD est un langage dont la définition et le développement sont le fruit d'une collaboration interuniversitaire aux Etats-Unis [WUL75]. Cet important projet vise à réaliser la synthèse entre, d'une part les idées actuelles sur la méthodologie de la programmation et,d'autre part, les techniques de vérifications de programmes. L'idée est de proposer un langage qui permette d'améliorer la qualité de la programmation tout en réduisant son coût.

La base du mécanisme d'abstraction en ALPHARD est la *form*. Comme le *cluster* de CLU, elle permet de regrouper dans une même unité syntaxique les éléments significatifs d'une abstraction. Ces éléments sont divisés en trois parties distinctes:

- la partie *spécification:*

Où l'on trouve les informations nécessaires à l'utilisateur de la *form*. Contrairement à l'interface de CLU, qui rappelons-le, ne contient que les noms des fonctions du type, cette partie fournit des renseignements plus formels sur les objets abstraits ainsi que sur les fonctions qui les manipulent. Ces renseignements sont essentiellement la donnée d'un prédicat caractérisant, par extension, les objets abstraits ainsi que les pré et post conditions [HOA69].

- la partie *représentation:*

Elle fournit la représentation concrète des objets abstraits, cet ensemble d'objets concrets est lui-même défini par un prédicat. La correspondance entre un objet concret et l'objet abstrait qu'il représente est exprimée par une fonction de représentation notée *rep*.

- la partie *implémentation:*

C'est là que l'on trouve les définitions des fonctions opérant directement sur la représentation ainsi que les pré et post conditions concrètes qui leur sont attachées.

On voit donc qu'en ALPHARD, les objets définis par une *form* sont spécifiés à deux niveaux. Ils sont, dans un premier temps, caractérisés formellement par des prédicats, eux-mêmes exprimés en fonction d'entités mathématiques supposées prédéfinies (entier, booléen, séquence...). Ce sont ces spécifications qui permettent de vérifier les utilisations de la *form*. L'autre niveau qui nous est maintenant plus familier est celui de la représentation des objets qui exige lui aussi un ensemble de vérifications internes à la *form* pour assurer la validit de la représentation. Cette séparation entre la preuve des programmes utilisant une abstraction et la preuve de l'implantation de cette abstraction constituent un des points importants du langage. Le programme principal sera vérifié par la méthode traditionnelle des assertions [FLO67] en considérant les objets abstraits et leurs opérations comme primitifs. Quant aux *form*, on vérifiera que la représentation concrète est cohérente avec les spécifications (ceci conduisant à des vérifications de programmes analogues à celles du programme principal).

Voyons comment s'écrit notre exemple en ALPHARD:

```
form smallintset (maxsize:integer)
    beginform
        specifications
          requires maxsize ≥ 0;
          let smallintset = {...x,i...} where x,i is integer;
          invariant cardinality (smallintset) ≤ maxsize;
          initially smalintset={ };
          function
             insert(s:smallintset,x:integer)
                pre cardinality({x}∪s) ≤ maxsize
                post s=s'∪{x}
             remove(s:smallintset,x:integer)
                post s=s'-{x}
             has (s:smallintset,x:integer)returns(b:boolean)
                post b=x∈s'
        representation
          unique v:vector(integer,1,maxsize),m:integer init m←0;
          rep(v,m) = {v[i]|i∈[1..m]};
          invariant 0 ≤ m ≤ maxsize ∧ (∀i,j∈[1..m] (v[i]=v[j]⊃i=j));
        implementation
          body insert in (∃i∈[1..s.m]st x=s.v[i] ∨ s.m < maxsize)
                      out (∀i∈[1..s.m'] (s.v[i] = s.v'[i])∧
                           (∃j∈[1..s.m] st s.v[j]=x))=
                      first p:upto(1,s.m) suchthat s.v[p]=x
                           else (s.m←s.m+1 ; s.v[s.m]←x);
          body remove out (∀j∈[1..s.m](s.v[j]≠x) ∧ (∀i∈[1..s.m'] ∃j∈[1..s.m]
                           (s.v'[i]≠x ⊃ s.v[j]=s.v'[i]))) =
                      first p:upto(1,s.m) suchthat s.v[p]=x
                           then (s.v[p]←s.v[s.m]; s.m←s.m-1);
          body has out (b ≡ (∃i∈[1..s.m] st s.v[i]=x) ∧
                            s.v'=s.v ∧ s.m'=s.m) =
                      first p:upto (1,s.m) suchthat s.v[p]=x
                           then b←true else b←false;
    endform
```

Quelques remarques sur les notations utilisées:

- l'abréviation st, utilisée dans les pré et post conditions, signifie *such that*.
- la notation abrégée *s.m* désigne l'entier *maxsize* associé à l'ensemble *s*.
- la notation primée *s.v'[i]* par exemple désigne la valeur du symbole *(s.v[i])* avant exécution de l'opération, elle est utilisée essentiellement dans les post-conditions.
- l'instruction *first* est un itérateur, elle permet d'énumérer, dans notre cas, les éléments d'un intervalle d'entiers.

On retrouve sur cet exemple les trois parties évoquées plus haut. La spécification donne (*let*...) l'équivalent formel du type que l'on est en train de définir. Il s'agit dans notre cas d'ensembles d'entiers, restreints, à l'aide de l'invariant abstrait, à la classe des ensembles d'entiers dont le cardinal est inférieur au paramètre *maxsize*. Les fonctions quant à elles, sont définies par leur nom et leur fonctionnalité (spécification syntaxique des paramètres et du résultat) ainsi que la donnée des pré et post conditions qui portent sur les objets formels. C'est au niveau suivant que l'on trouve la représentation des objets du type *smallintset*, il s'agira dans notre cas de vecteurs d'entiers, la fonction *rep* permettant de passer d'un vecteur v à l'objet formel qu'il représente.

De la même façon qu'au niveau abstrait, la classe des représentations valides est définie par extension de l'invariant concret (c'est-à-dire l'invariant de la représentation). C'est dans la partie *implémentation* que l'on commence à programmer, on trouve là, en effet, le corps des fonctions, spécifiées plus haut,auxquels on ajoute les assertions concrètes d'entrée et de sortie. Confronté à une telle spécification, le système va entreprendre un certain nombre de vérifications.

Il devra tout d'abord s'assurer de la validité de la représentation, c'est-à-dire vérifier que si une représentation respecte l'invariant concret, alors l'application de la fonction *rep* à cette représentation respecte elle, l'invariant abstrait.

Dans notre cas:

$$0 \leq m \leq maxsize \wedge (\forall i,j \in [1..m]\ (v[i]=v[j] \supset i=j)) \supset$$
$$cardinality\ (\{v[i] \mid i \in [1..m]\}) \leq maxsize$$

Il vérifiera ensuite que l'initialisation des objets est faite correctement, et que, pour chaque fonction l'implémentation est correcte. C'est-à-dire que si l'assertion concrète d'entrée est vérifiée ainsi que l'invariant concret (I_c),

alors après exécution du corps de la fonction l'assertion concrète de sortie et l'invariant concret sont vérifiés.

Dans le cas de la fonction *insert* de l'exemple, la formule à vérifier serait:

$(\exists i \in [1..s.m]\ st\ x=s.v[i] \vee s.m<maxsize) \wedge I_c \{\underline{first}\ p:upto(1,s.m)\ \underline{suchthat}\ s.v[p]=x\ \underline{else}\ (s.m \leftarrow s.m+1; s.v[s.m] \leftarrow x\}\ (\forall i \in [1..s.m']\ (s.v[i]=s.v'[i]) \wedge (\exists j \in [1..s.m]\ st\ s.v[j]=x)) \wedge I_c$

Ceci peut se décomposer en deux parties correspondant aux deux alternatives de l'instruction *first*.

Pour la première alternative, on doit vérifier:

$(\exists i \in [1..s.m]\ st\ x=s.v[i] \vee s.m < maxsize) \wedge Ic \wedge (\forall k \in [1..p-1]\ (s.v[k] \neq x)) \wedge s.v[p]=x\ \{\}\forall i \in [1..s.m']\ (\exists i \in [1..s.m]\ st\ s.v[i]=s.v'[i]) \wedge (\exists i \in [1..s.m]\ st\ s.v[j]=x) \wedge Ic$

Cette alternative ne modifiant pas l'exemple, il est clair que la formule sera vérifiée en choisissant $j=p$.

Examinons maintenant la deuxième alternative:

$(\exists i \in [1..s.m]\ st\ x=s.v[i] \vee s.m < maxsize) \wedge Ic \wedge (\forall k \in [1..s.m]\ (s.v[k] \neq x)) \{s.m \leftarrow s.m+1\ ;\ s.v[s.m] \leftarrow x\} \forall i \in [1..s.m']\ (s.v[i]=s.v'[i]) \wedge (\exists j \in [1..s.m]\ st\ s.v[j]=x) \wedge Ic$

Le premier terme de la conjonction est vrai du fait que $s.m > s.m'$ ($s.m'$ désignant la valeur de $s.m$ avant l'exécution de la fonction), pour le second il suffit de prendre $j=s.m$. Le premier terme de l'invariant concret Ic , est vrai car le terme $\forall k \in [1..s.m]\ (s.v[k] \neq x)$ impose que $s.m < maxsize$ est vrai dans l'hypothèse ; le second terme de l'invariant concret est vérifié de façon évidente.

Il reste maintenant à vérifier la cohérence entre les spécifications abstraites et les spécifications concrètes, c'est-à-dire que chaque fois que la pré-condition abstraite est vraie, alors l'assertion d'entrée de l'opération concrète est vérifiée. De même, toutes les fois que les conditions d'applications de l'opération concrète sont vérifiées, alors l'assertion de sortie est suffisamment forte pour impliquer la post-condition abstraite. Le lecteur pourra vérifier l'exactitude de ces relations sur l'exemple de la fonction *insert*, on donne ci-dessous les formules à vérifier:

1) $I_c \wedge cardinality\ (\{x\} \cup s) \leq maxsize \supset$
$(\exists i \in [1..s.m]\ st\ x=s.v[i] \vee s.m < maxsize)$

2) $I_c \wedge cardinality\ (\{x\} \cup rep(v',s.m')) \leq maxsize$
$\wedge\ (\forall i \in [1..s.m']\ (s.v[i]=s.v'[i])$
$\wedge\ (\exists j \in [1..s.m]\ st\ s.v[j]=x)) \supset s=s' \cup \{x\}$

II.2.3. Les spécifications algébriques

L'idée de considérer un type comme un ensemble de valeurs construit à partir d'opérateurs a conduit à ce genre de spécifications dont les principaux développements sont décrits dans[ZIL75],[GOG75],[GUT76a],[GUT76b]. Un des avantages de cette méthode est de fournir une description complète du type abstrait qui soit indépendante de toute représentation.

La spécification est divisée en deux, une partie syntaxique (déjà vue dans les langages précédents) définit les noms des fonctions associées aux types abstraits ainsi que leur domaine et leur co-domaine. La partie sémantique contient un ensemble d'axiomes qui décrivent les relations entre les expressions formelles du type. Les deux spécifications sont suffisantes pour décrire l'ensemble des expressions valides que l'on peut construire à l'aide des opérations.

Ainsi pour notre exemple, on aurait:

Spécifications syntaxiques

1/ *EMPTY:* → *SMALLINTSET*

2/ *INSERT:* *SAMLLINTSET* × *INTEGER* → *SMALLINTSET*

3/ *REMOVE:* *SMALLINTSET* × *INTEGER* → *SMALLINTSET*

4/ *HAS:* *SMALLINTSET* × *INTEGER* → *BOOLEAN*

Spécifications axiomatiques

a/ *HAS(EMPTY,I) = FALSE*

b/ *HAS(INSERT(S,I),J) = IF EQ(I,J) THEN TRUE*
ELSE HAS(S,J)

c/ *REMOVE(EMPTY, I) = EMPTY*

d/ *REMOVE(INSERT(S,I),J) = IF EQ (I,J) THEN REMOVE(S,J)*
ELSE INSERT(REMOVE(S,J),I)

La difficulté avec ce genre de spécification est de savoir si les axiomes construits décrivent bien l'ensemble de valeurs auquel on s'intéresse et rien que cet ensemble. On retrouve d'ailleurs ce genre de problème avec les spécifications logiques d'ALPHARD (la post-condition d'une fonction est-elle suffisamment forte pour en décrire l'effet?). La réponse à ces questions est, évidemment, qu'il est impossible d'en avoir une certitude formelle ; il faudrait en effet disposer d'une spécification de la spécification pour laquelle se poserait le même problème.

Ainsi, dans l'exemple que nous avons donné, les quatre axiomes décrivent tous les ensembles possibles d'entiers.

Il est possible de rajouter l'axiome suivant:

e/ *INSERT (INSERT(S,I)J) = IF EQ(I,J) THEN INSERT(S,I)*
ELSE INSERT(INSERT(S,J),I)

qui permet de considérer les trois expressions:

INSERT (INSERT(EMPTY,1),2)

INSERT (INSERT(EMPTY,2),1)

INSERT (INSERT(INSERT(EMPTY,1),2),2)

comme étant trois notations d'un même ensemble.

Un point très important est de savoir si l'ensemble d'axiomes est non contradictoire, question en général indécidable. Ainsi si on ajoute l'axiome:

f/ *HAS(REMOVE(S,I),J) = HAS(S,J)*

aux quatre premiers axiomes.
Il existe des valeurs du type *SMALLINTSET* pour lesquelles on peut démontrer *HAS(S,I) = TRUE* et *HAS(S,I)=FALSE* selon les axiomes qu'on choisit d'utiliser. (Il suffit de considérer l'expression *HAS(REMOVE (INSERT(EMPTY,1),1),1)* à laquelle on "applique" l'axiome (f) ou l'axiome (d)).

Dernier aspect enfin, celui de la validité de la représentation choisie. Il s'agit en fait de montrer qu'elle est une image isomorphe du type abstrait défini, c'est-à-dire qu'elle vérifie les axiomes énoncés. La théorie des catégories fournit un support mathématique à ce genre de vérifications , son application à la théorie des types en programmation est étudiée dans [FOS77], [GOG73].

II.3. Conclusion

Dans ce chapitre consacré aux types abstraits, les trois langages évoqués correspondent à trois approches du concept d'abstraction à la fois différentes (en ce qui concerne le niveau de spécification),mais aussi complémentaires,si on considère leur champ d'application. A l'heure actuelle,il

existe bon nombre de langages qui reprennent, en les approfondissant, certaines des idées exposées.

On peut citer EUCLID [LAM77], langage plus orienté vers l'écriture de systèmes où l'on retrouve certaines techniques de vérifications de programmes développées dans ALPHARD, ainsi qu'un approfondissement des notions de portée, de visibilité visant à éliminer les effets de bords. Les SCHEMES [MIT77] de Mitchell et Webreit qui apportent une réponse partielle aux problèmes de paramétrisation des définitions, nous aurons l'occasion de revenir en détail sur ce point dans le chapitre suivant. MODULA [WIR76] extension du langage PASCAL en vue de l'écriture de systèmes sur mini-calculateurs. GYPSY [AMB76] orienté vers l'écriture de programmes "vérifiables" pour les systèmes de communication.

La majorité de ces langages constituent à l'heure actuelle des voies de recherche, ils ont été pour la plupart, implantés de manière expérimentale. A ce sujet, il convient de citer un important projet dont le Département de la Défense des USA a été l'instigateur. Son objet est de définir un langage qui deviendra le langage de programmation utilisé dans tous les organismes qui dépendent de lui. Il est intéressant de noter que dans les spécifications [IRO77] du projet, une large part a été consacrée aux possibilités d'abstractions et aux mécanismes de protections associés, certains des contractants ont même proposé des solutions relativement originales pour un langage à vocation "industrielle". En France, on peut citer les projets; LEST[BET77], SOC[BAN78], HELOISE[ROU77] et MEFIA[CUN78] qui tous proposent un mécanisme d'abstraction des objets au service d'applications variées: écriture de systèmes, rétention d'objets typés, spécifications de programmes, etc...

La plupart des langages évoqués s'appuient sur PASCAL dont ils constituent des extensions plus ou moins importantes, on peut alors se poser la question de savoir en quoi ce langage n'est pas adapté au concept d'abstraction? Pour ce faire, voyons comment traiter, en PASCAL, notre exemple d'ensemble d'entiers.

Le type lui-même peut être défini de la façon suivante:

(1) *type smallintset = array[1..m] of integer*

avec par exemple, l'en-tête de procédure:

procedure insert (s:smallintset ; i:integer) ;

Afin de montrer l'insuffisance de ce langage à imposer et par là même à supporter l'idée d'abstraction, il convient de revenir en quelques mots sur cette dernière.

Il est clair maintenant qu'une abstraction est un concept recouvrant plusieurs réalisations et, qu'en ce sens, un type défini par un utilisateur est une abstraction à la fois de la représentation et des fonctions qui la manipulent. Un point important est évidemment de savoir comment les langages de programmation favorisent l'expression de ces abstractions. Il ressort de ce que nous avons vu dans ce chapitre que le support linguistique commun est la "modularisation". Elle permet de regrouper, dans une même unité syntaxique, tous les traits caractéristiques d'une abstraction, en favorisant l'utilisation de méthodologies de programmation. En ce qui concerne les spécifications et les preuves qui en découlent, la modularisation permet de les localiser et de les "atomiser", ce point prend toute son importance lorsqu'on sait que la complexité d'une preuve croît de façon exponentielle en fonction de la taille du programme. Dernier point enfin, le module permet de protéger la représentation contre tout accès anarchique (en dehors des fonctions du module). C'est à ce niveau que l'insuffisance de PASCAL apparaît, elle est principalement due au fait que la définition (1) ne constitue qu'une abréviation (*smallintset*) du type de la représentation (*array[1..m] of integer* .Dans la procédure suivante:

procedure f(smallintset:s ; i,j:integer)

des accès comme $s[i]$ sont tout à fait valides, bien que f ne soit pas une fonction caractéristique du type *smallintset*.

C'est pour toutes ces raisons que des langages comme PASCAL, voire même, ALGOL 68 ne permettent pas de considérer correctement des objets de façon abstraite. Réciproquement, on pourrait maintenant se demander si tous les

langages évoqués permettent d'exprimer dans sa totalité l'idée d'abstraction dont une des composantes essentielles, que nous n'avons pas encore évoquée, est la paramétrisation. C'est le but du chapitre suivant d'étudier ce problème.

CHAPITRE III

LES TYPES ABSTRAITS GENERIQUES

INTRODUCTION

L'exemple, utilisé dans le chapitre précédent pour illustrer les différents langages étudiés, met en relief une lacune importante dans le choix de l'extension. Il parait en effet surprenant de définir un type abstrait "ensemble d'entiers", muni des fonctions insérer, enlever, appartient, et redéfinir par la suite un type "ensemble de réels", disposant des mêmes fonctions (dont l'écriture sera vraisemblablement identique à celle des précédentes mais auxquelles on devra éventuellement donner un nom différent) et, muni probablement de la même représentation pour les objets (tableau, liste, ...).
Il parait souhaitable de pouvoir définir un type abstrait "ensemble de n'importe quoi", c'est-à-dire paramétrer la définition du type ensemble par le type de ses éléments afin de décrire en une seule fois la représentation (commune et donc elle-même paramétrée par le type des éléments) des objets ainsi que les fonctions qui définissent le type.

Les types abstraits, qui ne devaient jusqu'à maintenant leur qualificatif qu'à l'abstraction de la représentation des objets qu'ils réalisaient, sont alors munis d'un mécanisme qui augmente notablement leur capacité d'abstraction dans le sens où il permet de définir toute une classe de types.

Ce chapitre constitue la partie principale de cette thèse, il présente les résultats essentiels de nos travaux concernant la paramétrisation des types abstraits par des types.

III.1. LA GENERICITE

Avant d'aborder plus en détail la paramétrisation des types abstraits et les problèmes techniques qu'elle pose, il est nécessaire de revenir en quelques mots sur la notion de généricité qui joue un rôle important, sinon essentiel, en informatique.

A ce sujet, il convient de rappeler que l'ordinateur en constitue un exemple fondamental. En effet, cette machine à calculer, à partir du programme qu'on lui soumet en entrée, se spécialise pour effectuer un traitement particulier dont le résultat dépend lui-même des données fournies. L'analogie avec la notion de fonctions en mathématiques est évidente, elle traduit la dépendance de l'évaluation d'une expression à la valeur d'un certain nombre de variables appelées paramètres en programmation.

Dès 1965, Landin [LAN65] assimile un programme à l'expression d'une fonction, et, entre les diverses théories de la calculabilité édifiées par les logiciens, c'est celle du lambda-calcul de Church [CHU41] qu'il choisit afin de traduire tout programme sous forme de λ-expression, c'est-à-dire un notation de fonction dans la théorie. Ce dernier point est précisément essentiel, l'exemple suivant fait apparaître la nécessité de disposer d'une notation particulière pour les fonctions.

Considérons l'expression $2x^2 + 3xy$, appliquée aux arguments (2,3), par exemple, elle constitue une notation ambigüe dont la valeur peut être 26 ou 36 .

Par contre la fonction $\lambda xy.\ (2x^2 + 3xy)$ appliquée aux arguments (2,3) prend la valeur unique 26.

Parmi les trois règles de formation des λ-expressions, c'est la règle d'abstraction qui permet de passer de la notation d'une expression à celle de la fonction correspondante en isolant un certain nombre de variables.

C'est-à-dire que si E est une λ-expressions bien formée et x,y des variables libres dans E alors (λxy.E) est la notation de fonction correspondante. On voit donc que paramétrer une expression revient, dans la théorie, à définir une fonction.

III.1.1. LA GENERICITE DES FONCTIONS

Nous avons déjà évoqué dans le chapitre II les possibilités d'abstraction fonctionnelle offertes par les langages qui permettent la définition de procédures (fonctions, sous-programmes). Le passage de la notation de programme à la notation de procédure consiste, là aussi, à isoler un ensemble de variables, qui deviennent les paramètres, et à nommer la procédure afin de pouvoir s'y référer par la suite.

Considérons la notation de programme suivante, écrite dans un langage de type ALGOL:

```
(1) début
        entier s ;
        s := 0 ;
        pour i:=1 jusqua n faire s:=s+a(i) ;
    fin
```

Ce programme calcule la somme des éléments d'un tableau d'entiers à une dimension. Les variables libres de cette notation sont, évidemment, le tableau d'entiers, noté a, et la borne supérieure de la dimension, notée n. Ces variables seront donc les paramètres de la notation de fonction:

```
(2) fonction Reduc (entier tableau a(*),entier n) retourne entier ;
      début
        entier s ;
        s:=0 ;
        pour i:=1 jusqua n faire s:=s+a(i) ;
        s
      fin Reduc ;
```

Nous voyons sur cet exemple que les paramètres de la fonction sont contraints par la donnée de leur type. Cette spécification, du point de vue de l'utilisateur, sert à valider les différents appels de la fonction en vérifiant la

compatibilité entre le type des paramètres effectifs et celui des paramètres formels. C'est en fait un moyen de restreindre le domaine de la fonction.

Si on se place maintenant du point de vue du compilateur (c'est-à-dire de l'implanteur) ou tout simplement de l'interprète du programme, la donnée du type va permettre, entre autres, de lever un certain nombre d'ambiguités de la notation. Il est évident que la construction pour ... jusqua ... faire possède une interprétation, donc une traduction standard et, de ce fait, ne pose pas de problème particulier. Ce n'est, par contre, pas le cas de l'affectation. Définie sur tous les types simples du langage (entier, réel, booléen...) elle dispose de plusieurs interprétations prédéfinies parmi lesquelles l'interprète (le compilateur dans notre cas) devra choisir la bonne. C'est bien évidemment la connaissance du type des arguments qui va guider son choix. Le mêmeproblème se pose pour l'addition, définie aussi sur les réels et possédant également la même notation (+).

Tout se passe comme si := et + étaient des notations formelles utilisées chacune pour désigner une classe de fonctions (la classe des fonctions d'affectation, celle d'addition) et que la donnée du type des paramètres, sur lesquels opèrent ces fonctions dans la procédure suffit pour choisir la bonne interprétation. De tels opérateurs (ou fonctions) sont qualifiés de génériques, ils possèdent plusieurs interprétations dont le choix se fait en fonction du type de leurs arguments. Leur utilisation, dans les langages de programmation, permet une notation synthétique en ce sens qu'ils réalisent l'abstraction d'un ensemble de fonctions. On peut se demander alors s'il n'est pas possible de donner au programmeur la possibilité de définir ses propres procédures (ou fonctions) génériques puisque ce mécanisme offre une dimension nouvelle à l'abstraction fonctionnelle.

III.1.1.1. La généricité incrémentale

Le problème est de savoir de quelle façon ce mécanisme sera mis en oeuvre dans un langage particulier. Nous en distinguerons deux. Le premier, que nous baptiserons généricité incrémentale consiste à redéfinir un opérateur (ou une fonction) sur de nouveaux types de façon à disposer de plusieurs interprétations différentes.

Par exemple, la fonction Reduc, définie sur les tableaux d'entiers peut être redéfinie sur les booléens:

```
(3) fonction Reduc(booléen tableau a(*),entier n) retourne booléen ;
      début
        booléen s ;
        s:=faux ;
        pour i:=1 jusqua n faire s:=s ou a(i) ;
        s
      fin Reduc ;
```

puis sur les réels:

```
(4) fonction Reduc (réel tableau a(*),entier n) retourne réel ;
      début
        réel s ;
        s:=0. ,
        pour i:=1 jusqua n faire s:=s+a(i) ;
        s
      fin Reduc ;
```

On dispose alors de trois interprétations de Reduc dont le choix, pour un appel particulier, se fera sur la base du type des arguments:
Ainsi après les déclarations suivantes:

```
      booléen b ;
      booléen tableau bt (1::10) ;
        ⋮
(5) b:=Reduc(bt,10) ;
```

Le compilateur choisira l'interprétation (3) pour l'appel (5) de la fonction Reduc

Cette façon de mettre en oeuvre la généricité a été adoptée dans plusieurs langages (vWI77), (IR077). On peut en trouver une étude assez complète dans (SCH75), le choix de l'interprétation étant guidé par l'évaluation d'un prédicat attaché à chaque membre de la fonction. Dans le cas le plus simple ce prédicat est simplement un test de types sur les opérandes (approche ALGOL 68), mais le mécanisme peut s'étendre en considérant des prédicats testant (dynamiquement?) la valeur des arguments, le problème du choix déterministe de la "bonne" interprétation se pose alors. La solution envisagée consiste à considérer un ordre sur ces prédicats (ordre induit par l'inclusion des ensembles qui en sont les extensions), en cas d'ambiguité on choisira alors le plus petit.

Le problème se trouve maintenant déplacé, il s'agit en effet de valider un ensemble de prédicats attachés aux différents membres d'une fonction générique. Cette validation consiste à montrer que les ensembles associés sont, soit inclus proprement, soit disjoints (si deux prédicats sont vrais en même temps, alors l'un est toujours "plus petit" que l'autre), ce genre de vérification (en dehors du cas simple des types, qui sont disjoints par nature) semble particulièrement difficile à réaliser, qui plus est, dans cette approche il est clair que c'est le nom des différentes fonctions qui est privilégié ; c'est la seule spécification qu'elles ont en commun.

Nous pensons que ce mécanisme est la porte ouverte à de nombreuses erreurs dues en particulier à ce que l'on peut redéfinir la fonction avec un nombre de paramètres différents, mais aussi que certaines interprétations sont complètement différentes de la signification initiale. A la limite, ce mécanisme peut s'utiliser simplement pour faire l'économie d'un nouvel identificateur en perdant complètement l'idée d'abstraction sous-jacente.

III.1.1.2. La généricité structurale

Au-delà de ces critiques, l'exemple que nous avons traité conduit à faire une remarque intéressante. On peut en effet noter que les différents corps de fonctions que nous avons écrits sont quasiment "semblables". A la limite, la version (2) et la version (4) possèdent la même notation de programme, la seule différence réside dans le type des éléments du tableau passé en paramètre (ceci est dû au fait que l'opérateur + est générique sur les entiers et les réels et que 0 est une notation de constante entière convertible automatiquement en une valeur réelle). En poursuivant la démarche adoptée pour la définition d'abstractions fonctionnelles, on est tenté de faire du type des éléments du tableau, un paramètre de la fonction, ce qui conduit à l'écriture suivante de Reduc:

```
(6) fonction Reduc (type t, t tableau a(*), entier n) retourne t ;
    début
       t s ;
       s := 0 ;
       pour  i:=1 jusqua n faire s:=s+a(i) ;
       s
    fin Reduc ;
```

En première approximation, on peut considérer cette définition comme une macro qui permet d'engendrer autant de fonctions (différentes) qu'il y a d'appels avec des types différents dans un programme donné.

Nous appellerons généricité structurale cette deuxième mise en oeuvre de la généricité, elle a fait l'objet d'un certain nombre d'études (LIN), (BOO76) et a été implantée avec plus ou moins de réussite dans quelques langages (WUL76),(LIS75).

Contrairement à l'approche précédente, les différentes interprétations de la fonction Reduc ne partagent plus seulement le nom mais aussi les paramètres ainsi qu'une notation de programme dans laquelle subsiste un certain nombre d'éléments indéterminés. En effet, le choix de l'interprétation de := (s:=0, s:=s+a(i)) et de + ne pourra pas se faire à la compilation de la définition de Reduc, ce choix dépend du type t passé en paramètre (il en est de même pour la constante 0 qui initialise la variable s). Par contre au moment de l'appel, il devra être possible de lever l'ambiguité, le type t étant alors connu.

La généricité de la fonction Reduc dépend donc essentiellement de celle des opérateurs := et + sur l'ensemble des types t susceptibles d'être paramètres effectifs d'un appel de Reduc.

Le problème est maintenant de caractériser cette classe de types afin de valider statiquement les appels de Reduc et de fournir une notation formelle pour l'opérateur + de la définition qui sera lié à un opérateur effectif lors d'un appel (on a vu en effet que pour la définition de Reduc sur les booléens, c'est l'opérateur ou qui est utilisé à la place de +). Par contre en ce qui concerne l'affectation on peut admettre que la notation := est utilisée pour tous les types du langage.

Nous verrons (cf. III.2) une nouvelle notion, la propriété, qui permet, à la fois de caractériser une classe de type par l'ensemble minimum de fonctions qu'il doit implanter, et également de lier, lors d'un appel, les éléments (opérateurs, constantes) formels de la définition aux éléments effectifs correspondants du type passé en paramètre.

III.1.1.3. Discussion

Les deux mécanismes de mise en oeuvre de la généricité évoqués dans ce paragraphe appellent quelques remarques:

- Le premier, nous l'avons vu, consiste à lier à un même identificateur (le nom de la fonction), plusieurs définitions complètes (fonctionalité + notation de programme) de fonctions. Cet identificateur constitue alors la seule spécification commune aux différents membres.

- Le second consiste à écrire un programme qui utilise des opérateurs ou des fonctions eux-mêmes génériques au sens incrémental, ce qui revient à faire du type de certains objets un paramètre de la fonction. Il y a donc partage, entre les différentes interprétations d'une fonction générique, d'une même notation de programme ; ceci nous parait fondamental pour deux raisons:

  a/ on conserve la notion d'abstraction fonctionnelle (une même notation de programme paramétrée) ;

  b/ on peut compiler les définitions de fonctions génériques, tout en conservant les possibilités de liaisons et de vérifications statiques lors d'un appel.

Il faut cependant noter que la généricité structurale n'est possible que s'il existe des fonctions génériques au sens incrémental. Il convient donc de donner un cadre de définition pour ces dernières qui réponde aux critiques formulées (cf. III.1.1.1.). Nous verrons comment la notion de propriété (cf. III.2.) résoud élégamment ce problème.

III.1.2. LA GENERICITE DES OBJETS

La généricité structurale pour les fonctions consiste, nous l'avons vu, à passer un (ou plusieurs) types(s) en paramètre.

On peut noter à ce propos que le type en question est celui d'éléments d'une structure élaborée comme celle de tableau dans l'exemple de Reduc. Il est facile de voir que cette forme de généricité, en général, est très liée à celle des structures de données manipulées par ces fonctions.

Ce paragraphe a pour but de présenter les types abstraits génériques, notion qui va permettre d'étendre notablement les possibilités des types abstraits et qui servira de cadre naturel à la définition de fonctions génériques au sens structural.

III.1.2.1. Les constructeurs

L'idée de paramétrer des définitions de types par d'autres types n'est pas neuve. Des langages comme ALGOL 68 ou PASCAL en donnent une première approche au moyen des constructeurs, mécanisme d'extension dont les caractéristiques principales sont:

1/ Les constructeurs de base sont en nombre fini:
par exemple en ALGOL 68: row, struct, proc, union, ref.
Leur choix traduit l'influence de grands groupes d'usagers, et reste, dans le cas d'ALGOL 68, très lié aux contraintes de représentation en machine.

2/ Le mécanisme des constructeurs procède de façon strictement hiérarchique: on construit un type à partir d'autres types eux-mêmes construits à partir d'autres types, etc... jusqu'aux types de base:
exemple en ALGOL 68:

mode département = struct (ent no, nb arrondissement,
[1:7] struct (chaîne cheflieu, ent population) arrondissement) ;

Le mode départment est construit par composition des constructeurs struct et row paramétrés par les modes de base ent et chaîne.

3/ Le répertoire d'opérations s'appliquant aux objets d'un type défini à l'aide de constructeur est figé et complètement déterminé par le ou les constructeurs utilisés.
exemple: après la déclaration
département isère ;
on peut écrire
(arrondissement de isère) [1] := ("grenoble",350 000) ;

L'accès au premier élément du tableau qui est le deuxième champ de la structure repérée par isère se fait par composition, des fonctions d'accès aux structures (sélecteur de champs) et des fonctions d'accès aux tableaux (indexation).

Notons au passage que ces fonctions d'accès sont génériques au sens que nous avons défini précédemment (III.1.1.2. La généricité structurale). L'indexation des tableaux, par exemple, est paramétrée par le type des éléments du tableau.

4/ Réciproquement, dans un langage comme PASCAL, la seule opération exigée sur les types passés en paramètres d'un constructeur est l'affectation (contrairement à ALGOL 68 où cette opération est associée au constructeur ref). Ceci découle du fait que les seules opérations associées aux constructeurs sont des accès à la structure que l'on définit.

Nous allons développer ce qui précède en considérant le constructeur tableau. Ce constructeur permet de regrouper sous un même nom une collection d'objets de même type tout en autorisant un accès direct aux éléments grâce au mécanisme d'indexation.

Traditionnellement, le répertoire d'actions disponibles pour les objets de type tableau est un peu noyé dans le "sucre syntaxique", reflet de la notation mathématique utilisée pour les vecteurs, les matrices, et, plus généralement, pour les séquences. On peut cependant isoler un accès privilégié, l' indexation, qui permet de repérer un élément de tableau pour, soit en modifier la valeur, soit simplement y accéder.

De plus, si on autorise que les éléments d'un tableau puissent être eux-mêmes des tableaux, il est alors nécessaire (remarque 4 précédente) de définir l'affectation entre tableaux.

Ces deux fonctions (indexation et affectation) sont prédéfinies dans le langage, il en est d'ailleurs de même pour la représentation qui reste complètement cachée à l'utilisateur ; ceci impose évidemment que leurs définitions soient paramétrées.

Ainsi la fonction d'indexation effectue un calcul bien défini pour fournir un repère d'élément, ce calcul est paramétré par, le tableau, la taille de ses éléments, les bornes inférieure et supérieures et évidemment la valeur de l'index. Il est bien clair que, du point de vue de l'utilisateur seul le tableau et la valeur de l'index sont significatifs (ce que traduit, d'ailleurs, la notation T(I)) , les autres paramètres sont "superflus" et n'ont pas à être fournis à l'appel de la fonction. Ceci n'est rendu possible que si la donnée du tableau contient implicitement tous ces attributs. Cette facilité de notation pour les fonctions génériques au sens structural va être approfondie dans le paragraphe suivant.

III.1.2.2. Les types abstraits génériques

Un des points important qui ressort du paragraphe consacré aux constructeurs est que ces derniers sont, dans un langage donné, en nombre fini.
C'est dire que si des mécanismes d'extension existent dans le langage, ils ne peuvent en aucun cas s'appliquer aux constructeurs.

Nous avons vu également la forte analogie existant entre la notion de constructeur et celle de type abstrait, en mettant en évidence la nécessité pour ces derniers d'admettre des paramètres qui puissent être des types.

Disposant d'un tel mécanisme, il devient alors possible d'étendre les constructeurs d'un langage, non pas en les composant de façon hiérarchique, mais en créant de nouveaux constructeurs définis par un ensemble de fonctions d'accès originales.

III.1.2.2.1. Définition

Nous appellerons types abstraits génériques [BER77a] un type abstrait admettant un ou plusieurs types comme paramètres de sa définition. Une telle spécification définit en fait toute une famille de types dont chaque membre est caractérisé, dans une occurrence de définition , par la donnée de son nom générique et celle de ses paramètres effectifs. De la même façon qu'un constructeur, un type abstrait générique est un générateur de types. Nous donnons ci-dessous l'exemple d'une telle spécification, la syntaxe du langage utilisé n'étant pas définitivement fixée, on reconnaîtra des traits particuliers à PASCAL et ALGOL 68, cependant, chaque fois qu'une construction nouvelle est introduite, nous en expliquons la signification.

Nous reprenons l'exemple utilisé dans le chapitre II, en faisant du type des éléments de l'ensemble, un paramètre de la spécification.

```
(1)  type petitensemble (type t) ;
(2)       rep [1:100] t finrep ;
(3)       fonction insérer (petitensemble (t) pe , t elem) ;
                    :
(4)       fonction enlever (petitensemble (t) pe, t elem) ;
                    :
(5)       fonction appartient (petitensemble (t) pe, t elem)retourne bool ;
                    :
(6)  fintype ;
```

Une définition de type abstrait générique est donc contenue dans un module délimité par les mots clefs type et fintype. L'en-tête de définition (ligne 1)

donne le nom générique, petitensemble, ainsi que la liste des paramètres formels, ici le seul paramètre formel est t,type des éléments de l'ensemble. Nous trouvons ensuite (ligne 2) la représentation des objets du type que l'on définit; dans notre cas il s'agit d'un tableau de 100 éléments, évidemment tous de type t ; la représentation est inscrite entre les mots-clefs rep et finrep. Comme dans les langages du chapitre II, cette représentation n'est connue qu'à l'intérieur du module de définition et inaccessible de l'extérieur. Viennent ensuite les fonctions (lignes 3, 4 et 5) associées au type, nous n'en avons donné que l'en-tête où l'on retrouve, bien sûr, le type t des éléments. Ces fonctions sont génériques au sens structural, leur définition est partagée par toute une classe de type .

Ainsi, après l'occurrence de définition:

petitensemble (ent) si ;

La liaison des paramètres détermine complètement la fonction insérer, par exemple:

fonction insérer (petitensemble (ent) pe, ent elem. ;

Cette liaison, afin d'assurer la vérification des types à la compilation (ce qui est rappelons le, le but que nous poursuivons) doit être entièrement statique. C'est-à-dire qu'on s'interdit de considérer un type comme une valeur à part entière du langage, en fait les seules expressions de types que l'on se permet de manipuler sont des constantes qui apparaîtront dans les occurrences de définition comme paramètres effectifs de ces dernières.

D'autre part, il est important de noter, qu'après cette liaison, les fonctions génériques du type en question sont complètement déterminées. Par exemple, après les occurrences de définition:

petitensemble (ent) si ;
petitensemble (bool) sb ;

On disposera d'abord de deux représentations ([1:100]ent et [1:100] bool), mais aussi de deux interprétations différentes de la fonction appartient, respectivement associées aux types petitensemble(ent) et petitensemble(bool).

Ces deux interprétations ne diffèreront, en fait, que par les fonctions qu'elles utilisent sur les objets de type ent ou bool, c'est-à-dire les objets du type formel t de la définition générique.

On touche là un des points importants concernant les types abstraits génériques, celui de la compatibilité d'un type, paramètre effectif d'une occurrence de définition avec les fonctions définissant le type générique. Afin d'illustrer schématiquement ce point, revenons sur l'exemple de la fonction appartient. Cette fonction, afin d'assurer l'appartenance d'un objet à un ensemble, devra tester l'égalité de cet objet avec les éléments de l'e semble.Ceci impose que l'opération d'égalité soit définie entre deux objets du typ Nous reviendrons de façon plus détaillée sur ce problème dans la suite de ce chapitre, pour le moment nous nous intéresserons uniquement aux types abstraits considérés comme extension de la notion de constructeur.

Nous avons vu (cf. III.1.2.1.) que la seule contrainte imposée sur les types passés en paramètre de constructeurs est qu'ils implantent une opération d'affectation, de la même façon les seules opérations introduites par un constructeur sont des opérations d'accès à la structure que l'on définit.

Nous allons voir dans le paragraphe suivant un exemple complet de définition d'un nouveau constructeur, via les types abstraits génériques.

III.1.2.2.2. Exemple: la pile

Cet exemple, désormais classique dans la littérature consacrée aux types abstraits, va nous permettre d'esquisser une première approche de la généricité des types.

Tout d'abord,il convient de remarquer qu'une pile est seulement un moyen particulier de structurer un ensemble d'objets et d'y accéder ; ceci correspond donc tout à fait à l'idée d'extension de constructeurs telle que nous l'avons développée précédemment.

Les fonctions primitives associées aux piles sont bien connues, nous choisirons d'implanter: empiler, dépiler, sommet qui délivre la valeur de l'élément du sommet de pile et le prédicat pilevide. En ce qui concerne la représentation, plusieurs possibilités sont envisageables (tableau, liste, ...), cependant nous supposerons que dans le langage utilisé, l'objet composé de base est la séquence de longueur variable. Nous en donnons ci-dessous une spécification semblable à celle des types abstraits génériques sans toutefois y inclure la représentation des objets ni le corps des fonctions puisqu'il s'agit d'un constructeur primitif.

type seq (type t) ;

fonction bsup (seq(t)s) retourne ent ;
co valeur de l'indice maximum de la séquence, initialement zéro co

fonction ote (seq(t)s, ent i) ;
co enlève, s'il existe, le ième élément de s co

fonction ajoute (seq(t)s, t e) ;
co ajoute l'élément e en bout de séquence co

fonction elem (seq(t)s, ent i) retourne t ;
co délivre la valeur du ième élément de s, s'il existe co

fonction range (seq(t)s, ent i, t e) ;
co range la valeur e dans le ième élément de s, s'il existe co

fintype ;

Nous pouvons maintenant donner la définition du type abstrait générique pile.

```
    type pile (type t) ;
(1)   rep seq(t) finrep ;
(2)   fonction empiler (pile(t)p, t val) ;
(3)       ajoute (rep:p, val) ;
(4)   fonction dépiler (pile(t)p) retourne t ;
          début
            tv ;
            v:=elem (rep:p, bsup(rep:p)) ;
            ote (rep:p, bsup(rep:p)) ;
            v
          fin dépiler ;
(5)   fonction sommet (pile(t)p) retourne t ;
          elem (rep:p, bsup(rep:p)) ;
(6)   fonction pilevide (pile(t)p) retourne bool ;
          bsup (rep:p) = 0 ;
    fintype ;
```

La représentation des objets de type pile est définie ligne (1). L'utilisation du constructeur seq assure la généricité de cette représentation via le paramètre t. Les fonctions associées à la pile (lignes 2, 4, 5 et 6) sont spécifiées à partir des fonctions définies sur les séquences de longueur variable.

On peut noter (ligne 3), l'utilisation du forceur (au sens ALGOL 68) rep: , en effet les fonctions empiler, dépiler, sommet et pilevide seront utilisées, à l'extérieur ou à l'intérieur du module, (on pourrait par exemple utiliser pilevide pour valider l'appel de dépiler), c'est bien ce que spécifient les entêtes (2), (4), (5) et (6).

Par contre, le corps de ces fonctions utilise, en général, des opérateurs qui s'appliquent sur la représentation, c'est cette conversion, de l'objet abstrait vers sa représentation, qu'indique le forceur rep :

Cet exemple de la pile est une première approche de l'extension des constructeurs via les types abstraits génériques.

Le nouveau constructeur est défini en restreignant l'ensemble des accès possibles à la représentation (les opérations propres à la représentation ne peuvent pas être appliquées directement aux objets du type pile).

La généricité ne pose pas de problème particulier, le constructeur utilisé pour la représentation étant lui-même générique.

Exemple:

pile (ent) is ;
pile (pile (ent)) iss ;
empiler (iss, is) ;
dépiler (sommet (iss)) ;

On voit donc que cet exemple, qui a fait couler beaucoup d'encre, se traite de façon simple et naturelle dans le cadre des types abstraits génériques.

III.2. LES PROPRIETES

Le mécanisme de définition des types abstraits génériques, proposé pour étendre la notion de constructeur dans les langages de programmation, conduit, comme nous l'avons vu, à spécifier un ensemble de fonctions elles-mêmes génériques. Nous prendrons comme exemple, pour illustrer notre propos, la définition d'un type T dans laquelle les fonctions ne sont caractérisées que par la donnée du type de leurs paramètres et celui de leur résultat:

```
type T (type t) ;
    fonction F1: T(t) × int → t
    fonction F2: T(t) × T(t) → T(t)
fintype ;
```

Les fonctions F1 et F2 sont génériques au sens structural. En effet, leur définition est paramétrée par le type t qui est le paramètre de la définition de T. Il a été établi, dans le paragraphe précédent(II.1.1.3),que passer un type en paramètre d'une fonction revenait à passer, de manière explicite ou implicite, un ensemble de fonctions définies sur t (ici, implicitement, il s'agit des fonctions de t utilisées dans les définitions de F1 et F2).

A priori, n'importe quel type peut remplacer t dans une occurrence de définition, les notations formelles de fonctions utilisées dans F1 et F2 sont alors remplacées par leurs homologues du type effectif. Il est alors clair que ces notations formelles de fonctions sont génériques au sens incrémental (elles peuvent de plus être génériques au sens structural si le type passé en paramètre est lui-même générique, mais cette distinction n'est pas fondamentale).

Cette partie a pour but de proposer un mécanisme permettant de résoudre les problèmes de liaison des fonctions (génériques au sens incrémental) passées en paramètres de fonctions génériques au sens structural.

III.2.1. CARACTERISATION DE CLASSES DE TYPES

En considérant un ensemble de définitions de types abstraits génériques on pourra reconnaître des types qui ne sont que des constrúcteurs obtenus à partir des constructeurs de base (nous avons vu l'exemple de la pile, il y en a bien d'autres: liste(t), queue(t), ...). La caractéristique essentielle de ces types est que les fonctions qui les définissent sont toutes des fonctions d'accès à la nouvelle structure, obtenues par composition des fonctions d'accès à la représentation. Dans la mesure où cette représentation peut toujours être ramenée à une composition des constructeurs de base du langage, n'importe quel type peut remplacer le type formel.

Il existe cependant des définitions de types pour lesquelles:

- les fonctions utilisent, sur les objets du type passé en paramètre, des opérations qui ne sont pas des accès. Dans l'exemple,F2 est probablement une de ces fonctions, définissant une composition interne, dans I elle utilise certainement une composition interne dans t.
- du même coup, n'importe quel type ne peut pas être substitué au paramètre formel.

Ce qui revient à dire que les fonctions définies dans un type abstrait générique ne seront valides que si les types, substitués aux paramètres formels dans une occurrence de définition, appartiennent à un certain sous-ensemble de types.

Afin de clarifier ce dernier point, nous allons examiner quelques exemples:

1/ Considérons le type liste dont les objets sont des listes linéaires où le type des éléments est quelconque.

type liste (type t) ;

Supposons qu'on veuille ajouter à la définition un prédicat qui teste l'appartenance d'un élément à une liste.

fonction existe (liste(t) l, t elem) retourne bool ;

L'algorithme de cette fonction utilise une opération d'égalité sur les objets de type t.

2/ Soit le type vecteur, paramétré par le type de ses éléments. On veut pouvoir définir l'addition entre deux vecteurs.

fonction v+(vecteur(t) v1, v2) retourne vecteur (t) ;

Il est clair que l'addition entre vecteurs est définie à partir de l'addition entre ses éléments. Le type t doit donc disposer d'une opération "d'addition". On peut noter que v+ dépend du + défini sur t. S'il s'agit de vecteur d'entiers, c'est l'addition entre entiers qui est utilisée ; si, par contre il s'agit de vecteurs de chaînes de caractères, c'est, par exemple, la concaténation.

3/ On peut définir, à l'aide de ce formalisme, des constructeurs déjà existants dans le langage

type tableau (type t1, t2) ;

t1 est ici le type de l'index, t2 le type des éléments.

t1 est en général un intervalle d'entiers, mais ce peut être également un intervalle construit sur un type discret ordonné, ou encore un produit cartésien d'intervalles. En suivant la terminologie de PASCAL nous dirons que t1 doit être "scalaire" pour que le type tableau soit lui-même défini.

Les exemples précédents montrent que les conditions de validations exprimées sur les types formels sont de natures très diverses, il s'agit de:

- relations (égalités dans les listes),
- opérations internes, reflets d'une structure algébrique particulière (addition des vecteurs),
- existence de fonctions particulières (index de tableaux).

En informatique, où tout est calculé, ces conditions se traduisent en fait par l'existence de fonctions définies dans le type effectif. C'est ainsi que la relation d'égalité se traduira par l'existence d'une fonction à deux paramètres et à résultat booléen. Cette caractérisation syntaxique est évidemment insuffisante pour déterminer, dans un type particulier, quelle est la fonction qui implante le calcul de la relation d'égalité (la relation d'ordre par exemple est implantée par une fonction de fonctionnalité identique). Ceci nous a amenés à définir un cadre pour des déclarations plus précises.

L'idée informelle est la suivante:

- les conditions imposées aux types formels sont appelées des propriétés et sont caractérisées par un nom.

- une propriété est constituée essentiellement d'un ensemble de fonctions dont la fonctionnalité dépend d'un type formel.

- un type particulier possède la propriété p si sa définition contient les fonctions implantant celles de la propriété p.

- un type peut être substitué à un type formel s'il possède au moins les propriétés exigées pour le type formel.

Reprenons l'exemple de la liste et de la fonction existe qu'on voulait lui ajouter.

```
type liste (type t) ;
        :
    fonction existe (liste(t) l, t elem) retourne bool ;
       exige t : égalité (=) ;
           premier y dans l tel que x=y
                alors vrai
                sinon faux
    fin existe ;
fin type
```

existe est définie au moyen de l'itérateur "premier" qui énumère un à un les éléments de la liste et s'arrête sur l'alternative alors dès que la condition est vérifiée et sur sinon lorsque tous les éléments de la liste ont été énumérés.

La condition exigée sur t est l'égalité, la notation pour l'opérateur formel est "=", sa fonctionnalité, prédéfinie pour l'égalité est: t × t → bool. Cette fonctionnalité pour les opérateurs ou fonctions des propriétés permet de faire la vérification des types dans les fonctions génériques.

Sachant que pour le type standard ent, l'égalité est implantée, on peut écrire:

```
liste (ent) li ;
      :
  si existe (li,4) alors ...
```

par contre avec:

```
liste (liste(ent)) lli ;
```

l'instruction:

```
si existe (lli,li) alors
```

bien que correcte du point de vue syntaxique n'est pas valide puisque le type liste (t) n'a pas la propriété d'égalité. Pour remédier à cet inconvénient, il faut définir liste de la manière suivante:

type liste (type t) : égalité (eq-liste) ;

où eq-liste est la fonction, définie dans le type, qui implante la relation d'égalité pour les objets de type liste.

```
fonction eq-liste (liste(t) l1,l2) retourne bool ;
   premier y dans l1 telque ¬existe(l2,y)
      alors faux
      sinon premier z dans l2 telque ¬ existe (l1,2)
            alors faux
            sinon vrai ;
```

On peut noter que la fonction "existe" utilisée dans eq-liste exige la propriété d'égalité pour t, qui sera donc exigée également pour eq-liste. Cette propagation des propriétés est faite automatiquement par le compilateur. L'instruction

si existe (lli,li) alors ...

est maintenant correcte.

Arrivés à ce stade on peut d'ores et déjà dégager deux points de vue complémentaires concernant les types génériques et les propriétés:

1/ Du point de vue de la généricité

- un type générique définit une famille de types abstraits, c'est-à-dire des familles de fonctions génériques au sens structural.
- une propriété définit une famille de types (génériques ou non), c'est-à-dire des familles de fonctions génériques au sens incrémental.

2/ Du point de vue de la vérification

- un type caractérise un ensemble d'objets pouvant être paramètres de certaines fonctions.
- une propriété p caractérise une classe de types pouvant être paramètre de types dans les fonctions qui exigent p.

Avant de continuer, on peut citer les travaux les plus récents ayant pressenti la nécessité des propriétés sous l'un ou l'autre de ces aspects:

On trouve dans [BOO76] l'idée de fournir localement, c'est-à-dire dans la fonction que l'on écrit, une spécification syntaxique des opérateurs formels associés à un type passé en paramètre, ceci afin d'accroître la souplesse du langage tout en permettant au compilateur d'effectuer un minimum de vérifications statiques.

Parallèlement à nos premiers travaux [BER77a], Tennent [TEN77] proposait une extension au langage PASCAL en considérant trois grandes catégories prédéfinies de types:

- les class caractérisent les types dont la représentation peut être quelconque, étant entendu qu'aucune fonction de cette dernière n'est héritée par le nouveau type.
- les data dont les éléments disposent d'une représentation qui possède l'affectation (et cette opération est transmise au nouveau type).
- les index, la représentation possède l'indexation qui est, là aussi, transmise au nouveau type.

Comme on le voit, l'idée de propriété est, dans cette approche, très liée à la caractérisation du "degré" de transmission des fonctions d'une représentation vers le type qu'elles représentent.

Avec ces travaux, on peut citer également [LAM77], [MIT77], [IRO77] qui tous, et à des degrés divers, reprennent ces idées de paramétrisation sans toutefois assurer les vérifications des définitions génériques.

III.2.2. DEFINITION DES PROPRIETES

La définition des propriétés a, comme nous allons le voir, de très fortes analogies avec la définition des types abstraits génériques. Une propriété est caractérisée par son nom, un type formel et des spécifications de fonctionnalités.

Par exemple, la propriété d'égalité, déjà évoquée, sera définie de la manière suivante:

```
prop égalité (type t) ;
   opérateur =: t × t → bool ;
finprop ;
```

et la propriété monoïde:

```
prop monoïde (type t) ;
   opérateur + : t × t → t ;
   constante 0 : → t ;
finprop ;
```

On a défini dans ces deux propriétés des opérateurs, qui, dans un type possédant cette propriété, pourront être implantés par des fonctions, la distinction entre les deux étant purement syntaxique (notation infixée dans un cas, fonctionnelle dans l'autre). La constante 0 de la propriété monoïde est considérée comme une fonction (ou un opérateur) sans paramètre et à résultat dans le type formel t.

L'analogie avec la définition des types abstraits génériques nous oblige à donner quelques précisions. L'objectif des propriétés n'étant pas, comme pour les types, de définir des ensembles d'objets, la notion de représentation n'intervient évidemment pas. La définition des fonctions se limite donc à la donnée de leur fonctionnalité. D'autre part, l'idée de protection, fondamentale dans la définition des types abstraits génériques, disparaît au niveau des propriétés. Un type peut très bien posséder une propriété et donc implanter toutes les fonctions de celle-ci, mais il peut également en implanter bien d'autres.

En d'autres termes les fonctions attachées au type pile par exemple ne peuvent pas constituer une définition de propriété. S'il en était ainsi le type liste pourrait alors (comme n'importe quel autre type) posséder cette propriété et donc implanter les fonctions empiler, dépiler, sommet et pilevide ; mais, disposant également de fonctions propres aux listes telles que insérer, enlever qui peuvent opérer directement dans la liste, l'intégrité des accès à la pile ne serait pas assurée.

On peut donc dire que si toute propriété, moyennant la donnée d'une représentation et des corps de fonctions, peut être considérée comme un type abstrait générique ; l'inverse n'est pas toujours vrai. Ce qui traduit bien le fait que dans un cas (celui des types), on s'intéresse à la spécification d'un ensemble d'objets caractérisés par les fonctions qui les manipulent, alors que dans l'autre cas (celui des propriétés) on s'intéresse à la spécification d'un ensemble de fonctions caractérisant une classe de types.

Ainsi le fait pour un ensemble d'objets d'être considéré comme "scalaire" est plus une propriété attachée à un type, qu'un type lui-même. Le fait d'être scalaire indépendamment de toute représentation, est caractérisé par un ensemble minimum de fonctions comme le traduit la définition de propriété suivante:

```
prop scalaire (type t) ;
        constante prem :  → t ;
        constante der  :  → t ;
        fonction suc : t → t ;
        fonction pred : t → t ;
        constante card :  → ent ;
        fonction ord : t → intervalle(ent,1,card) ;
finprop ;
```

III.2.2.1. Fonctions fondamentales et fonctions dérivées

L'idée de n'avoir à spécifier, dans une propriété, qu'un ensemble minimum de fonctions, amène à penser que de nouvelles fonctions peuvent être construites à partir de cet ensemble minimum. Ces définitions complètes apparaissant dans une propriété p sont valables pour tous les types qui implantent p sans avoir à être redéfinies pour chaque type. Les fonctions qui doivent obligatoirement être définies dans le type sont appelées fonctions fondamentales ; les fonctions qui peuvent être définies entièrement dans les propriétés sont appelées fonctions dérivées.

Par exemple, à l'aide de la spécification de l'opérateur = de la propriété d'égalité (cf. III.2.2.) on est capable de définir un opérateur ¬ = par:

```
opérateur ¬ = (t  x,y) retourne bool ; ¬ (x=y) ;
```

Cette dérivation est évidemment très simple, voyons maintenant un exemple complet de définition de propriétés avec les fonctions fondamentales et les fonctions dérivées:

```
prop ordre total (type t)
        opérateur  <= : t → t → bool ;

   derive
        opérateur > (t x,y) retourne bool ; ¬ (x <=y);
        opérateur = (t x,y) retourne bool ; (x <=y) et (y <=x) ;
        opérateur >= (t x,y) retourne bool ; (x > y) ou (x = y) ;
        opérateur < (t x,y) retourne bool ; ¬ (x >=y) ;
        opérateur ¬= (t x,y) retourne bool ; ¬ (x = y) ;
finprop ;
```

Si le type date par exemple, doit posséder la propriété ordre-total avec les fonctions antérieur, postérieur-strict, même-jour, postérieur, jour-différent, il suffit dans ce type d'implanter la fonction:

fonction antérieur (date d1,d2) retourne bool ;

pour que les autres fonctions soient automatiquement dérivées.

Par souci d'efficacité, on peut redéfinir certaines fonctions dérivées pour un type donné ; la nouvelle définition remplace alors la fonction dérivée standard.

La possibilité d'avoir des fonctions fondamentales de propriétés et des fonctions dérivées pose un problème de choix qui n'est pas simple. En effet, on n'a, en général, aucun moyen formel de décider quelles fonctions sont fondamentales et quelles fonctions sont dérivées puisque l'ensemble fondamental peut ne pas être unique pour une propriété.

Ce problème est à rapprocher de celui de la définition d'un ensemble de fonctions à l'intérieur d'un type abstrait générique. On doit,dans ce cas, définir un ensemble "de base" à partir duquel, dans un programme donné, on définira des actions plus complexes. De la même manière on supposera que l'on est capable, pour chaque propriété, de décider quel est l'ensemble de fonctions fondamentales (resp : dérivées) "le plus naturel", étant entendu que l'on aura toujours une seule façon de dériver une fonction donnée.

Ceci implique que si l'on définit l'opérateur $\neg=$ comme dérivé de <= par la formule:

opérateur $\neg=$ *(t x,y) retourne bool ;* $(x < y) \vee (y < x)$ *;*

on ne peut avoir le même opérateur à partir de = par $\neg$ (x=y). Dans ce cas, il faudrait avoir recours à la démonstration formelle pour s'assurer que les deux définitions sont équivalentes. Le même raisonnement conduit à interdire d'avoir une même fonction formelle définie dans deux propriétés distinctes (ex.: l'opérateur = dans les propriétés d'égalité et d'ordre total). Le paragraphe suivant ébauche une solution à ce problème.

III.2.2.2. Inclusion de propriétés

Pour résoudre le dernier problème évoqué et permettre les liaisons des appels formels en conservant une certaine souplesse au langage, il est nécessaire d'introduire la notion d'inclusion de propriété.

Définition:

Soient p1 et p2 deux propriétés, on dira que p1 est incluse dans p2 (noté p1 ⊂ p2) si et seulement si toutes les fonctions de p1 sont définies dans p2.

Dans cette définition il peut s'agir des fonctions fondamentales ou dérivées de p2. Par exemple:

égalité$(=, \neg=) \subset$ *ordre_total* $(<=, >, =, >=, <, \neg=)$

La position des fonctions est importante dans cette notation, une autre manière d'écrire est:

égalité(1,2) ⊂ *ordre_total(,,1,,,2)*

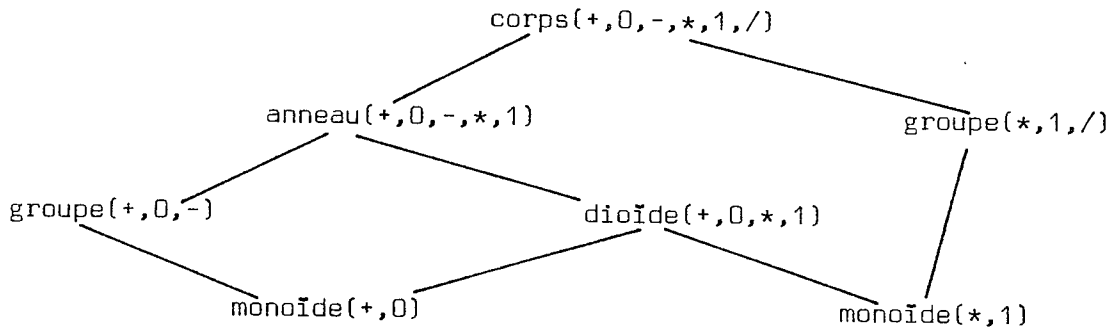
La relation d'inclusion est évidemment transitive, elle peut se définir à l'aide de schémas d'inclusion portant sur les noms des fonctions comme le montre l'exemple ci-dessous:

anneau (f1,f2,f3,f4,f5) ⊂ *corps(f1,f2,f3,f4,f5)*
groupe (f1,f2,f3) ⊂ *corps (,,, f1,f2,f3)*
dioïde (f1,f2,f3,f4) ⊂ *anneau (f1,f2, ,f3,f4,)*
groupe (f1,f2,f3) ⊂ *anneau (f1,f2,f3,,)*
monoïde(f1,f2) ⊂ *groupe (f1,f2,)*
monoïde(f1,f2) ⊂ *dioïde (f1,f2,,)*
monoïde(f1,f2) ⊂ *dioïde (,,f1,f2).*

A l'aide de ces schémas d'inclusion on est capable de déterminer toutes les propriétés introduites par la déclaration:

*corps (+, 0, -, *, 1, /)*

Sous forme de diagramme on a:



De manière intuitive, ce schéma peut s'interpréter de la façon suivante: quand dans une définition de type abstrait générique on exige pour le paramètre de type d'avoir la propriété dioïde, on peut passer en paramètre effectif un type ayant cette propriété ou une propriété plus forte (anneau, corps).

Dans le cas où on exige la propriété monoïde et que c'est un dioïde qui est paramètre effectif, on peut alors le considérer comme monoïde de deux façons. Nous verrons dans le paragraphe sur la liaison des appels formels comment lever cette ambiguité.

Si on veut pouvoir disposer de la souplesse offerte par ce mécanisme, il convient donc lors de la définition de nouvelles propriétés (et si cela est nécessaire), de les combiner par des schémas d'inclusion aux propriétés déjà existantes. Les seules vérifications que peut faire alors le compilateur sont d'ordre syntaxique (identité des fonctionnalités).

III.2.2.3. Définition de types avec propriétés

Afin qu'une propriété puisse caractériser une classe de types, il est nécessaire lors de la définition d'un nouveau type, d'associer à celui-ci une ou plusieurs propriétés.

Comme nous l'avons vu, le type doit, pour posséder la propriété, implanter les fonctions fondamentales de cette dernière. Le lien entre les fonctions du type et celles de la propriété se fait de la manière suivante:

Soit la propriété p, contenant n fonctions

prop p (*type* t)
  fonction f1 : $\phi_1(\underline{t}) \rightarrow \phi_1^r$;
  fonction f2 : $\phi_2(\underline{t}) \rightarrow \phi_1^r$;
  ⋮
  fonction fn : $\phi_n(\underline{t}) \rightarrow \phi_n^r$;
finprop ;

$\phi_i(\underline{t})$ désigne le type des paramètres (dont un, au moins, dépend du type formel $\underline{t}$), $\phi_i^r(\underline{t})$ désigne le type du résultat qui dépend en général lui aussi de $\underline{t}$, mais qui peut également être un type de base (bool pour l'égalité par exemple).

Si le type abstrait générique τ doit posséderla propriété p, on le définira par:

type $\underline{\tau}(...)$: $p(g_1, g_2, \ldots g_n)$;
  ⋮
  fonction $g_1(\phi_1(\underline{\tau}(...)))$ *retourne* ϕ_1^r ;
  ⋮
  fonction $g_n(\phi_n(\underline{\tau}(...)))$ *retourne* ϕ_n^r ;
fintype ;

On peut noter que c'est le nom des fonctions de $\underline{\tau}$ qui apparaît dans la liste de p, le lien avec les fonctions de p se fait donc de manière positionnelle. Cette facilité permet une plus grande souplesse pour les noms des fonctions de propriété, par exemple l'opérateur + de la propriété monoïde s'appellera bien + pour les entiers mais ou pour les booléens et conc pour les chaînes de caractères.

III.2.3. LIAISONS DES APPELS FORMELS

Soit une fonction f définie dans un type abstrait générique $\underline{\tau}(\underline{t})$, dans laquellle on souhaite utiliser une fonction sur $\underline{t}$. Cette fonction doit obligatoirement être caractérisée par une propriété p, dans la liste de fonctions de laquelle elle apparaît (les autres fonctions étant ou non utilisées, dans f).

exemple: p(g)

L'appel de g dans f sera qualifié de formel dans la mesure où g, au moment de la définition de f dans $\underline{\tau}(\underline{t})$, n'est connue qu'au travers de la propriété p. f n'est alors définie que pour les objets $\underline{\tau}(\underline{t})$ dans lesquels $\underline{t}$ possède au moins la propriété p.

Par exemple, on demande à un type formel d'être scalaire si l'on a besoin de considérer les valeurs de ce type comme une suite finie ordonnée. C'est ce que traduira l'emploi de certaines fonctions comme premier, dernier, successeur, etc...

De même, si dans le type matrice($\underline{t}$) on veut définir le produit de deux matrices, on exigera, pour $\underline{t}$, d'avoir la propriété minimale dioïde puisqu'on utilisera les opérations + et * avec éventuellement leurs éléments neutres respectifs.

Les conditions sur les paramètres formels sont décrites par des instructions exige (cf. III.2.1.) qui ont essentiellement trois fonctions:

- caractériser l'ensemble des types qui peuvent être liés à $\underline{t}$,
- déclarer les noms de fonctions qui seront utilisés dans les appels formels,
- permettre de lier les noms de fonctions des appels formels aux fonctions effectives d'un type particulier possédant la propriété p.

III.2.3.1. Liaisons implicites

Reprenons l'exemple du type liste (cf.III.2.1.) pour compiler l'expression:

si existe(lli,li) alors ...

On voit que le domaine de l'appel de existe est:

ϕ_{appel}: *liste (liste(ent))* × *liste (ent)*

a/ existe est définie dans le type liste avec la fonctionnalité:

liste(t) × *t* → *bool*

Le résultat de l'appel de existe est donc de type bool et le paramètre formel t est substitué par liste(ent).

b/ on exige pour t (liste (ent)) la propriété d'égalité avec l'opérateur noté = . Or le type liste(t) a la propriété égalité avec la fonction notée eq-liste. L'appel formel de = (dans existe) est donc lié à l'appel de la fonction eq-liste.

c/ eq-liste a la fonctionnalité

liste(t') × *liste(t')* → *bool*

et elle est appelée avec

liste(ent) × *liste(ent)*

Son résultat est donc de type bool et t' est substitué par ent.

d/ eq-liste appelle existe de fonctionnalité d'appel liste(t) × t où t est substitué par ent. L'instruction exige est satisfaite puisque ent a la propriété d'égalité dont l'opérateur est standard (il n'appelle aucune autre fonction). L'appel formel de = est maintenant lié à cette opération sur les entiers.

Le paragraphe suivant (III.3.) détaille le mécanisme de liaison entre un appel formel et une fonction particulière via les propriétés ; il étudie en particulier les conditions sous lesquelles cette liaison est possible.

On peut noter ici, que la définition de la fonction existe est utilisée récursivement à cause du paramètre dont le type est lui-même construit récursivement (liste (liste (...))) et que, d'autre part, la validation de la propriété est aussi récursive. En effet, c'est parce que ent a la propriété égalité (=) que liste(ent) a la propriété égalité (eq-liste).

La liaison entre = et eq-liste se fait parce que la propriété du type liste et celle exigée pour t est la même (l'égalité). Cette liaison peut se faire également si la propriété exige est incluse dans la propriété du type passé en paramètre.

III.2.3.2. Liaisons explicites

Lorsque, dans un programme, on appelle explicitement une fonction dans laquelle sont écrits des appels formels, on peut vouloir forcer les liaisons pour une des trois raisons suivantes:

a/ modifier le résultat attendu de la liaison implicite ;

b/ définir une nouvelle liaison ;

c/ lever une ambiguité introduite par l'inclusion de propriété.

Considérons le type vecteur(t,m) auquel on veut ajouter une opération d'addition. La structure minimum requise pour le type formel t est celle de monoïde, il est en effet nécessaire de disposer, d'une opération d'addition sur les éléments, et d'un élément neutre.

Soit v+ la fonction:

```
fonction v+ (vecteur(t,m)v1,v2) retourne vecteur (t) ;
   exige t : monoïde (+,0) ;
   début
      vecteur (t,m) v ;
      pour i:=1 jusqua m
         faire ranger(v,i,acces(v1,i) + acces(v2,i)) finfaire ;
   v
fin v+ ;
```

Soit maintenant la déclaration

```
vecteur (réel, 30) vx, vy ;
```

l'expression v+ (vx,vy) n'est pas définie puisque à partir du corps des réels il y a deux monoïdes possibles, donc deux opérations, définies sur les réels qui peuvent être liées à +.

L'instruction permettant de forcer les liaisons a la forme suivante:

<appel> avec <type> : <nom de propriété>(<liste de fonctions définies dans <type>>)

Pour que l'appel précédent soit correct, on peut écrire:

v+ (vx,vy) avec réel : monoîde (+,0) ;

ou encore:

v+ (vx,vy) avec réel : monoîde (,1) ;*

Toutefois cette forme de l'instruction d'appel peut également servir à redéfinir les fonctions qui confèrent à un type donné une propriété particulière.

Considérons le type matrice utilisé dans un contexte de traitement de graphes de relations:

type matrice (type t ; ent n) ;

Dans cette définition nous trouvons, par exemple, la fonction Warshall, qui calcule la fermeture transitive de la relation

fonction Warshall (matrice (t,n)m) ;
exige t : treillis-distributif (+,b,,v) ;*

(b est l'élément neutre de +, absorbant pour *; v est l'élément neutre de *).

Considérons les déclarations:

matrice (bool,n) mb ;
matrice (réel,n) mr ;

où mb(i,j) = vrai signifie qu'il existe un chemin entre x et y et où la longueur de ce chemin est mr(i,j).

On peut alors écrire:

```
Warshall (mb) ;
Warshall (mr) avec réel : treillis distributif
  (fonction (réel a,b) retourne réel ;
      si a <= b alors a sinon b finsi ,
   maxréel ,
   fonction (réel a,b) retourne réel ;
      si (a = maxréel) ou (b = maxréel)
      alors maxréel sinon a+b finsi ,
   0.) ;
```

Ce dernier appel est un exemple de caractérisation de la propriété d'un type donné au moment de l'appel d'une fonction.

III.3. LES PROBLEMES DE VERIFICATION

Dans le premier paragraphe de ce chapitre, la présentation des types abstraits génériques a été faite de façon relativement informelle. Cette démarche convenait pour exprimer un certain nombre d'idées dans le cadre d'une définition "intuitive" de langage.

En abordant les problèmes de vérification inhérents au mécanisme, la nécessité de préciser de manière plus formelle l'ensemble engendré par les types abstraits génériques se fait sentir. C'est le but de la première partie de ce paragraphe de donner les définitions nécessaires et de montrer que les types abstraits génériques forment un système de production assimilable aux langages de premier ordre. Notre but n'est évidemment pas d'étudier en détail ce langage.

Nous nous cantonnerons à introduire les notions nécessaires, d'une part à sa caractérisation et d'autre part aux manipulations utilisées dans les algorithmes de vérification que nous étudierons dans la suite. Le formalisme utilisé s'inspire largement de celui que l'on trouve dans [HUE76].

III.3.1. Définitions

III.3.1.1. Caractérisation du langage T

Nous considérerons deux ensembles primitifs:

- l'ensemble TF des types formels (notés $\underline{t}$, $\underline{t}_i$, $\underline{t}_2$,...,$\underline{t}_i$) introduit par les définitions génériques ; il constitue un ensemble de variables.
- l'ensemble NT des noms de types qui est assimilé à un ensemble de symboles de fonctions constantes. Chaque élément de NT est affecté d'une arité n positive ou nulle. On notera $\alpha(\tau)$ l'arité de $\underline{\tau}$ élément de NT. Les types de base: ent, bool, ... ont évidemment une arité nulle.

Chaque définition de type abstrait générique ajoute donc un élément dans NT et un ou plusieurs éléments dans TF.

Si on note T ce langage:

1/ $TF \subset T$

2/ Quelque soient $e_1, e_2, \ldots, e_{\alpha(\underline{\tau})}$ éléments de T et $\tau \in NT$
alors $\underline{\tau}(e_1, e_2, \ldots, e_{\alpha(\tau)}) \in T$

Ce langage contient donc tous les types que l'on peut construire à partir des définitions de types abstraits génériques et des types de base (on ne prend pas en compte les contraintes, imposées par les propriétés, qui évidemment restreignent T).

Nous appellerons termes les éléments de T.

Nous noterons V(e) l'ensemble des variables d'un terme e, et $\nu(e)$ le nombre de ces variables ($\nu(e) = |V(e)|$).

Si $\nu(e)=0$ on dit que le terme e est fermé. Un terme fermé correspond à un type complètement déterminé. En particulier, les types utilisés dans des déclarations de variables (en dehors des définitions de types abstraits génériques) doivent être des termes fermés de T.

III.3.1.2. Substitutions

Nous avons besoin, dans la suite de l'exposé, de caractériser la substitution d'un ou plusieurs types formels (éléments de TF) par un ou plusieurs types effectifs (termes de T). Dans ce paragraphe nous définissons cette notion comme une application de TF dans T égale à l'identité presque partout [HUE76].

On appelle composante de substitution une paire $<\underline{t},e>$ ou $t \in TF$ et $e \in T$.

Une substitution est un ensemble fini de composantes de substitutions se rapportant à des variables distinctes.

$$\sigma = \{<\underline{t}_i, e_i> \mid 1 \leq i \leq n\} \quad \forall i,j \leq n \quad \underline{t}_i = \underline{t}_j \Rightarrow i=j \quad \text{et} \quad \forall i \leq n \; e_i \neq \underline{t}_i$$

Soit σ une substitution. On définit pour tout $\underline{t}$ dans TF, $\sigma\underline{t}$ par:

$$\sigma\underline{t} = \begin{cases} e & si \quad <\underline{t},e> \in \sigma \\ \underline{t} & sinon \end{cases}$$

Cette application est étendue à T par les règles suivantes:

$$\sigma\underline{\tau}(\ldots,e_i,\ldots) = \underline{\tau}(\ldots,\sigma e_i,\ldots) \quad \forall\, \tau \in T$$

exemple:

si $\underline{liste} \in NT$ *et* $\underline{liste}(\underline{t}_1) \in T$

et si $\sigma = \{<\underline{t}_1, \underline{table}(\underline{ent},\underline{t}_2)>\}$

alors $\sigma\ \underline{liste}(\underline{t}_1) = \underline{liste}(\underline{table}(\underline{ent},\underline{t}_2))$

La composition de deux substitutions σ_1, σ_2 notée $\sigma_1\ \sigma_2$ est définie comme la substitution:

$$\sigma_1\ \sigma_2 = \{<\underline{t},\ \sigma_1(\sigma_2\ (\underline{t}))> \mid \underline{t} \in TF\}$$

La composition des substitutions est associative, mais non commutative ; la substitution vide en est l'élément neutre.

Une substitution dans laquelle tous les e_i appartiennent à TF est une substitution <u>formelle</u>, son application à un terme e de T consiste uniquement à renommer certaines variables.

Enfin la restriction de σ à un ensemble F de types formels ($F \subset TF$) est définie par:

$$\sigma \restriction F = \{< \underline{t}_i, e_i > \mid \underline{t}_i \in F\}$$

III.3.1.3. Relations dans T

Egalité:

Deux types e_1 et e_2 , éléments de T, sont égaux ($e_1 = e_2$) si et seulement si les deux chaînes qui les représentent sont égales.

Pré-ordre:

Soient e_1 et e_2 deux éléments de T. On dit que e_1 est plus générique que e_2 (noté $e_1 > e_2$) si et seulement si il existe une substitution σ telle que $\sigma e_1 = e_2$.

Equivalence:

Deux types e_1 et e_2 sont équivalents ($e_1 \equiv e_2$) si et seulement si $e_1 > e_2$ et $e_2 > e_1$.

Remarque:

Si e_1 et e_2 sont équivalents, alors il existe deux substitutions formelles σ_f et σ'_f telles que: $e_1 = \sigma_f e_2$ et $e_2 = \sigma'_f e_1$.

Ordre partiel:

La relation > est un préordre dans T, elle détermine un ordre partiel (noté $\geq$) dans l'ensemble quotient de T par la relation $\equiv$.

exemples:

$pile\ (t_1) \geq pile(ent)$

$ensemble(t_1) \geq ensemble(liste(t_2)) \geq ensemble(liste(table(int, t_3)))$

$t_4 \geq bool$

$matrice\ (t_5) \equiv matrice(t_6)$

Dans la suite c'est l'ensemble ($T/\equiv$) qui désignera l'ensemble des types et on considérera donc, sauf indication contraire, la relation $\geq$.

III.3.1.4. Fonctionnalité

L'ensemble T, tel que nous l'avons défini, ne suffit pas pour caractériser les fonctions génériques. Nous allons être amenés à nous intéresser à l'ensemble des fonctionnalités, noté F , construit à partir de T de la manière suivante:

1/ quelque soit $e \in T$, alors $e \in F$.

2/ si $\Phi \in F$ et $e \in T$, alors $(e \rightarrow \Phi) \in F$.

Remarque: On note $e_1 \times e_2 \times \ldots \times e_n \rightarrow e_{n+1}$ l'expression $(e_1 \rightarrow (e_2 \rightarrow (\ldots (e_n \rightarrow e_{n+1}) \ldots)$ obtenue par applications successives de la règle 2.

Le point important, à souligner dans cette définition séparée de T et F , est que les variables apparaissant dans les éléments de F, prennent leurs valeurs dans T, c'est-à-dire que F, considéré comme langage algébrique, reste du premier ordre.

Soit f une fonction générique, la fonctionnalité de définition de f, notée $\Phi_{def(f)}$, est une expression $e_1 \times e_2 \times \ldots \times e_n \rightarrow e_{n+1}$, où $e_1, e_2, \ldots, e_n$ désignent les types des paramètres et e_{n+1} celui du résultat.

On notera $\Phi_{domaine(f)}$ l'expression $e_1 \times e_2 \times \ldots \times e_n$ et $\Phi_{res(f)}$ le terme e_{n+1}.

On a donc:

$$\Phi_{def(f)} = \Phi_{domaine(f)} \rightarrow \Phi_{res(f)}$$

Remarque: Les types formels apparaissant dans les fonctionnalités de deux fonctions distinctes doivent être considérés comme distincts (au besoin on les renommera).

Par extension, et abus de langage, on dira que $\Phi_{def(f)}$ est le type de définition de f.

Soit $f(x_1,x_2,\ldots,x_n)$ un appel de la fonction f et $e_1,e_2,\ldots,e_n$ les types des paramètres:

$$e_1 \times e_2 \times \ldots \times e_n \text{ est noté } \Phi_{appel(f)}$$

Les substitutions et les relations introduites sur T et sur $(T/\equiv)$ peuvent être étendues à F de manière évidente:

Soit $\Phi = (e_1 \to (e_2 \to (\ldots(e_n \to e_{n+1})\ldots)))$ un élément de F
alors

$$\sigma\Phi = (\sigma e_1 \to (\sigma e_2 \to (\ldots(\sigma e_n \to \sigma e_{n+1})\ldots)))$$

et si Φ_1 et Φ_2 appartiennent à F alors:

$\Phi_1 \geq \Phi_2$ si et seulement si il existe une substitution σ (non formelle ni vide)telle que $\sigma\Phi_1 = \Phi_2$.

III.3.2. Résolution des appels explicites

III.3.2.1. Appels explicites

Définition:

Soit $f(x_1,x_2,\ldots,x_n)$ un appel de fonction dont le type des paramètres est $\Phi_{appel(f)} = e_1 \times e_2 \times \ldots \times e_n$. $f(x_1,x_2,\ldots,x_n)$ est un appel explicite si et seulement si il existe une définition de fonction de même nom f dans un type τ et de fonctionnalité telle que

$$\Phi_{domaine(f)} \geq e_1 \times e_2 \times \ldots \times e_n$$

Exemple:

Soit pile(t) un type générique dans lequel il existe une fonction "empiler" telle que $\Phi_{def(empiler)} = [pile(t) \times t]$.

1/ Avec les déclarations suivantes:

$pile(liste(t_1))\ x\ ;$

$liste(t_1)\ y\ ;$

l'appel empiler (x,y) est un appel explicite puisque

$$[pile(t) \times t] \geq [pile(liste(t_1)) \times liste(t_1)]$$

2/ Par contre, avec les déclarations:

$t_1\ a\ ;$

$liste(t_2)\ b\ ;$

empiler (a,b) ne constitue pas un appel explicite.

III.3.2.2. Non ambiguité des appels explicites

Définition:

L'appel explicite d'une fonction f, avec le type $\Phi_{appel(f)}$ des paramètres est ambigu s'il existe au moins deux définitions distinctes de fonctions ayant le même nom f et les domaines de définition Φ_1 et Φ_2 tels que

$$\Phi_1 \geq \Phi_{appel(f)} \quad \text{et} \quad \Phi_2 \geq \Phi_{appel(f)}$$

Ceci correspond au cas d'une fonction générique au sens incrémental pour laquelle certaines fonctionnalités d'appel ne permettent pas de déterminer une interprétation.

Définition:

Un ensemble de définitions de types abstraits génériques est déterministe par rapport aux appels si et seulement si tout appel possible de fonctions est non ambigue.

Remarque:

Une règle suffisante pour qu'un ensemble de définitions de types soit déterministe par rapport aux appels est que tout nom de fonction soit unique. Cette solution nous semble cependant trop contraignante dans la mesure où elle interdit toute possibilité de généricité incrémentale. La solution apportée par CLU (préfixer le nom de la fonction par le nom du type où elle est définie) nous semble lourde et de toute manière pas adaptée à la généricité qui impose l'homonymie de façon naturelle.

III.3.2.3. Notion de PGSFC

Définition:

Soient Φ_1 et Φ_2 deux éléments de F. La Plus Grande Sous Fonctionnalité Commune de Φ_1 et Φ_2 , notée PGSFC(Φ_1,Φ_2) est définie par les deux propriétés suivantes:

1/ PGSFC (Φ_1,Φ_2) $\leq \Phi_1$ et PGSFC(Φ_1,Φ_2) $\leq \Phi_2$

2/ $\forall\ \Phi'$ telle que $\Phi' \leq \Phi_1$ et $\Phi' \leq \Phi_2$ alors $\Phi' \leq$ PGSFC(Φ_1,Φ_2)

Remarque:

PGSFC(Φ_1,Φ_2) n'est évidemment pas définie pour tout couple Φ_1 Φ_2 et, tous les éléments de la classe PGSFC(Φ_1,Φ_2) sont des PGSFC particulières et équivalentes.

III.3.2.4. Calcul de la PGSFC

Si on reprend la définition de l'extension de la relation $\leq$ aux fonctionnalités, les deux propriétés de la PGSFC s'expriment de la manière suivante:

1/ Il existe une substitution σ telle que $\Phi_1 = \sigma\Phi_2 =$ PGSFC(Φ_1,Φ_2)

2/ Quelle que soit la substitution σ_1 telle que $\sigma_1\Phi_1 = \sigma_1\Phi_2$ alors il existe une substitution σ_2 telle que $\sigma_1 = \sigma\sigma_2$.

On reconnait là la définition du plus grand unificateur de deux termes dans un langage algébrique. Une version rapide de cet algorithme a été établie dans [HUE76] pour les langages du premier ordre.

III.3.2.5. Caractérisation d'un ensemble de types déterministe par rapport aux appels

Théorème 1:

Un ensemble de définitions de types abstraits génériques T est déterministe par rapport aux appels si pour tout couple de fonctions de même nom et de fonctionnalités respectives Φ_1 et Φ_2 il n'existe pas de plus grande sous-fonctionnalité commune à Φ_1 et Φ_2 .

Démonstration:

On a vu précédemment que si T n'est pas déterministe par rapport aux appels alors il existe un appel explicite à une fonction f qui est ambigu. Donc il existe au moins deux définitions de f de fonctionnalités respectives Φ_1, Φ_2 telle que la fonctionnalité Φ de l'appel explicite de f vérifie:

$$\Phi_1 \geq \Phi \quad \text{et} \quad \Phi_2 \geq \Phi$$

Φ est donc sous-fonctionnalité commune à Φ_1 et Φ_2 ce qui implique l'existence de $PGSFC(\Phi_1, \Phi_2)$ avec $PGSFC(\Phi_1, \Phi_2) \geq \Phi$ cqfd

Ce théorème assure que l'on peut vérifier le caractère déterministe (par rapport aux appels) d'un ensemble de types génériques et surtout que cette vérification peut être faite statiquement à l'introduction de nouveaux types dans l'ensemble T .

Exemple:

Soient les deux fonctionnalités Φ_1 et Φ_2 correspondant à deux définitions d'une fonction f générique au sens incrémental

$$1/ \quad \Phi_1 = [\underline{ent} \times t_1 \times \underline{\tau_1}\ (\underline{\tau_2}(t_2))]$$
$$\Phi_2 = [t_3 \times \underline{\tau_1}(t_3) \times \underline{\tau_1}(t_n)]$$

Alors il existe $\sigma = \{<t_3, \underline{ent}>, <t_1, \underline{\tau_1}(\underline{ent})>, <t_4, \underline{\tau_1}(t_2)>\}$ telle que

$$\sigma\Phi_1 = \sigma\Phi_2 = PGSFC\ (\Phi_1, \Phi_2)$$

2/ $\Phi_1 = [\underline{\tau_1}(t_1) \times t_1]$; $\Phi_2 = [t_2 \times \underline{\tau_2}(t_2)]$

Aucune substitution ne peut unifier ces deux types, donc aucun appel explicite de f ne peut être ambigu.

III.3.2.6. Type du résultat d'un appel explicite

Soit $f(x_1,x_2,\ldots,x_n)$ un appel explicite tel que $\Phi_{appel(f)} = [e_1 \times e_2 \times \ldots \times e_n]$.
Dans un ensemble de types déterministe par rapport aux appels, il existe une seule définition f et une seule substitution σ telles que:

$$\Phi_{appel(f)} = \sigma\Phi_{domaine(f)}$$

Si le type du résultat de f, dans la fonctionnalité de définition, est $\Phi_{res(f)}$ alors le type du résultat de l'appel est $\sigma\Phi_{res(f)}$.

Exemple:

Soient les définitions de fonctions:

chercher : de fonctionnalité: $\underline{table}(\underline{t_1},\underline{t_2}) \times \underline{t_1} \rightarrow \underline{t_2}$
elem : de fonctionnalité: $\underline{matrice}(\underline{t_3}) \times \underline{ent} \times \underline{ent} \rightarrow \underline{t_3}$

et les déclarations:

$\underline{table}(\underline{t_4}, \underline{matrice}(\underline{t_4}))$ *tab* ;
$\underline{t_4}$ *x*;

Alors l'affectation x := elem(chercher(tab, x),1,2) est correcte du point de vue des types car:

$$\Phi_{appel(chercher)} = [\underline{table}(\underline{t_4}, \underline{matrice}(\underline{t_4})) \times \underline{t_4}]$$

donc $\Phi_{appel(chercher)} = \{<\underline{t_1},\underline{t_4}> , <\underline{t_2},\underline{matrice}(\underline{t_4})>\}\ \Phi_{domaine(chercher)}$

Le type du résultat de chercher est donc $\underline{matrice}(\underline{t_4})$

$$\Phi_{appel(elem)} = [\underline{matrice}(\underline{t_4}) \times \underline{ent} \times \underline{ent}]$$

qui est en fait

$$\{<\underline{t_3},\underline{t_4}>\}\Phi_{domaine(elem)}$$

le type du résultat de élément est donc $\underline{t}$.

REGLES

On peut toujours déterminer le type d'une expression à partir du type de ses composants si les deux conditions suivantes sont remplies:

1/ Quelleque soit la fonction générique f:

$$V(\Phi_{domaine(f)}) = V(\Phi_{def(f)})$$

c'est-à-dire que tous les types formels d'une fonctionnalité apparaissent au moins une fois dans $\Phi_{domaine(f)}$.

Exemple:

Les définitions suivantes sont autorisées:

fonction f_1 (τ_1 (t_1,t_2), τ_2(t_3)) *retourne* τ_3(t_1,t_2,t_3)
fonction f_2 (τ_1 (t_1,t_2)) *retourne* t_2

Par contre

fonction f_3 (τ_1 (t_1)) *retourne* τ_2 (t_2)

est interdite.

2/ Les constantes, c'est-à-dire les fonctions sans paramètre, doivent être typées:

- soit par leur notation:

ex.: 5 ou "a" pour les entiers ou les caractères

- soit en les préfixant par leur type:

ex.: *vecteur(ent)* : *(1,9,5,7)*

III.3.3. Infinité dynamique de types

III.3.3.1. Appels formels et implicites

Dans le paragraphe consacré aux propriétés nous avons évoqué le mécanisme de liaison implicite (III.2.3.1.) des fonctions dans le cas d'appels formels. L'exemple, rappelons-le, concernait l'appel de la fonction existe définie sur les listes:

si existe(lli,li) alors ...

ou

$$\Phi_{appel(existe)} = [liste(liste(ent)) \times liste(ent)]$$

On a vu que l'appel formel de = dans existe était lié à la fonction eq-liste parce que la propriété du type liste et celle exigée pour t (type formel de la définition de liste) était la même à savoir l'égalité.

Nous prenons la notation $L_p(h,\sigma)=g$ pour exprimer qu'un appel formel h dans une fonction exigeant la propriété p, avec la substitution σ, est résolu par l'appel implicite de g.

Dans notre exemple:

$$L_{égalité}(=,\{<t,liste(ent)>\}) = \text{eq-liste}$$

Le type formel de l'appel de g dépend lui-même d'une substitution σ_g de la manière suivante:

$\Phi_{appel(g)} = \sigma_g \; \Phi_{def(g)}$ où σ_g est déterminée de la façon suivante:

Soit $\tau(t_1,t_2,\ldots)$ le type abstrait générique dans lequel g est définie, t_h le type formel, dans la fonction appelante, qui est supposé avoir la propriété p et σ la substitution associée à l'appel formel, on a:

$$\{<t_h, \tau(t_1,t_2,\ldots)>\} \; \sigma_g = \sigma\lceil\{t_g\}$$

Dans l'exemple précédent la substitution de l'appel formel est: $\{<\underline{t},\underline{liste}(\underline{ent})>\}$, celle de l'appel implicite est $\{<t',\underline{ent}>\}$ et on a:

$$\{<\underline{t}',\underline{liste}(\underline{t})>\} \quad \{<\underline{t}',\underline{ent}>\} \quad = \quad \{<\underline{t},\underline{liste}(\underline{ent})>\}$$

Si le type τ n'est pas générique, la substitution de l'appel implicite est alors vide. Par exemple pour la liaison:

$$L_{\text{égalité}}(=, <\underline{t},\underline{ent}>) = =_{\underline{ent}} ;$$

où $=_{\underline{ent}}$ est la notation de l'opérateur d'égalité sur les entiers.

On a:

$$\{<\underline{t},\underline{ent}>\} \quad \{\ \} = \{<\underline{t},\underline{ent}>\}$$

ce qui est cohérent avec: $\Phi_{\text{appel}(=_{\underline{ent}})} = \sigma_{id} \ \Phi_{\text{domaine}(=_{\underline{ent}})}$

La résolution des appels formels doit être faite à la compilation. Mais ceci n'est possible que si le nombre de fonctions distinctes appelées dynamiquement est borné ; ce qui équivaut à avoir un nombre fini de types.

III.3.3.2. Le problème:

Une fonction générique, au sens structural, peut être définie de façon récursive, elle peut également être appelée récursivement par une chaîne d'appels explicites ou implicites. Pour un appel explicite, cette récursion générique peut être une récursion "simple" pour chaque membre de la famille générique.

Exemple:

```
fonction f (τ(t)x) ;
   si c alors α
        sinon f(g(x))
   finsi ;
```

ou

$$\Phi_{def(g)} = [\tau(t_1) \rightarrow \tau(t_1)]$$

Pour chaque type effectif substitué à t dans une occurrence de définition, la fonction f est récursive au sens habituel:

f : τ(ent) appelle f : τ(ent) etc...

Mais la récursion générique peut conduire à une séquence d'appels de fonctions distinctes à l'intérieur de la famille générique. Cette famille étant potentiellement infinie, le nombre de fonctions distinctes appelées peut également être infini.

Exemple:

```
fonction f(τ(t)x) ;
   début
     τ(τ(t))Y ;
     si  c  alors α
            sinon y:=β ; f(y)
     finsi ;
   fin ;
```

Pour un appel particulier: f(a) où a est par exemple de type $\tau(\underline{ent})$, on a potentiellement la séquence d'appels:

$$f: \tau(\underline{ent}) \rightarrow f: \tau(\tau(\underline{ent})) \rightarrow f: \tau(\tau(\tau(\underline{ent}))) \rightarrow \ldots$$

GRIES et GEHANI [GRI77] ont posé le problème en termes d'infinité de types, nous parlerons plutôt du nombre de fonctions distinctes appelées.

III.3.3.3. Définition des relations d'appel

Le paragraphe précédent suggère : pour détecter statiquement la possibilité d'avoir à l'exécution une infinité de types, il est nécessaire de connaître, pour chaque fonction, les différents appels possibles qu'elle effectue (explicites ou implicites) et de disposer des substitutions associées.

A cette fin, nous allons nous intéresser à des relations binaires renseignées. Les renseignements appartiennent à un ensemble S d'ensembles de substitutions. L'absence de relation est notée par l'ensemble vide Ø. Nous définissons deux opérations sur S:

1/ $\forall$ S1, S2 $\in$ S S1 $\cup$ S2 $\in$ S c'est l'union au sens ensembliste
2/ $\forall$ S1, S2 $\in$ S S1 $\circ$ S2 $\in$ S ou $S1 \circ S2 = \{\sigma_1\sigma_2 \mid \sigma_1 \in S1 \text{ et } \sigma_2 \in S2\}$

Nous notons $f \rightarrow g$ l'appel explicite de g par f ($f \rightarrow g$ n'existe que s'il y a au moins un appel de g dans f).

Relation E:

$$E(f,g) = \begin{cases} \{ \sigma \mid \Phi_{appel(g)} = \sigma\Phi_{domaine(g)} \} & \underline{si}\ f \rightarrow g \\ \emptyset\ \underline{sinon} \end{cases}$$

Cette relation ne considère que les appels explicites, nous allons l'étendre en une relation R qui prend en compte les appels implicites.

C'est-à-dire que si une substitution σ appartient à R(f,g) alors l'une des deux assertions suivantes est vraie:

1/ $\sigma \in E(f,g)$
2/ Il y a un appel formel dans une fonction appelée par f qui est résolu par l'appel implicite de g.

Et dans les deux cas on a:

$$\Phi_{appel(g)} = \sigma\Phi_{domaine(g)}$$

Relation R

$$R(f,g) = \{\sigma \mid [(f \rightarrow g) \vee ((\exists\ h \text{ formel})\ (f \overset{+}{\rightarrow>} h) \text{ ou } \Phi_{appel(h)} = \sigma' \Phi_{domaine(h)}$$

$$\wedge L_p(h,\sigma') = g)] \wedge (\Phi_{appel(g)} = \sigma \Phi_{domaine(g)})\}$$

Remarques:

- La notation $f \rightarrow> g$ signifie $R(f,g) \neq \emptyset$, la relation R est donc définie de manière récursive. Nous proposons un algorithme itératif (A1) pour calculer R.
- Les types formels de la définition de g sont les variables des substitutions de R(f,g) dont les termes dépendent des types formels de la définition de f.

Dans toutes les substitutions σ', de la définition de R(f,h), il existe une seule composante dont la variable est le type formel de h et dont le terme, dépendant des formels de f va permettre de résoudre l'appel formel de h ($L_p(h,\sigma') = g$).

Il est donc nécessaire de faire le lien entre le formel de f et le formel de l'appel de h, c'est-à-dire propager le formel de f tout au long des appels qui aboutissent à h sans ajouter de nouveaux termes dans la substitution. Pour cela nous définissons la relation suivante, notée RF.

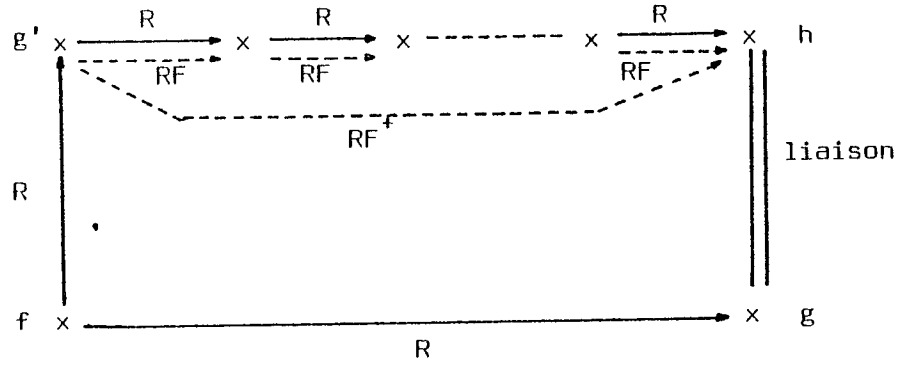
$RF(f,g) = \{<\underline{t_2},\underline{t_1}>\}$ si et seulement si: $(\exists\sigma)\ \sigma \in R(f,g)$ dans laquelle le type formel $\underline{t_1}$ de la définition de f remplace le type formel $\underline{t_2}$ de la définition de g. ($<\underline{t_2},\underline{t_1}> \in \sigma$).

La composition:

$$RF(f,g) \circ RF(g,h) = RF^2(f,h)$$

possède le renseignement $\{<\underline{t_3},\underline{t_1}>\}$.

Avant de décrire complètement l'algorithme qui permet de calculer R, nous allons donner par un exemple, le principe de son fonctionnement:



Ce diagramme résume le calcul de R(f,g). La relation R existe entre f et g', c'est-à-dire que f appelle g' avec une substitution σ dont une des composantes (non formelle) est déterminante pour la liaison de g à h.

Une cascade d'appels existe éventuellement entre g' et l'appel formel h. Toutes ces fonctions sont reliées entre elles par la relation R. Dans toutes les substitutions associées on ne s'intéresse en fait qu'aux composantes formelles (relation RF), n'introduisant pas de nouveaux types, elles ne modifient pas la liaison. On peut alors calculer la fermeture transitive de RF ce qui va permettre de propager le formel de g', et d'effectuer la liaison en définissant la substitution associée à R(f,g).

L'algorithme suivant résume ces opérations.

Algorithme A1 pour le calcul de la relation R

(1) $(\forall f,g)\ R_0(f,g) \leftarrow E(f,g)$

$i \leftarrow 0$;

allera (2) ;

(2) $\forall(f,g)$

$R_{i+1}(f,g) \leftarrow R_i(f,g) \cup$

$\{\sigma \mid (\exists$ h formel$)\ (\exists$ g'$)\ (\exists \sigma' \in R_i(f,g'))$ tel que

$RF_i(g',h) = \{<\underline{t_h}, \underline{t_g},>\}$

$\wedge \{<\underline{t_g},, \underline{\tau}(\underline{t_{g1}},\ldots)>\}\sigma = \sigma' \upharpoonright \{\underline{t_g},\}$

$\wedge\ L_p(h, \{<\underline{t_h}, \underline{\tau}(\underline{t_{g1}}, \underline{t_{g2}},\ldots)>\}\sigma) = g\}$;

allera (3) ;

(3) si $\forall(f,g)\ R_{i+1}(f,g) = R_i(f,g)$

alors stop

sinon $i \leftarrow i+1$; allera (2)

finsi

$\underline{t_{g1}}, \underline{t_{g2}},\ldots$ sont les types formels de la définition de $\underline{\tau}$ dans lequel la fonction g est déclarée, $\underline{t_h}$ est le type formel de l'appel de h, il possède la propriété p. $\underline{t_g}$, désigne le formel de g' à propager jusqu'à l'appel de h $(RF_i^+(g',h) = \{<t_h, t_g,>\})$.

Il est évidemment essentiel que cet algorithme itératif s'arrête ; nous en donnons une preuve dans [BER78b]. L'idée de la démonstration repose sur le fait que au pas (2) la "taille" de la substitution σ (c'est-à-dire le degré d'imbrication de ses termes) est inférieure à celle de σ', et que dans un ensemble de types abstraits génériques un type formel d'une définition de fonction ne peut se substituer qu'à un nombre borné de formels différents.

Exemple:

Nous considérons un ensemble: $\{f, g_1, g_2, g_3\}$ de noms de fonctions dont nous donnons ci-dessous la fonctionnalité de définition ainsi que les différents appels avec les fonctionnalités d'appel associées.

$$f: \tau(t) \rightarrow h:t$$

Ceci signifie que la fonction f, de fonctionnalité de définition $\tau(t)$, appelle la fonction h avec la fonctionnalité t (il s'agit donc là d'un appel formel).

$$\begin{aligned}
g_1 &: \tau_1(t_1) \rightarrow f: \tau(\tau_2(t_1)) \\
&\rightarrow f: \tau(t_1) \\
g_2 &: \tau_2(t_2) \rightarrow f: \tau(\tau_1(\tau_2(t_2))) \\
g_3 &: \tau_3(t_3) \rightarrow g_1 : \tau_1(\tau_3(t_3))
\end{aligned}$$

On suppose que h peut être lié à g_1, g_2 ou g_3. La relation R est alors:

$$\begin{aligned}
R(g_1,f) &= \{\{<t,\tau_2(t_1)>\},\{<t,t_1>\}\} \\
R(g_2,f) &= \{\{<t,\tau_1(\tau_2(t_2))>\}\} \\
R(g_3,g_1) &= \{\{<t_1,\tau_3(t_3)>\}\} \\
R(g_1,g_2) &= \{\{<t_2,t_1>\}\} \\
R(g_2,g_1) &= \{\{<t_1,\tau_2(t_2)>\}\} \\
R(g_3,g_3) &= \{\sigma_{id}\} \\
R(g_2,g_2) &= \{\sigma_{id}\}
\end{aligned}$$

III.3.3.4. Récursivité propre, récursivité infinie

Définition:

Une fonction f est récursive (on note Rec(f)) si et seulement si $f \overset{+}{\rightarrow\!\!>} f$.
A chaque fonction récursive f, on associe une information $\Sigma(f)$ de la façon suivante.

Soit s une séquence d'appels:

$s = (f_1, f_2, \ldots, f_n)$ telle que $(f_1 = f) \wedge (f_n = f) \wedge (\forall_i \in [2, n-1] f_i \neq f \wedge \forall i \neq j,\ f_i \neq f_j)$

avec $\forall i \in [1, n-1]\ \ R(f_i, f_{i+1}) \neq \emptyset$;

$$\sigma_s = R(f_{n-1}, f_n) \circ R(f_{n-2}, f_{n-1}) \circ \ldots \circ R(f_1, f_2)$$

alors $\Sigma(f) = \bigcup_s \sigma_s$ pour toutes les séquences s de f à f.

Dans l'exemple précédent, on a:

$Rec(g_1)$ et $\Sigma(g_1) = \{\{<\underline{t_1}, \underline{\tau_2}(\underline{t_1})>\}\}$

$Rec(g_2)$ et $\Sigma(g_2) = \{\{<\underline{t_2}, \underline{\tau_2}(\underline{t_2})>\}\ ,\ \{\ \}\}$

$Rec(g_3)$ et $\Sigma(g_3) = \{\{\ \}\}$

Définition:

Une substitution σ est stable si et seulement si:

$(\exists n,\ card(\sigma) \geq n > 0)\ (\exists p,\ card(\sigma) \geq p > 0)$ tels que

$$\sigma^n = \sigma^{n+p}$$

(card(σ) est la notation pour le cardinal de σ).

Nous donnons ci-dessous un algorithme qui calcule si une substitution donnée est stable ou non.

Algorithme A2 pour déterminer si un substitution est stable

Soit une substitution σ.

1/ On considère la relation d'équivalence (notée ≡) sur les variables de σ

$\underline{t_i} \equiv \underline{t_j}$ si et seulement si $<\underline{t_i}, \underline{t_j}>$ ou $<\underline{t_j}, \underline{t_i}>$ appartiennent à σ et ce pour $i \neq j$

et $\underline{t_i} \equiv \underline{t_j}$ si $i = j$.

La première étape de l'algorithme consiste à calculer les classes d'équivalences de cette relation (on note $[t_i]$ la classe de t_i).

2/ On calcule ensuite la relation $\overset{S}{\rightarrow}$ entre classes.

$$[t_i] \overset{S}{\rightarrow} [t_j] \quad \text{si et seulement si} \quad \langle t_i, \tau(\ldots,t_j,\ldots)\rangle \in \sigma$$

Si le graphe obtenu est acyclique, alors la substitution initiale est stable.

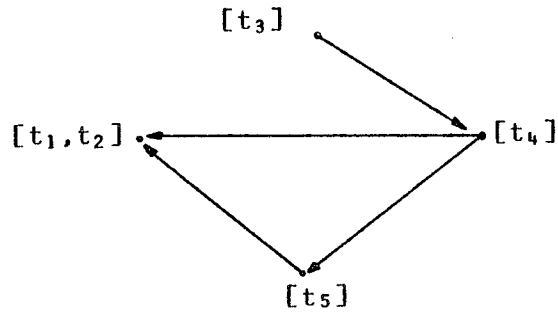
On trouvera dans [BER78b] une preuve complète de cet algorithme. On peut toutefois faire l'analogie entre ce problème et celui du langage infini pour les langages hors-contexte [VAU70]. En considérant les variables de σ comme les noms de classe d'une grammaire dont l'ensemble des terminaux est constitué par: les symboles (",")", ", ", auxquels s'ajoutent les noms de types. Il est alors facile de voir que l'existence d'un cycle dans le graphe est équivalent à l'existence d'un symbole auto-imbriqué dans la grammaire (condition nécessaire et suffisante pour que le langage soit infini).

Exemple:

Considérons la substitution:

$$\sigma_1 = \{\langle t_1,t_2\rangle, \langle t_2,t_1\rangle, \langle t_3,\tau_1(t_4)\rangle, \langle t_4,\tau_2(t_1,t_5)\rangle, \langle t_5,\tau_3(t_2)\rangle\}$$

le graphe associé est:



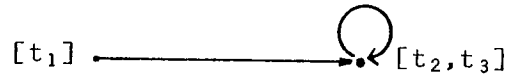
Ce graphe étant sans cycle, la substitution σ_1 est stable, en effet:

$$\sigma_1^3 = \sigma_1^{3+2}$$

Par contre:

$$\sigma_2 = \{<t_1,\tau(t_3,t_2)>, <t_2,t_3>, <t_3,\tau(t_2)>\}$$

a le graphe associé suivant:



qui possède un cycle, σ_2 n'est donc pas stable.

Définition:

On dit qu'une fonction est récursive propre si et seulement si Rec(f) et $\forall \sigma \in \Sigma(f)$, σ est stable. Une fonction f est récursive infinie (on note $Rec^{\infty}(f)$) si et seulement si Rec(f) et $(\exists \sigma \in \Sigma(f))$ telle que σ n'est pas stable.

Dans l'exemple précédent, on a $Rec^{\infty}(g_1)$ et $Rec^{\infty}(g_2)$.

Théorème 2:

Etant donné un ensemble T de types abstraits génériques, une condition nécessaire et suffisante pour qu'aucun programme ne produise un nombre non borné d'appels à des fonctions distinctes (n'engendre un nombre non borné de types) est que:

$$\forall f \text{ définie dans } T \quad \neg\, Rec^{\infty}(f).$$

Preuve:

1/ La condition est nécessaire. Soit f une fonction telle que $Rec^{\infty}(f)$, alors $\exists \sigma \in \Sigma(f)$ telle que σ n'est pas stable ; il existe donc un appel à f de fonctionnalité $\sigma_1\ \Phi_{domaine(f)}$ qui conduira dynamiquement, à une suite d'appels d'un nombre infini de fonctions distinctes:

$f : \sigma_1\ \Phi_{domaine(f)}$; $f: \sigma\sigma_1\ \Phi_{domaine(f)}$; ... ; $\sigma^n\sigma_1\ \Phi_{domaine(f)}$; .....

2/ La condition est suffisante. Supposons qu'il existe un appel à une fonction f, qui dynamiquement produise un nombre non borné d'appels à des fonctions distinctes avec la substitution σ_1 sur les formels de la définition de f. Nous divisons la séquence de fonctions appelées explicitement ou implicitement en sous-séquences à l'intérieur desquelles la dernière fonction f_i appelle une fonction formelle h qui est liée à la première fonction f_{i+1} de la séquence suivante à l'aide d'un type de la substitution initiale σ_1. Les termes de σ_1 étant bornés, le nombre de sous-séquences est borné, il existe donc au moins une sous-séquence s, de longueur infinie.

Pour chaque liaison d'une fonction g de s à un appel formel, il existe une fonction k dans s telle que R (k,g). En effet, les liaisons d'appels formels avec la substitution initiale ont tous servi à diviser la séquence d'appels en sous-séquences. Un des deux cas suivants est alors possible dans s.

Soit le nombre d'appels explicites est infini, soit le nombre d'appels implicites est infini.

Mais dans les deux cas on peut extraire de s une séquence s' telle que $\forall i\ R(f_i, f_{i+1})$ et s' est infinie. Comme le nombre de définitions de fonctions est borné dans l'ensemble $\mathcal{T}$, il existe une fonction f' dans s' telle que Rec(f'). De plus, si $F' = \{f' \in s' \mid Rec(f')\}$, il existe au moins une fonction $f'' \in F'$ telle que $Rec^{\infty}(f'')$, sinon le nombre de fonctions distinctes appelées serait borné, ce qui contredit l'hypothèse.

Remarques:

1/ Lorsque toute fonction récursive est récursive propre, le nombre de fonctions distinctes appelées est toujours borné même s'il y a séquence infinie d'appels, il en est de même pour le nombre de types engendrés.

2/ Si pour toute fonction f définie dans $\tau \in \mathcal{T}$ on a $\neg$ Rec(f), il est quand même possible d'avoir des fonctions appelées récursivement par le jeu des liaisons d'appels formels ; ces fonctions sont appelées avec des fonctionnalités distinctes, mais en nombre fini.

III.3.3.5. Compilation d'un appel de fonction

Soit T un ensemble de types génériques satisfaisant à la propriété: ($\forall$f définie dans T)$\neg$ Rec$^{\infty}$(f). Pour toute fonction f, on peut déterminer d'après le graphe de la relation R, l'arbre des fonctions appelées avec leur fonctionnalité.

Les feuilles de cet arbre sont de trois types:

- Ce sont des fonctions terminales (notées "T") qui n'appellent aucune autre fonction.
- Ce sont des appels formels (notés "F") dépendant , après substitution, d'un type formel de la fonction f.
- Ce sont des appels récursifs propres (notés "R") à une fonction déjà rencontrée dans l'arbre.

Pour un appel particulier à f avec une substitution σ_1, il suffit de considérer l'arbre d'appels de la fonction et de remplacer chaque feuille "f" par l'arbre d'appels de la fonction qui est liée à cet appel formel à l'aide de la substitution σ_1, et ainsi de suite jusqu'à épuisement des feuilles "f". Le processus est évidemment fini puisque les termes des composantes de σ_1 sont tous de longueur finie. Dans le cas où il y a récursivité, cette dernière peut apparaître d'une manière anticipée dans l'arbre d'un appel particulier.

Exemple:

Soient les définitions de fonctions et les appels suivants:

f : $\underline{\tau(t\_)}$ → h : $\underline{t}$ (formel)

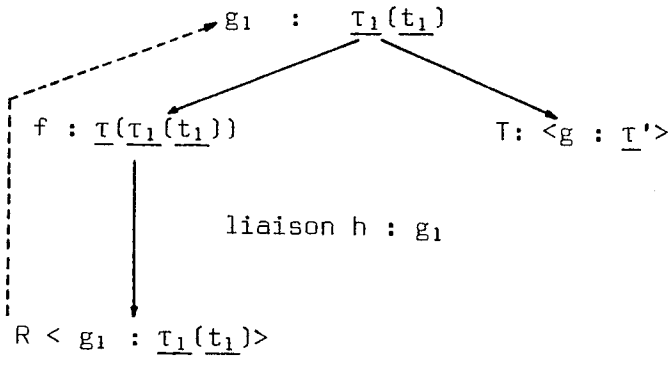
g_1 : $\underline{\tau_1(t_1)}$ → f : $\underline{\tau(\tau_1(t_1)}$

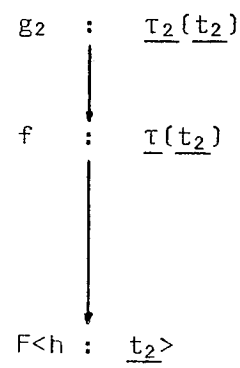
g : $\underline{\tau'}$

g_2 : $\underline{\tau_2(t_2)}$ → f : $\underline{\tau(t_2)}$

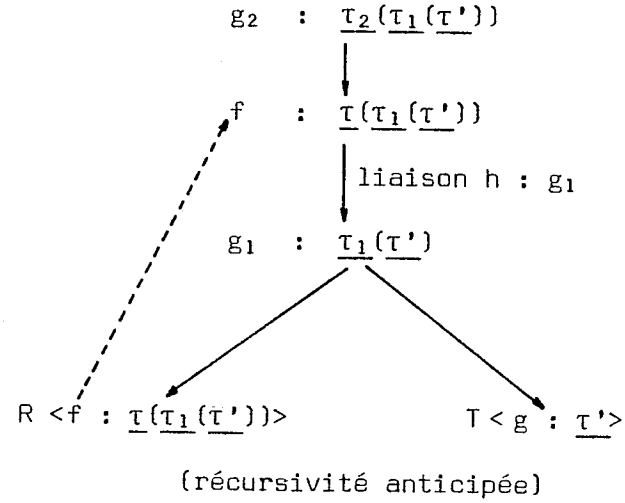
h peut être liée à g, g_1 ou g_2 ; d'après R, on a Rec(g_1).

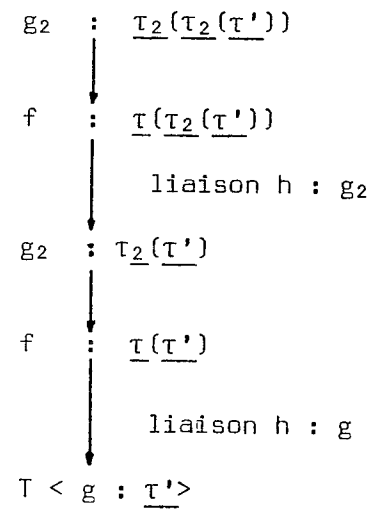
Les arbres d'appels de g_1 et g_2 sont:





Les arbres des appels $g_2 : \tau_2(\tau_1(\tau'))$ et $g_2 : \tau_2(\tau_2(\tau'))$ sont:





D'un arbre d'appel particulier il est possible d'extraire les adresses des fonctions qui devront être liées aux appels formels (c'est un arbre d'adresses). A l'exécution un pointeur détermine à tout moment quelle est la prochaine adresse de fonction appelée par appel formel ; un appel formel est donc traduit par un branchement indirect à ce pointeur, suivi de l'avancement de ce pointeur dans l'arbre. La suite des appels explicites détermine éventuellement le choix des branches à suivre et une récursivité conduit à repositionner le pointeur à une adresse antérieure.

Remarque:

Une étude détaillée ainsi qu'une mise en oeuvre expérimentale des algorithmes présentés dans cette partie se trouvent dans [SOL78].

III.4. MULTIREPRESENTATION

Dans les exemples présentés jusqu'à maintenant, nous avons toujours considéré que les objets d'un type abstrait disposaient d'une seule représentation. Notons cependant que, pour les types abstraits génériques, la représentation étant elle-même générique, chaque occurrence de définition d'un type de la famille engendrait une occurrence de définition de la représentation. La multireprésentation à laquelle nous nous intéressons dans cette partie est de nature différente.

Il arrive souvent que certaines caractéristiques propres aux objets d'un type abstrait (éventuellement générique) font qu'une représentation particulière offre un gain appréciable, soit du point de vue de l'encombrement mémoire, soit, du fait que certaines fonctions peuvent être implantées de manière plus efficace. Dans ce cas,il est souhaitable que ces différentes représentations puissent coexister dans un programme particulier sans que l'écriture de ce dernier dépende d'un choix particulier. Il n'en reste pas moins que le choix initial doit être fait par le programmeur sur la base de la nature de l'objet et du sous-ensemble de fonctions qui lui sont appliquées. Ce n'est donc pas la donnée explicite d'une représentation particulière (qui reste donc cachée) mais plutôt celle d'un ensemble de propriétés qui guideront le compilateur dans son choix.

La définition modulaire des types abstraits fournit une première approche de la solution. En effet, la représentation étant cachée à l'utilisateur, seul l'ensemble des fonctions définissant le type doit être concerné par d'éventuelles représentations multiples, ceci afin que toute modification dans la, ou les représentations, ainsi que dans les fonctions qui y accèdent, n'entraîne pas de modification dans les programmes utilisant le type (et réciproquement d'ailleurs). Il faut cependant noter: cette remarque ne fait que repousser le problème au niveau de la définition du type et en particulier des fonctions qui le caractérisent puisque ce sont

elles qui doivent alors prendre en charge les diverses représentations.

Nous allons essayer de préciser le problème sur un exemple et examiner deux solutions possibles avant de proposer un mécanisme compatible avec la généricité.

III.4.1. Exemple

Nous avons choisi comme exemple la spécification paramétrée du type abstrait matrice.

```
type matrice (type t, ent n) : dioïde (add, mmul, mult, munit)
    rep [1:n, 1:n] t finrep ;
 fonction ranger (matrice (t,n)m, ent i,j,t val) ;
    si (i ≥ 1) ∧ (i ≤n) ∧ (j ≥ 1) ∧ (j ≤ n) alors rep:m (i,j) := val sinon erreur finsi
 fonction select (matrice (t,n)m, ent i,j) retourne t ;
    si (i ≥ 1) ∧ (i ≤m) ∧ (j ≥ 1) ∧ (j≤n) alors rep:m(i,j) sinon erreur finsi ;
 constante m nul retourne matrice (t,m) ;
    exige t : monoïde (+,0) ;
  début
    matrice (t,n)m ;
    pour i de 1 a n faire pour j de 1 a n faire rep:m(i,j):=0 fait fait ;
    m
  fin ;
constante munit  retourne matrice (t,n)
    exige t : dioïde (+,0,*,1) ;
  début
    matrice (t,n) m;
    pour i de 1 a n faire pour j de 1 a n faire rep:m(i,j):=0 fait
        rep:m(i,i) := 1 fait ;
  . m
  fin ;
fonction add (matrice(t,n) m1,m2) retourne matrice (t,m) ;
    exige t : monoïde (+,0) ;
  début
    matrice (t,n) m ;
    pour i de 1 a n faire pour j de 1 a n faire
        rep:m(i,j) := rep:m1 (i,j) + rep:m2 (i,j) fait fait ;
    m
  fin ;
```

```
fonction mult (matrice (t,n) m1,m2) retourne matrice (t,n) ;
        exige t : dioïde (+,0,*,1) ;
    début
        matrice (t,n)m ; t s ;
        pour i de 1 a m faire pour j de 1 a n faire s:=0 ;
            pour k de 1 a n faire s:=s+rep:m1(i,k) * rep:m2(k,j) fait ;
            rep : m(i,j) := s fait fait ;
        m
    fin ;
fintype
```

La représentation par un tableau bidimensionnel peut ne pas convenir dans le cas de matrices creuses. On souhaite alors disposer d'une représentation en liste chaînée qui permet d'optimiser l'occupation mémoire.

Il semble à peu près évident que le choix de la représentation doit se faire au moment de la déclaration de l'objet. Ceci va à l'encontre de la philosophie des types abstraits (transparence complète de la représentation), mais comme on ne dispose pas de moyens automatiques pour choisir, parmi les représentations possibles, la meilleure,c'est au programmeur d'en décider, étant entendu que dans la suite du programme il n'aura plus à en tenir compte.

On peut alors faire de la représentation un paramètre du type matrice

```
        type repmat = (pleine, creuse) ;
type matrice (type t, ent n, repmat rm) ;
        rep cas rm dans
            (pleine) : [1:n , 1:n] t |
            (creuse) : type elem = struct (ent l,c,t val) ;
                       liste (elem) ;
            fincas
        finrep ;
```

Le paramètre de représentation est d'un type scalaire, si sa valeur est connue à la compilation (dans une occurrence de définition) le choix de la représentation est fait statiquement sinon sa valeur est connue au moment de l'élaboration de la déclaration à l'exécution.

Notons que dans le cas de matrices creuses, on a choisi pour représentation un type abstrait déjà défini (liste).

III.4.1.1. Approche sans conversion

La multireprésentation qui vient d'être introduite va évidemment avoir un certain nombre de répercussions sur les fonctions du type matrice. Nous allons nous intéresser dans un premier temps aux fonctions d'accès en réécrivant la fonction ranger par exemple:

```
fonction ranger (matrice(t,n,m) mat, ent i,j,t v) ;
   exige t : monoïde (+,0) ;
   si (i ≥ 1) ∧ (i ≤ n) ∧ (j ≥ 1) ∧ (j ≤ n) alors
      cas m dans
         (pleine): rep : mat(i,j) := v |
         (creuse): elem x := (i,j,v) ;
            si trouve (rep:mat,i,j) alors
              si  v  =0
                  alors  val de element (rep:mat,i,j):=v
                  sinon  oter(rep:mat,i,j)
              finsi
            sinon
              si  v  =0
                  alors ajouter (rep:mat,x)
              finsi
            finsi
      fincas
   sinon erreur fsi
```

On retrouve, bien entendu, la discrimination sur le paramètre de représentation pour spécifier les deux accès. Notons que ranger est dans ce cas une fonction d'accès qui exige, elle aussi, pour t une propriété de monoïde. Il est en effet nécessaire dans le cas de matrices creuses de disposer d'une notation formelle pour l'élément neutre. Les fonctions d'accès aux listes ont été réécrites pour les éléments particuliers utilisés ici, seule ajouter reste identique.

> Dernière remarque enfin, le forceur rep n'est défini qu'à l'intérieur d'un bloc où le paramètre de représentation est connu.

Le cas de la fonction select se traite de manière analogue et ne pose pas de problème particulier. Voyons maintenant ce qu'il en est des deux fonctions add et mult.

Ces deux fonctions ont ceci de particulier, qu'elles admettent deux opérandes de type matrice et délivrent un résultat du même type. Si on adopte pour add et mult la même démarche que pour ranger et select on sera conduit à écrire, pour add par exemple, autant de fonctions qu'il y a de combinaisons possibles entre les représentations des opérandes et du résultat.

add : pleine, pleine → pleine
add : pleine, pleine → creuse
add : creuse, creuse → pleine
add : creuse, creuse → creuse
add : pleine, creuse → pleine
add : pleine, creuse → creuse
add : creuse, pleine → pleine
add : creuse, pleine → creuse

Ce qui fait au total 8 membres pour la fonction add, autant pour la fonction mult auxquels s'ajoutent les deux membres de ranger et select, ce qui en tout fait 20 fonctions pour implanter les 4 fonctions initiales, ceci n'est évidemment pas très réaliste surtout si on ajoute une troisième représentation ... ou si on écrit des fonctions avec 3, 4 ou 5 matrices en paramètres ...

III.4.1.2. Approche avec conversion

Nous avons vu que l'approche précédente conduisait à une croissance exponentielle du nombre de fonctions à écrire dans le cas de représentation multiple. Une solution à ce problème est de disposer de fonctions de conversion de représentation (dans le cas des matrices, deux fonctions, l'une transformant une matrice pleine en matrice creuse et l'autre inversement).

Un compromis doit être alors réalisé entre le nombre de fonctions à écrire, dans une définition de type, et le nombre de conversions à effectuer dans un programme particulier.

On pourrait par exemple, n'écrire les fonctions ranger, select, add et mult que sur la représentation en tableau. Le programme utilisant ces fonctions sur des matrices creuses passerait alors la majeure partie de son temps d'exécution à effectuer des conversions et en plus consommerait une place mémoire importante.

Une solution qui semble raisonnable, consiste à écrire les fonctions d'accès (ranger et select) sur les diverses représentations (pleine et creuse). Il est en effet absurde de devoir convertir une matrice creuse en matrice pleine simplement pour accéder à un élément.

Quant aux autres fonctions, on peut ne les écrire que pour des combinaisons particulières de représentation des opérandes et du résultat.

Par exemple, pour la fonction add on peut envisager deux algorithmes:

1/ $add : pleine \times pleine \rightarrow pleine$
2/ $add : creuse \times creuse \rightarrow creuse$

qui tiennent compte des représentations particulières. Ainsi (1) s'écrit:

```
fonction add (matrice(t,m,pleine)m1,m2)retourne matrice(t,m,pleine) ;
    exige t : monoïde (+,0) ;
  debut
      matrice (t,n,pleine)m2 ;
    pour i:=1 jusqua n faire
        pour j:=1 jusqua n faire
            rep:m2(i,j) := rep:m1(i,j) + rep:m2(i,j) ;
        fait fait ;
        m2
  fin ;
```

Dans le cas où les représentations des opérandes et du résultat sont différentes, le compilateur va déterminer, dans l'arbre abstrait d'une expression, l'emplacement des conversions.

Cette approche n'est toutefois pas entièrement satisfaisante. En effet l'algorithme qui localise les conversions [BER77b] a comme seul critère d'en engendrer un nombre minimum. Ce critère n'est pas toujours compatible avec les contraintes d'occupation mémoire. Ceci nous conduit à proposer une troisième approche.

III.4.2. Approche générique

Le paragraphe précédent suggère une assez forte analogie entre la généricité des types dont nous avons déjà parlé et la généricité de la représentation. En effet, il est clair que les fonctions d'accès, telles que nous les avons définies, sont génériques au sens incrémental sur la représentation. Dès lors on peut se demander s'il n'est pas possible de définir les autres fonctions (add et mult) de façon générique, mais au sens structural cette fois, en laissant aux fonctions d'accès utilisées le soin de jouer le rôle des conversions.

La seule contrainte imposant les conversions dans l'approche précédente est que la représentation doit être la même pour les opérandes et le résultat des fonctions add et mult. Si on décide que cette représentation ne fait plus partie du type mais devient simplement un attribut qui doit être fixé une fois pour toutes à la déclaration, il est alors possible d'envisager des fonctions dont les opérandes sont de représentations différentes, les fonctions d'accès étant elles définies sur chaque représentation.

Add s'écrit alors:

```
fonction add (matrice(t,n)m1,m2) retourne matrice (t,n)
    exige t : monoïde (+,0) ;
        début
          matrice (t,n)mr ;
          pour i de 1 à n faire
                pour j de 1 a n faire
                  ranger(mr,i,j,select(m1,i,j)+select(m2,i,j))
                fait fait ;
          mr
        fin ;
```

La seule différence avec l'écriture précédente est que la spécification du type des opérandes et du résultat ne contient plus l'attribut de

représentation, si bien que les deux appels de select qui opéraient avant sur la même représentation peuvent maintenant travailler chacun sur une représentation différente. Pour un appel particulier de add les opérandes disposeront d'une représentation connue et le compilateur pourra choisir les corps de fonctions d'accès associés.

Un problème subsiste toutefois en ce qui concerne la détermination de la représentation du résultat. Dans les fonctions génériques évoquées précédemment le type du résultat se déduisait par substitution uniforme (cf. III.3.2.6.) du ou des formels de la définition, cette règle ne peut pas s'appliquer ici. Deux solutions sont cependant envisageables:

1/ Il est possible de donner, a priori, des règles permettant de déterminer la représentation du résultat à partir de celles des opérandes.

$add : pleine \times pleine \rightarrow pleine$

$add : pleine \times creuse \rightarrow pleine$

Cette spécification supplémentaire ne semble pas très intéressante, d'autant que dans certain cas elle conduit à des conversions superflues que permet d'éviter la deuxième solution.

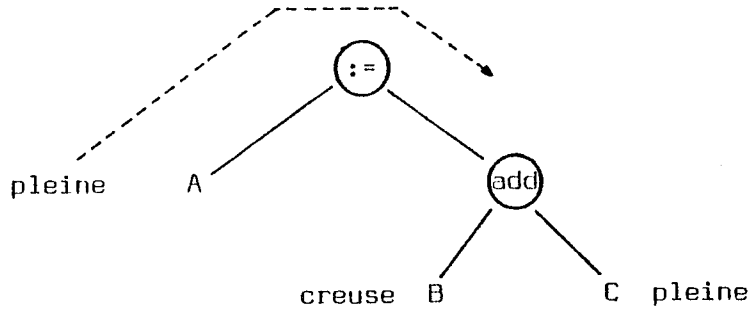
2/ On peut en effet faire que la représentation du résultat soit imposée par le contexte. Par exemple considérons l'instruction suivante:

$A := add(B,C)$

où A, B et C sont des matrices.

Si ces objets ne sont pas des paramètres de procédure, ils ont, lors de leur déclaration, reçu une représentation fixe, par exemple, A et C sont pleines, B est creuse.

Dans l'arbre abstrait de cette instruction, il est alors possible, statiquement, de propager la représentation de A, qui doit être celle du résultat de add, ceci afin d'éviter l'éventuelle conversion lors de l'affectation à A.



mr lors de sa déclaration dans add aura la représentation "pleine" et le compilateur choisira le corps de ranger correspondant.

Notons que cette possibilité, si elle permet d'éviter bon nombre de conversions, ne les supprime cependant pas toutes.

exemple: A := B ;

Si maintenant les opérandes sont les paramètres d'une procédure f il est évidemment impossible de choisir, lors de la compilation de f, la représentation des résultats comme il est d'ailleurs impossible de choisir les corps des fonctions d'accès opérant sur les paramètres de f.
Il faudra attendre les appels particuliers de la procédure pour déterminer ces inconnues. Cette compilation partielle rappelle évidemment celle des fonctions paramétrées par des types, à la différence toutefois que les paramètres, en ce qui concerne la multireprésentation, ne sont soumis à aucune contrainte, si ce n'est l'appartenance à un type, éventuellement générique, et que l'éventail de choix reste relativement restreint.

III.5. CONCLUSION

Arrivés au terme de ce chapitre, nous allons essayer de faire le point sur les diverses propositions évoquées en indiquant quelles sont les directions de nos futurs travaux.

Il est maintenant clair, du moins je l'espère, qu'un type doit être considéré comme un ensemble de fonctions. La très forte analogie entre la généricité, telle que nous l'avons vue (cf.III.1, III.2, III.3) et la multireprésentation (cf.III.4) nous a conduits à proposer trois niveaux de spécification de fonctions.

1/ Les propriétés

On y trouve des fonctions génériques au sens incrémental, elles permettent de caractériser toute une classe de types. La spécification, purement syntaxique exigée à ce niveau permet d'effectuer un minimum de vérifications statiques et de lier les appels formels (cf.III.3). Les développements récents des spécifications algébriques (cf.I) [KRI78], permettent de penser qu'on pourra les utiliser pour enrichir et compléter les spécifications de ce niveau, c'est une des voies de nos recherches actuelles.

2/ Les types abstraits génériques

La séparation déjà évoquée (cf.III.1) en deux classes des fonctions de ce niveau nous conduit à spécifier:

- de manière purement syntaxique les fonctions d'accès associées au type, elles sont génériques au sens incrémental sur les différentes représentations ;
- complètement les autres fonctions du type qui sont, elles, génériques au sens structural. Elles utilisent des fonctions définies sur les types passés en paramètre, mais aussi les fonctions d'accès définies sur la ou les représentations.

3/ Les modules de représentation

De la même façon qu'un type abstrait générique implante les fonctions d'une propriété, un module de représentation, nou seulement fournit une représentation aux objets d'un type, mais encore implante les fonctions d'accès spécifiées dans ce type. Cette séparation, qui n'avait pas été faite jusqu'à maintenant, nous semble justifiée au point de vue méthodologique. En effet, dans une approche descendante, il semble important de pouvoir spécifier un type abstrait générique, et éventuellement d'écrire des algorithmes qui l'utilisent, sans avoir fourni de représentation. Enfin, dernier point, ces modules de représentation sont le cadre naturel de définition des fonctions de conversion.

Cette séparation en trois niveaux est illustrée sur l'exemple des matrices.

```
prop monoïde (type t) ;
     operateur + : t × t → t
     constante 0 : → t
finprop

prop dioïde (type t) ;
     operateur + : t × t → t ;
     constante 0 : → t ;
     operateur * : t × t → t ;
     constante 1 : → t
finprop
```

```
type matrice(type t,ent n,repmat m) : dioîde(add, mnul,mult,munit) ;

    acces:
        fonction ranger : matrice(t,n) × ent × ent × t → neutre ;
        fonction select : matrice(t,n) × ent × ent → t ;
    finacces ;


    constante mnul retourne matrice(t,n) ;
        exige t : monoîde (+,0)
            debut
                matrice(t,n)m ;
                pour i de 1 a n faire pour j de 1 a n faire ranger(m,i,j,0) fait fait ;
                m
            fin ;

    constante munit retourne matrice (t,n) ;
        exige t : dioîde (+,0,*,1) ;
            debut
                matrice(t,n)m ;
                pour i de 1 a n faire pour j de 1 a n faire ranger(m,i,j,0) fait
                                    ranger(m ,i,i,1) fait
                m
            fin ;


    fonction add (matrice(t,n)m1,m2) retourne matrice(t,n) ;
        exige t : monoîde (+,0) ;
            debut
                matrice(t,n)mr ;
                pour i de 1 a n faire pour j de 1 a n faire
                            ranger(mr,i,j,select(m1,i,j) + select(m2,i,j)) fait fait
                mr
            fin ;


    fonction mult(matrice(t,n)m1,m2) retourne matrice(t,n)
        exige t : dioîde (+,0,*,1) ;
            debut
                matrice(t,n)mr ;
                t s ;
                pour i de 1 a n faire pour j de 1 a n faire s:=0 ;
                    pour k de 1 a n faire s:= s + select(m1,i,k)*select(m2,k,j) fait
                            ranger(mr,i,j,s) fait fait
                m2
            fin ;

fintype ;
```

```
repmodule matpl implante matrice (t,n,pleine) ;
   rep [1:n, 1:n] t finrep ;
   fonction ranger(matrice(t,n,pleine)m, ent i, j, t val) ;
      si (i≥1) ∧ (i≤n) ∧ (j≥1) ∧ (j≤n) alors rep:m(i,j) := val sinon erreur finsi ;
   fonction select(matrice(t,n,pleine)m, ent i,j) retourne t ;
      si (i≥1) ∧ (i≤n) ∧ (j≥1) ∧ (j≤n) alors rep:m(i,j) sinon erreur finsi ;
finrepmodule ;


repmodule matcr implante matrice(t,n,creuse) ;
   rep: type elem = struct (ent l,c,t val) ;
        liste (elem) finrep ;
   fonction ranger (matrice(t,n,creuse)m, ent i,j,t val) ;
      exige t : monoïde(+,0) ;
    si (i≥1) ∧ (i≤n) ∧ (j≥1)∧(j≤n) alors elem x := (i,j,val) ;
        si trouve(rep:m,i,j) alors
           si val   =0
               alors val de element (rep:m,i,j) := v
               sinon oter (rep:m,i,j)
           finsi
             sinon
               si val   =0
                 alors ajouter(rep:mat,x)
               finsi
             finsi
        sinon erreur finsi ;

   fonction select(matrice(t,n,creuse)m,ent i,j) retourne t ;
      exige t : monoïde (+,0) ;
    si (i≥1)∧(i≤n) ∧ (j≥1) ∧ (j≤n) alors
        si trouve (rep:m,i,j) alors val de element(rep:m,i,j)
                              sinon  0
        fin
    finsi
   sinon erreur finsi ;
finrepmodule ;
```

CONCLUSION

CONCLUSION

Il n'est pas sûr au terme de cette étude, que l'on puisse répondre à la question: "Qu'est-ce qu'un type?". Cependant les trois niveaux de spécifications que nous avons proposés (cf.III.5.) permettent de dégager ce qui à notre sens constitue les trois composantes fondamentales de la notion de type.

- Le niveau des propriétés permet de structurer l'espace des types sur la base de propriétés abstraites communes aux différents types. Si l'orthogonalité de la composition est souhaitable, sinon indispensable, au niveau des constructeurs du langage de représentation, il n'en est plus de même au niveau des types abstraits génériques. Les propriétés apparaissent comme le moyen de restreindre les compositions de types, un peu comme ces derniers restreignent les paramètres possibles d'une procédure.

- Au niveau des types abstraits génériques, on trouve toute l'information concernant le comportement des objets. Par les définitions de types (et les définitions de fonctions associées) le programmeur se définit une machine abstraite au niveau de laquelle il exprimera ses solutions.

- C'est seulement au niveau de la représentation que sont traités les problèmes de l'implantation des objets et des fonctions qui les manipulent. Ce niveau est évidemment caché à l'utilisateur d'une abstraction, l'interface étant, comme nous l'avons vu, constitué de l'ensemble des fonctions d'accès.

Cette caractérisation des types abstraits génériques nous amène à insister dans cette conclusion sur deux points importants, l'un concerne l'aspect "système de programmation" d'un langage disposant d'un tel mécanisme, l'autre les possibilités méthodologiques[JAC78], [PEY78] offertes.

Aspect "système de programmation"

A l'heure actuelle,il ne viendrait à l'idée de personne de contester la nécessité des bibliothèques de sous-programmes. Leur très large utilisation, aussi bien dans le domaine du calcul scientifique que dans celui de la gestion, est prouvée de longue date. Il est en effet très rare, à l'heure actuelle, que l'on réécrive un programme de résolution de système linéaire ou de tri de bande. Le système que nous nous proposons d'implanter transpose cette même idée au niveau des types.

Pour qu'un type "mérite" d'être inclu dans une bibliothèque de ce genre il est évidemment nécessaire que sa définition dispose d'un degré de paramétrisation assez poussé, ceci de façon à toucher le maximum d'utilisateurs. Les types génériques répondent à cette contrainte tout en assurant un maximum de traitement statique.

En effet, les fonctions d'un type générique catalogué en bibliothèque, seront compilées partiellement, les indéterminées (dépendant du ou des types passés en paramètre) seront localisées et résolues dans un programme particulier par l'intermédiaire des propriétés. D'autre part, la cohérence globale d'une telle bibliothèque (non-ambiguité des appels, non-infinité dynamique de type, cf.III.3) peut être assurée au moment de sa création (ou de sa modification).

Parallèlement à l'approfondissement des notions présentées dans cette thèse, la prochaine étape de notre travail va consister à implanter une version expérimentale, facilement transportable [BER78c] de ce système.

Impact méthodologique

En reprenant la discussion entamée dans le chapitre II, nous allons essayer de montrer sur un exemple, l'apport de la généricité en ce qui concerne la méthodologie de programmation.

Considérons la structure de donnée "arbre binaire", il est clair qu'une telle structure va être décrite par un type abstrait générique:

type arbrebin (type t) ;

où t est le type des valeurs associées aux noeuds.

Dans un premier temps on ne s'intéresse évidemment pas à la nature de t, ni même à la représentation d'un objet arbre. L'utilisation de arbrebin se fait par les fonctions d'accès génériques:

$$\begin{aligned}
\textit{fils_droit} &: \textit{arbrebin}(t) \rightarrow \textit{arbrebin}(t) \\
\textit{fils_gauche} &: \textit{arbrebin}(t) \rightarrow \textit{arbrebin}(t) \\
\textit{existe_fils_droit} &: \textit{arbrebin}(t) \rightarrow \textit{bool} \\
\textit{existe_fils_gauche} &: \textit{arbrebin}(t) \rightarrow \textit{bool} \\
\textit{valeur courante} &: \textit{arbrebin}(t) \rightarrow t
\end{aligned}$$

L'effet de ces fonctions étant défini, même de manière informelle, il sera alors possible d'écrire, par exemple, les algorithmes des différents parcours de l'arbre: postfixé, infixé, préfixé.

Si maintenant on s'intéresse à la construction d'un arbre binaire ordonné, il est alors nécessaire de disposer des fonctions d'insertion d'un élément:

$$\begin{aligned}
\textit{insérer_droit} &: \textit{arbrebin}(t) \times t \rightarrow \textit{neutre} \\
\textit{insérer_gauche} &: \textit{arbrebin}(t) \times t \rightarrow \textit{neutre}
\end{aligned}$$

Mais on s'aperçoit également que t ne peut pas être quelconque puisqu'il doit satisfaire la propriété d'être ordonné, c'est ce que traduira l'instruction:

exige t : ordretotal (≤) ;

C'est seulement plus tard, dans un programme particulier de construction d'un arbre binaire ordonné où le type t est, par exemple, un couple

(entier, chaîne) que le programmeur devra se préoccuper de définir l'ordre sur ce type effectif.

Après les spécifications fonctionnelles de arbrebin le programmeur peut choisir une (ou plusieurs) représentation(s) pour les objets ; il écrira alors toutes les fonctions d'accès qui manipulent cette (ou ces) représentation (s).

L'intérêt d'une telle démarche est évident aussi bien dans l'activité de programmation que dans son enseignement. Elle permet de décomposer l'acquisition de concepts complexes en étapes comportant chacune un nombre minimum de problèmes à résoudre. Le fait que le langage permette non seulement d'exprimer clairement à chaque niveau les solutions élaborées mais en plus guide la décomposition, nous semble fondamental.

BIBLIOGRAPHIE

[AMB76] A.AMBLET et al
"GYPSY: A language for specification and implementation of verifiable proragms"
University of Texas, Austin, Texas, 1976.

[BAN78] M.BANATRE, A.COUVERT, D.HERMAN, M.RAYNAL
"Présentation et évaluation du projet SOC: un système d'objets typés conservés"
IRISA PI/094, Université de Rennes, avril 1978.

[BER73] D.BERT
"Etude d'éléments fondamentaux des langages de programmation"
Thèse de 3ème cycle, U.S.M.GRENOBLE, mai 1973.

[BER77a] D.BERT, P.JACQUET
"Generic Abstract Data Types"
Proc 5th annual III Conference organized by WG2.1(IFIP), Guidel, May 1977

[BER77b] D.BERT, P.JACQUET
"Types abstraits génériques et multireprésentation"
GROPLAN, Bulletin n°2 (octobre 1977) pp.41/47, Groupe Programmation et langages, AFCET, DIV.TTI.

[BER78a] D.BERT, P.JACQUET
"Some validation problems with parameterized types and generic functions"
3ème Colloque international sur la programmation, organisé par l'Institut de Programmation, PARIS, mars 1978.

[BER78b] D.BERT, P.JACQUET
"Generic abstract data types, part two: Some validation problems"
Rapport de recherche, Laboratoire IMAG (à paraître), U.S.M.GRENOBLE.

[BER78c] D.BERT, D.BORRIONE, B.DAVID, M.DELAUNAY, P.JACQUET, M.SIMONET
"FORTISH: Une extension de FORTRAN vers un système de programmation de haut niveau".
Rapport de recherche, Laboratoire IMAG (à paraître), U.S.M. GRENOBLE.

[BET77] C.BETOURNE, L.FERRAUD, J.JOULIA, A.MOURADI, J.M.RIGAUD
"Le langage LEST"
Rapport de contrat SESORI, Toulouse, décembre 1977

[BOH66] BOHM and JACOPINI
"Flow diagrams, Turing machines and languages with only two formation rules"
CACM 9,5, mai 1966

[BOO76] H.J.BOOM
"Extended type checking"
New directions in algorithmic languages, 1976
Prepared for IFIP WG2.1, edited by S.A.Schumann, IRIA

[CHE66] T.E.CHEATMAN Jr.
"The introduction of definitional facilities into higher level programming languages"
AFIPS Conference proceedings, 1966 FJCC, vol.29, 2nd edition, pp.623/637.

[CHU41] A.CHURCH
"The calculi of lambda-conversion"
Annals of mathematical studies n°6, Princeton U.Press,Princeton, N.J., 1941

[COU74] J.COURTIN, J.VOIRON
"Introduction à l'algorithmique et aux structures de données"
Cours IUT II, Département Informatique, Grenoble, 1974

[COU75] P.COUSOT, R.COUSOT
"Static determination of dynamic properties of programs"
Colloque International sur la programmation, Paris 13/15 avril 1976, Dunod.

[CUN78] P.Y.CUNIN, M.GRIFFITHS, P.C.SCHOLL
"Aspects fondamentaux du langage MEFIA"
Journées d'études sur la fiabilité des logiciels dans les applications industrielles, EDF, Clamart, 26/27 avril 1978

[DAH68] O.DAHL, B.MYHRAUG, K.NIGAARD
"STIMULA 67: common base language"
Norwegian Computing Center, n° S-2, may 1968

[DES37] R.DESCARTES
"Discours de la méthode"
1637

[DIJ68] E.W.DIJKSTRA
"GOTO statement considered harmful"
CACM 11 (mars 1968), Letter to the editor

[FIS73] A.E.FISCHER, M.J.FISCHER
"Mode modules as representations of domains"
Proc. ACM Symposium on principle of programming languages, Boston 1973.

[FLO67] R.W.FLOYD
"Assigning meanings to programs"
Proc. Symp. in applied mathematics, vol.19 (J.T.Schwartz, ed.)
American Society, 1967 (pp.31/42).

[FOS77] J.M.FOSTER, P.D.FOSTER
"Abstract Date and Functors"
SIGPLAN Notices, vol.12, n°6, Juin 1977

[GOG73] J.A.GOGUEN, J.W.THATCHER, E.G.WAGNER, J.B.WRIGHT
"A junction between computer science and category theory"
IBM Research report, part I, RC 4526, 1973.

[GOG75] J.A.GOGUEN, J.W.THATCHER, E.G.WAGNER, J.B.WRIGHT
"Abstract data types as initial algebras and correctness of data representations
Proceeding conference on computer graphics, Pattern recognition and data
structure. Mai 1975

[GOL73] J.GOLDBERG(ed)
"Proceeding of a symposium on the high cost of software"
SRI, September 1973

[GRI77] D.GRIES, N.GEHANI

"Some ideas on data types in high-level languages"

CACM, vol. 20,6, June 1977, pp.414/420.

[GRI73] M.GRIFFITHS

"Relationship between definition and implementation"

Advanced course in software engineering, 1973, Springer Verlag.

[GUT76a] J.V.GUTTAG, E.HOROWITZ, D.R.MUSSER

"Abstract data types and software validation"

USC/Information science Institute, RR 76/48, Août 1976.

[GUT76b] J.V.GUTTAG, E.HOROWITZ, D.R.MUSSER

"The design of data type specifications"

USC/Information science Institute, RR 76/49, November 1976.

[HOA69] C.A.R. HOARE

"An axiomatic basis for computer programming"

CACM 12, 10, October 1969, pp.576/580, 583.

[HUC77] B.HUC

"Mise en oeuvre de la méthode de programmation déductive"

Thèse de docteur ingénieur, Université de NANCY I,1977

[HUE76] G.HUET

"Résolution d'équations dans les langages d'ordre 1,2,..., ω."

Thèse d'Etat, PARIS VII, septembre 1976

[IRO77] "IRONMAN"

Departement of Defense requirements for high order computer programming languages

Revised "IRONMAN", Dept. of Defense USA, July 1977

[JAC78] P.JACQUET

"La généricité comme outil d'abstraction dans les langages de programmation"

Congrès AFCET, Paris, Novembre 1978

[JOR72] Ph.JORRAND, D.BERT

"On some basic concepts for extensible programming languages"

ICS, Venise, April 1972, pp.2/16.

[JOR75] Ph.JORRAND

"Contribution au développement des langages extensibles"

Thèse d'Etat, U.S.M.Grenoble, Janvier 1975.

[KNU71] D.E.KNUTH, R.W.FLOYD

"Notes on avoiding GOTO statements"

IPL 1(1971) pp.23/31, North-Holland publishing company.

[KNU74] D.E.KNUTH

"Structured programming with GOTO statements"

STAN-C7-74-416, Standford University, May 1974

[KRI78] B.KRIEG-BRUCKNER

"Concrete and abstract specification, modularization and program development by transformation"

TUM. INFO-7805, January 1978, MUNCHEN.

[LAM77] B.W.LAMPSON et al.

"Report on the programming language EUCLID"

SIGPLAN Notices, volume 12, number 2, February 1977

[LAN65] P.J.LANDIN

"A correspondance between ALGOL 60 and Church's lambda notation"

CACM 8,2 ; 3, pp.89/101, 158/165, February 1965

[LIN] C.H.LINDSEY

"Différents articles sur les "modals""

ALGOL Bulletin, 37.4.3.

[LIS75] B.LISKOV

"An introduction to CLU"

New directions in Algorithmic languages, 1975

Prepared for IFIP WG2.1 edited by S.A.Schumann, IRIA

[MIT77] J.G.MITCHELL, B.WEGBREIT

"Schemes: a high level data structuring"

XEROX, CSL-77-1, January 1977

[NAU60] P.NAUR (ed)

"Revised ALGOL report"

CACM, january 1963

[PAR71] D.L.PARNAS

"Information distribution aspect of design methodology"

IFIP Congress 1971, Booklet TA-3 (pp. 26/30)

[PEY78] J.P.PEYRIN, M.DELAUNAY

"Conception d'algorithmes et langages"

Rapport de recherche n°120, Laboratoire IMAG, U.S.M.Grenoble.

[ROU77] R.ROUSSEAU

"HELOISE: un langage et une méthode pour la description fonctionnelle de grands logiciels séquentiels"

Thèse de spécialité, Université de NICE, 1977.

[RUS08] B.RUSSEL

"Mathematical logic as based on the theory of types"

Amer.J.Math., 30, pp. 222/262, 1908

[SCH75] S.A.SCHUMANN

"On generic functions"

New directions in algorithmic languages, 1975

Prepared for IFIP WG2.1, edited by S.A.Schumann, IRIA

[SOL78] R.SOLER
"Description des étapes de compilation d'une bibliothèque de types génériques"
Rapport de D.E.A., U.S.M.Grenoble, Septembre 1978

[STA67] T.A.STANDISH
"A data definition facility for programming languages"
Doctoral dissertation, Carnegie Mellon University, 1967

[TEN77] R.D.TENNENT
"On a new approach to representation independant class"
Acta Informatica, vol.8, fasc.4, 1977

[vWI77] A. van WIJNGAARDEN et al.
"Revised report on the algorithmic languages ALGOL 68"
SIGPLAN Notices, vol.12, n°5, May 1977.

[VID74] J.VIDART
"Extensions syntaxiques dans un contexte LL(1)"
Thèse de 3ème cycle, U.S.M. Grenoble, septembre 1974

[VAU70] B.VAUQUOIS
"Calculabilité des langages"
Cours Université de Grenoble, 1970.

[WEG70] B. WEGBREIT
"Studies in extensible languages"
Harvard University, May 1970, n° ESD-TR-70-297

[WIR71] N.WIRTH
"The programming language PASCAL"
Acta Informatica, 1, pp. 35/65, 1971.

[WIR76] N.WIRTH

"MODULA: a language for modular multiprogramming"

Institüt für Informatik, ETH, CH 8092 Zurich, mars 1976

[WUL75] W.A.WULF, R.L.LONDON, M.SHAW

"Abstraction and verification in ALPHARD"

New directions in algorithmic languages, 1975

Prepared for IFIP WG2.1, edited by S.A. Schumann, IRIA

[ZIL75] S.N.ZILLES

"Algebraic specification of data types"

MIT, Project MAC, Progress report 11, 1975.

Dernière page d'une thèse

VU

Grenoble, le 4 Septembre 1978

Le Président de la thèse

Professeur C. BOLLET

Vu, et permis d'imprimer,

Grenoble, le 1 Septembre 1978

Le Président de l'Université
Scientifique et Médicale

UNIVERSITÉ SCIENTIFIQUE ET MÉDICALE DE GRENOBLE