ÉCOLE NATIONALE SUPÉRIEURE
DES TÉLÉCOMMUNICATIONS

THESE

présentée par

Philippe KRUCHTEN

pour le

DOCTORAT de L'E.N.S.T.

Spécialité : Informatique

le 2 octobre 1986

Une Machine Ada® Virtuelle : Architecture

Président du Jury : Guy PUJOLLE
Directeur de Thèse : Robert B. K. DEWAR
Rapporteurs : Jean-François DUFOURD
Etienne MOREL
Examinateurs : Christine NORA, Dominique PIERRE
Olivier ROUBINE, Jean-Luc STEHLÉ

Je tiens à remercier monsieur le Professeur Guy PUJOLLE pour m'avoir fait l'honneur d'accepter de présider le jury de thèse, madame le Professeur NORA, messieurs les Professeurs Jean-François DUFOURD et Jean-Luc STEHLE et messieurs Etienne MOREL, Olivier ROUBINE et Dominique PIERRE pour m'avoir fait l'honneur d'en faire partie.

Je dois une reconnaissance toute particulière à Robert B. K. DEWAR, Edmond SCHONBERG et toute l'équipe de NYUADA pour l'accueil très chaleureux qu'ils m'ont réservé. Je garderai un excellent souvenir de mon séjour au Courant Institute et à New York University.

Merci à mes amis qui m'ont aidé à mener à bien ce travail, et notamment Jean-Pierre, Armand et Lisa.

Merci enfin au Département Informatique de l'E.N.S.T., à l'ARECOM (Association pour l'enseignement et la recherche dans les communications) et à la société Tradal pour leur soutien moral et financier.

A mes parents,

à Sylvie et Nicolas

Plan

Index des figures

Introduction

1. Le langage de programmation Ada®

1.1 Historique

En janvier 1975, le Ministère Américain de la Défense (DoD) formait un comité d'experts, le High Order Language Working Group (HOLWG), avec pour mission d'essayer de trouver une approche systématique aux problèmes de qualité et de coûts des logiciels militaires. Il faut savoir en effet que le DoD est le plus grand consommateur de logiciels du globe et il utilisait à ce moment-là pour ce faire environ 450 langages de programmation et dialectes différents. La plus grosse part de ces logiciels, ou tout du moins celle pour laquelle les frais de maintenance et de portage étaient les plus importants, se trouvaient dans le domaine des système dits «embarqués» (*embedded*), c'est-à-dire des systèmes informatiques qui ne sont qu'un composant d'un système plus vaste : systèmes d'armes, de radar, de télécommunications,... Développer un langage de programmation universel unique apparut alors être une des solutions à beaucoup de ces problèmes, et le HOLWG produisit une succession de cahiers des charges précisant les caractéristiques souhaitables d'un tel langage. Ces rapports, connus sous les noms de *Strawman*, *Woodenman*, *Tinman*, *Ironman*, *Revised Ironman*, ont suscité un intérêt considérable tant au sein du DoD qu'à l'extérieur et chacun a été successivement examiné et repensé et modifié jusqu'à aboutir au cahier des charges *Steelman* [DoD 78].

Au printemps de 1977, dix-sept organismes ont répondu à l'appel d'offres du DoD pour la conception de ce nouveau langage, et après quelques mois quatre d'entre eux ont été retenus pour une pré-étude : Softech,

® Ada est une marque déposée du Gouvernement des Etats-Unis d'Amérique, Ada Joint Program Office

Intermetrics, S.R.I. International et Cii-Honeywell-Bull. En mars 1978, il ne restait plus en lice que deux contractants : Intermetrics et Cii-Honeywell-Bull, l'équipe française dirigée par Jean Ichbiah remportant finalement l'appel d'offres un an plus tard, le 2 Mai 1979. Le "langage vert" est alors rebaptisé *Ada,* du nom de Ada Byron, fille du poète anglais, considérée comme le premier programmeur de l'histoire pour ses travaux sur la *Machine Analytique Mécanique* de Charles Babbage. Néanmoins du travail restait encore à faire, et ce n'est qu'en janvier 83 que le langage atteignait sa forme définitive et était normalisé. (Sources : [Barnes 84]).

1.2. Présentation sommaire du langage

Ada est un langage algorithmique d'une puissance d'expression considérable, dérivé de Pascal dont il a retenu les structures de contrôle et certains types de données, mais avec en plus des possibilités d'encapsulation des données, de modularité logique et physique, de modélisation de tâches parallèles et de traitement de situations exceptionnelles. Ada n'est pas à notre avis un langage de *très* haut niveau, pas du niveau de LISP, SNOBOL, SETL, APL ou PROLOG en tout cas, mais c'est un langage qui a un spectre sémantique assez vaste et dont les apports originaux sont surtout du domaine du génie logiciel : fiabilité, maintenabilité, modularité, portabilité, mécanisme de généricité, permettant l'écriture et la réutilisation de *composants logiciels.*

Le langage est défini par un gros document : le «Reference Manual for the Ada Programming Language» qui constitue la norme américaine (ANSI/MIL-STD 1815 A), et auquel nous ferons abondamment référence dans cette thèse sous l'appellation consacrée par l'usage de «LRM» [DoD 83].

2. Le projet NYUADA

Le projet NYUADA a démarré en 1978 au sein du département d'informatique du Courant Institute of Mathematical Sciences à New York University, sous l'égide du professeur Robert Dewar, de Gerald Fisher et d'Edmond Schonberg, financé par l'armée de terre des Etats-Unis (contrat n°DAAB027-82-K-J196 du CORADCOM à Fort Monmouth, N.J.) et par l'Ada Joint Program Office au Pentagone.

L'objectif initial du projet était de faire une petite étude des problèmes d'optimisation posés par le langage Ada. Il s'est très vite avéré à l'époque que le langage était assez mal défini par [Ichbiah 79a, Ichbiah 79b] et qu'une spécification plus rigoureuse était indispensable si l'on voulait aborder les problèmes d'optimisation. Ceci a conduit à un premier *modèle sémantique exécutable* d'Ada consistant en approximativement 2000 lignes de SETL, décrivant l'aspect sémantique dynamique sous forme d'un interprète exécutant une représentation intermédiaire du langage appelée AIS (*Ada Intermediate Source*). Deux mois plus tard un analyseur lexical et syntaxique y était ajouté, permettant de générer l'AIS automatiquement, mais seulement pour des programmes Ada corrects. Puis au cours des 6 mois qui suivirent, l'analyse de la sémantique statique du langage y a été progressivement ajoutée, et lorsque la définition d'Ada 1980 a été publiée [DoD 80], le traducteur Ada/Ed («Ed» pour «Educational») était déjà un modèle exécutable d'une majeure partie de sa sémantique statique et dynamique. Plusieurs des aspects du langage qui n'étaient pas implémentables au vu de leur définition en «Preliminary Ada» (comme les discriminants et la dérivations des sous-programmes) avaient été laissés de côté au début, puis ajoutés lors de la parution de l'avant-projet de norme en 1980. Par contre, la sémantique des processus parallèles : activation des tâches, rendez-vous, instructions "abort" et "select", avait été complètement réalisée dès le début [Dewar 80].

A ce point, il est devenu clair aux yeux de tous qu'Ada/Ed pouvait remplir deux fonctions : celle de *traducteur prototype,* et celle de *définition formelle* du langage, cette dernière étant une conséquence directe :

a) de la concision et de la lisibilité du système (25 000 lignes de SETL, documentation comprise) [NYU 83a, NYU 83b],

b) du modèle très abstrait choisi pour représenter l'environnement d'exécution [Kruchten 84],

c) de la démonstration de la conformité d'Ada/Ed à la suite de validation A.C.V.C. (*Ada Compiler Validation Capability*), développée par J.B. Goodenough et son équipe à Softech, qui constitue *de facto* une définition supplémentaire d'Ada [Goodenough 81].

L'effort de NYUADA s'est donc poursuivi dans ces deux directions après notre arrivée dans l'équipe. En à peu près 6 mois, Ada/Ed a été modifié

pour s'adapter aux modifications apportées par le nouveau manuel de référence publié en Juillet 1982 [DoD 82], puis à nouveau au début de 1983 pour refléter les dernières modifications contenues dans la norme ANSI de Janvier 1983 [DoD 83]. Finalement Ada/Ed a été le premier compilateur Ada a être officiellement validé par l'AJPO (*Ada joint Program Office*) le 11 Avril 1983. De son démarrage en 1979 jusqu'à la validation, le développement d'Ada/Ed n'aura coûté que 100 hommes-mois.

La preuve ultime de l'utilité d'un prototype est bien entendu la construction d'un système de production *grandeur nature* qui utilise le prototype comme modèle. Après la validation, le projet s'est engagé dans la voie de la construction d'un traducteur considérablement plus performant, écrit d'abord en SETL, puis finalement en C, utilisant le premier système, constamment tenu à jour, comme spécification de réalisation. La figure 1 ci-dessous montre comment le système a évolué du prototype initial en SETL au système opérationnel final en C, en ne faisant évoluer que certains éléments à la fois, avec des interfaces fixes, bien définies, de façon à avoir constamment un système exécutable et vérifiable [Schonberg 86].

3. Objectifs de notre étude

L'objectif principal de notre étude était de réaliser rapidement et de façon la plus économique possible un système de traduction portable, complet et validable pour le langage de programmation Ada, tel qu'il est décrit dans le LRM.

Ce système doit pouvoir être utilisé pour l'enseignement d'Ada et l'expérimentation à petite échelle. Il doit être simple d'emploi, robuste et bon marché pour lui assurer une large diffusion.

Il doit dans une deuxième étape servir de base pour une implémentation plus efficace destinée à des micro-ordinateurs personnels, genre IBM PC.

Il n'était pas dans nos objectifs de réaliser un système *performant,* le seul critère de performance que nous nous étions fixé était de gagner deux ordres de grandeur (soit un facteur 100) en temps d'exécution par rapport au système Ada/Ed originel, celui validé en Avril 1983.

Ces objectifs méritent quelques explications et imposent d'emblée quelques choix de conception :

- Une réalisation rapide et économique : le projet NYUADA n'est constitué que d'une demi-douzaine de personnes, dont la moitié ne travaillent pas à plein temps sur ce projet; d'autre part une partie de la force de travail devait être consacrée à la maintenance du système Ada/Ed initial.

- Un système destiné à l'enseignement et à l'expérimentation : c'est à la fois la vocation du Courant Institute (qui n'est pas une société commerciale de logiciel, mais un institut de recherche au sein d'une université) et l'objectif visé par l'organisme qui a supporté financièrement ce projet, à savoir l'armée de terre des Etats-Unis.

- Portabilité sur des petites machines : afin d'assurer une large diffusion d'un traducteur Ada complet dans les universités, collèges, entreprises, et par là étendre rapidement la connaissance de ce langage, clé de son succès. Il existait un précédent dans ce sens : le compilateur Pascal UCSD (Université de Californie à San Diego).

Pour ce faire, il était donc exclus de recourir à la démarche traditionnelle : écriture *ex nihilo* d'un compilateur complet, dans un langage d'implémentation existant (Pascal, C,...), avec des générateurs de code et des Noyaux Exécutifs pour diverses machines cibles. Nous avons choisi de faire évoluer progressivement le système initial, en définissant une machine virtuelle pour Ada, taillée sur mesure pour ce langage, en écrivant un *prototype exécutable d'interprète* pour simuler cette Machine Ada, et en greffant sur le système Ada/Ed existant un *générateur de code* pour cette machine. Puis après validation, les différents sous-ensembles du système ont pu être individuellement optimisés et retranscrits dans un langage d'implémentation de plus bas niveau, en l'occurence le langage C.

Nous allons décrire dans cette thèse la Machine Ada virtuelle que nous avons conçue, en essayant de justifier nos choix soit par rapport au langage

Ada, dont elle essaie d'intégrer tous les traits de la sémantique dynamique, soit par rapport aux objectifs énumérés ci-dessus.

A mi-chemin de la conception de cette Machine Ada Virtuelle, devant l'ampleur du travail à accomplir, nous avons invité J. P. Rosen de l'E.N.S.T. à nous rejoindre au Courant Institute pour s'occuper en particulier du sous-système de gestion des tâches, des entrées-sorties et de l'arithmétique réelle à point-fixe. Ses travaux, qui ne sont que sommairement évoqués ici, font l'objet de sa propre thèse de doctorat, présentée conjointement à la nôtre [Rosen 86].

4. Organisation de ce document

Une part importante de ce document est la traduction en français de la documentation que nous avons laissée aux membres du projet NYUADA pour leur permettre de compléter et de maintenir le prototype existant, et d'en comprendre les mécanismes pour pouvoir les transporter dans un langage de plus bas niveau. Certains textes sont repris de diverses publications en anglais ou en français, *cf.* [Kruchten 83, Kruchten 84b, Kruchten 85].

Les chapitres 1 à 10 décrivent la Machine Ada virtuelle : structure générale, modèle de mémoire et adressage, représentation des objets et valeurs, et les mécanismes plus spécifiques à Ada, tels que l'élaboration des types, le traitement des exceptions, les sous-programmes, tâches et paquetages. Le chapitre 9 résume la gestion des tâches, développée dans [Rosen 86]. Le chapitre 11 décrit le «dorsal» du compilateur Ada/Ed-2, constitué de l'expanseur, du générateur de code et du relieur, et enfin l'interprète qui simule et modélise la Machine Ada.

Nous conclurons en insistant sur un aspect méthodologique important, le prototypage rapide de logiciels à l'aide de langage de très haut niveau.

En annexe, on trouvera une description de notre outil de travail principal, déjà mentionné à plusieurs reprises : le langage de programmation SETL.

Nous ferons abondamment référence au *Language Reference Manual* [DoD 83] (ou Manuel de Référence du Langage Ada) sous la forme désormais consacrée par l'usage : [LRM c.s.p(a)], où **c**, **s**, **p** et **a** représentent respectivement les numéros de chapitre, de section, de paragraphe et d'alinéa. Notez que ces références s'appliquent de façon identique à l'avant-projet de norme française NF Z-65-700 [Afnor 86].

En ce qui concerne les termes propres au langage Ada, nous avons essayé d'utiliser leur traduction «officielle» en français, établie par le groupe d'experts de l'Afnor chargé de la norme française, norme dont nous avons eu l'honneur de préparer le texte initial. On trouvera les plus importants de ces termes expliqués dans le glossaire en annexe.

Pour les descriptions d'algorithmes et les définitions de structures de données, nous avons cherché autant que possible à utiliser le langage Ada lui-même ; mais souvent le niveau auquel se situe la description a imposé de faire des entorses au langage et d'utiliser un «simili-Ada» à la sémantique plus souple, ce dont nous espérons que le lecteur averti ne nous tiendra pas trop rigueur.

La figure 1 page suivante montre, du haut vers le bas, les différentes étapes de l'évolution du projet NYUADA. Les parties qui font l'objet de cette thèse sont en léger grisé.

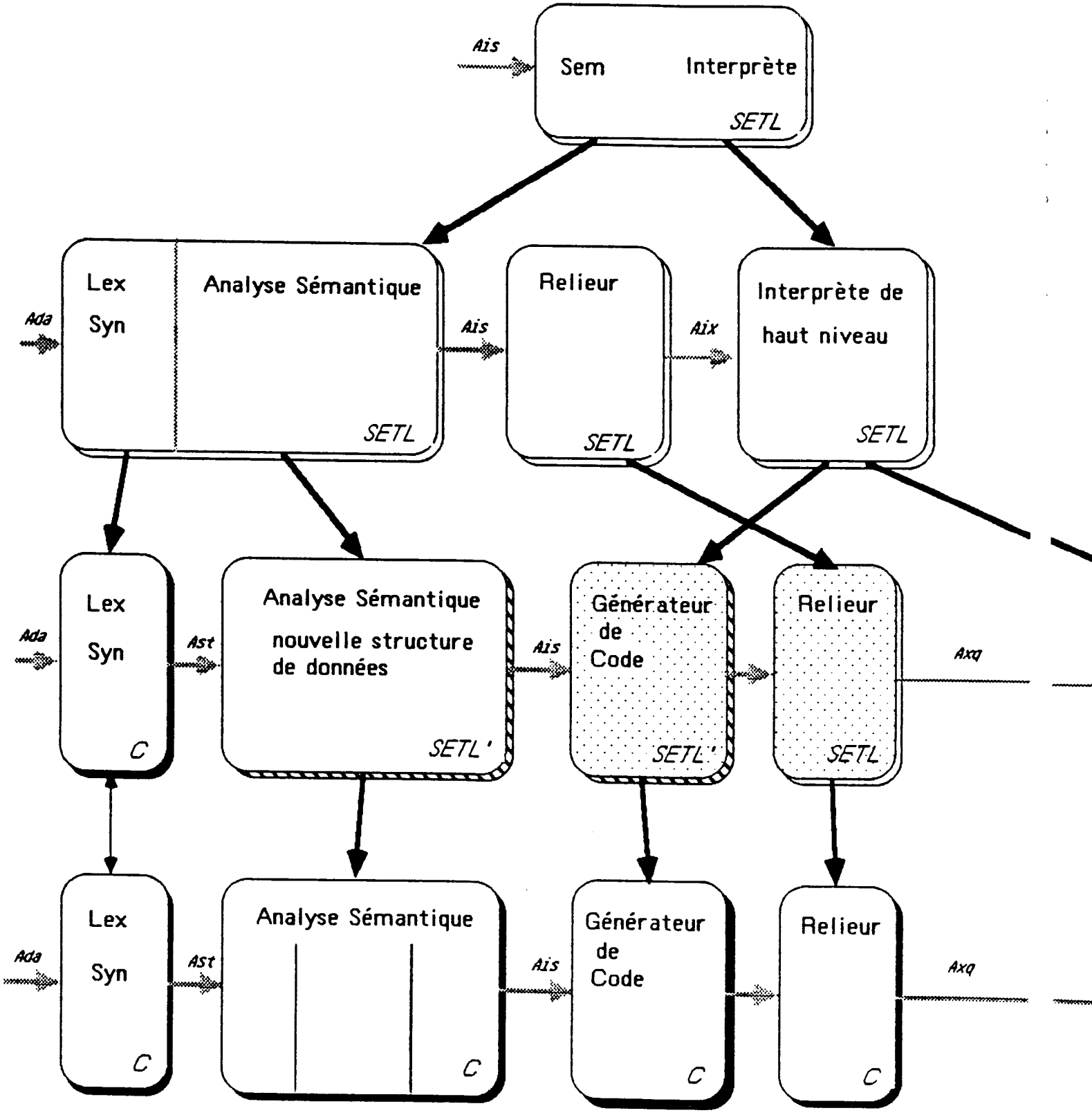


Figure 1 : le système Ada/ED

Etape 0

1979
2000 lignes
2–3 homme–mois?

Etape 1

1980–83
25000 lignes
Ada complet
Validé en Avril 83
100 homme–mois

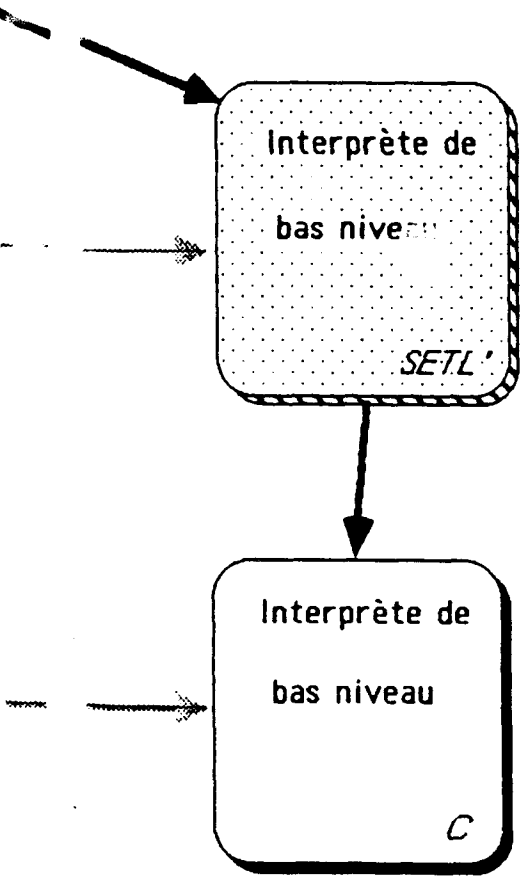
Légende:

Lex = Analyse lexicale
Syn = Analyse syntaxique
Sem = Analyse sémantique

SETL = Programme en SETL
SETL' = Programme en SETL de bas niveau
C = Programme en C

Ada = source en Ada
Ast = Abstract Syntax Tree
Ais = Ada Intermediate Source
Aix = Ada Intermediate eXecutable
Axq = Ada eXecutable code

→ Transformation de Programmes, Evolution
⇢ Flot de données



Etape 2

1983–1984
35000 lignes
Ada complet
Exécution: x100
Compilation: x2–3
30 homme–mois

Etape 3

1984–86
50000 lignes
Exécution: x5
Compilation: x2
30 homme–mois

de la première esquisse au produit final.

Chapitre 1.
Présentation de la Machine Ada

Dans ce premier chapitre nous introduisons l'architecture générale de la Machine Ada et celles de ses caractéristiques qui ne méritent pas plus d'explications. Les chapitres ultérieurs reprendront pour les développer les points les plus importants ou les plus originaux.

1. Architecture

La machine Ada est un processeur somme toute très classique d'apparence ; c'est une machine type Von Neumann, comportant :

- une *unité centrale*, avec des registres, et exécutant séquentiellement un programme composé d'instructions ;
- une *mémoire*, contenant ce programme et les données qu'il manipule, adressée comme un vecteur linéaire.

Bien que la machine supporte les tâches d'Ada, c'est un mono-processeur, exécutant un unique flot d'instructions et donc une et une seule tâche est active à un instant donné.

La mémoire est partagée en *segments* que l'on peut manipuler individuellement et que l'on peut loger n'importe où dans la mémoire. Un segment est un ensemble d'unités de mémoire contiguës de longueur variable, limitée par la taille des registres. Cette segmentation facilite l'exécution de gros programmes sur de petites machines en permettant un mécanisme de transfert simple entre un support de mémoire secondaire

comme un disque et une mémoire «centrale» de taille réduite. Dans le cas de la machine Ada, nous verrons que ces segments sont complètement «translatables» et peuvent être implantés n'importe où en mémoire.

Le lien entre l'unité de traitement, les segments et la mémoire se fait au moyen d'un dispositif de *topographie de mémoire*, blocs de registres qui assurent la traduction entre une adresse exprimée en termes de segment et déplacement dans ce segment d'une part, et l'adresse absolue en mémoire d'autre part.

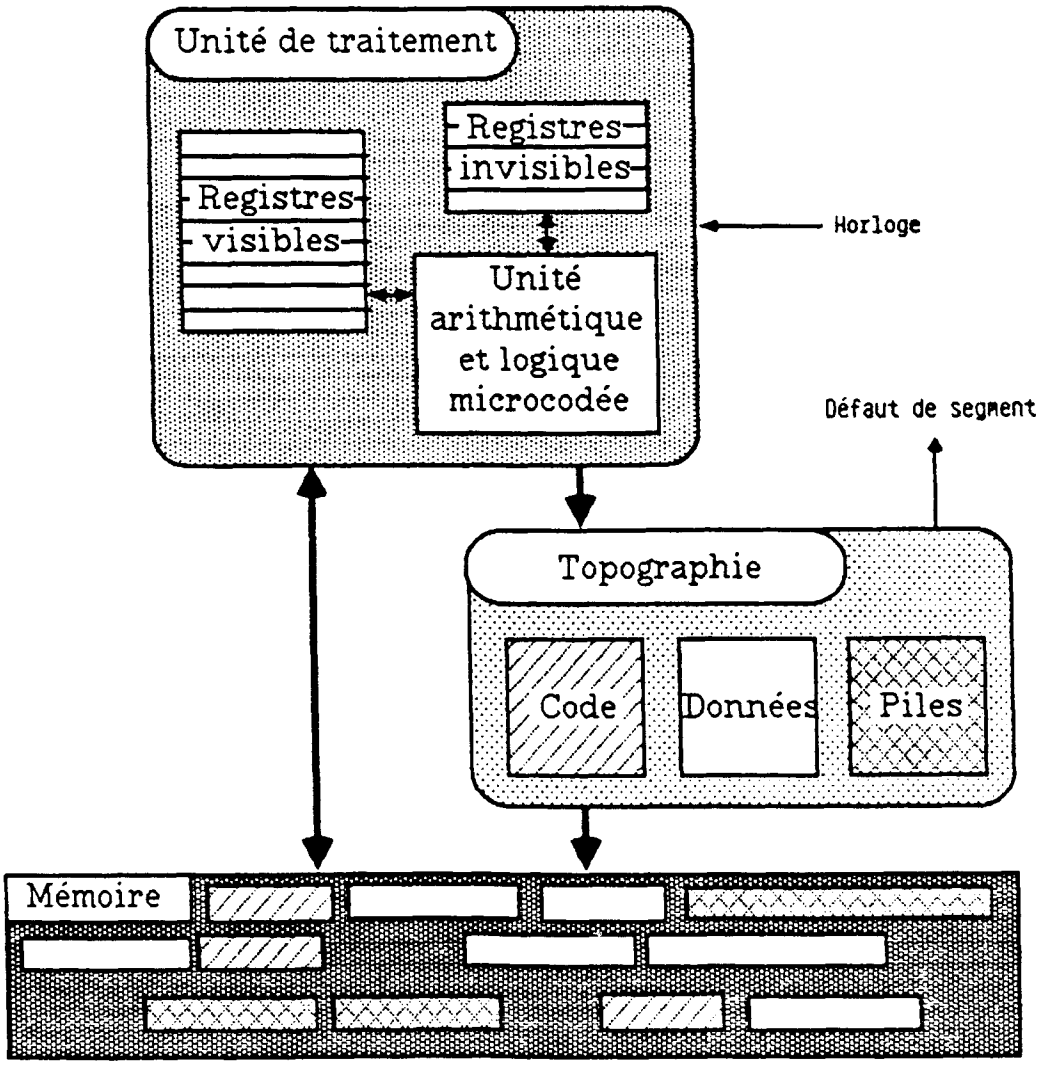


Figure 2 - Architecture générale

2. L'unité de traitement et les registres

L'unité de traitement contient un certain nombre de registres. On n'y trouvera ni accumulateur, ni drapeaux, ni registres de travail banalisés ; toutes les opérations arithmétiques et logiques se font par l'intermédiaire des piles. Les registres sont des registres spécialisés, qui ont presque tous un rôle dans la structuration de la mémoire et les mécanismes d'adressage.

Le registre le plus important est celui qui désigne la tâche qui est en train de s'exécuter ; un premier ensemble de registres est lié à cette tâche courante. Il en constitue le «contexte courant» et il est sauvegardé dans les données propres à la tâche en cas de commutation entre tâches, de «changement de contexte».

◇ **Numéro de tâche :**

Ce registre désigne la tâche courante. Il sert à accéder au segment contenant la pile associée à la tâche. Cette pile contient tout au fond le *bloc de contrôle de tâche*, puis au-dessus l'empilement des *environnements d'appel* correspondants aux appels de sous-programmes. Chaque environnement d'appel contient des données liées à l'appel : liaison entre environnements, paramètres, etc... et une pile d'*environnements de bloc*, correspondants aux «cadres» (*frames*) du langage, dont l'instruction de bloc "declare... begin... end;" est un exemple.

◇ **Base d'environnement d'appel :**

Ce registre contient le déplacement par rapport au fond de la pile de la tâche courante de l'environnement d'appel «actif». C'est par rapport à ce registre que se fait l'adressage des objets *locaux* à l'environnement d'appel.

◇ **Base de l'environnement de bloc :**

Ce registre contient le déplacement par rapport au fond dans la pile de la tâche courante de l'environnement de bloc «actif». C'est par rapport à ce registre que l'on accède aux données propres à un bloc : traite-exception, tâches déclarées, etc.

◇ **Sommet de pile :**

Ce registre contient le déplacement par rapport au fond dans la pile courante du premier emplacement mémoire libre. Il sert aux opérations

d'empilement, de dépilement, aux principales opérations arithmétiques et logiques, aux créations d'environnements d'appel et de bloc.

◇ **Numéro de segment de code**, et

◇ **Pointeur d'instruction :**

Ces deux registres servent à désigner l'instruction en cours d'exécution. Le premier désigne le segment de mémoire contenant le code, le second est le déplacement dans ce segment de l'instruction en cours.

◇ **Registre d'exception :**

Ce registre contient l'exception «courante» pour la tâche courante. Il ne sert que lorsqu'une exception est levée et il est manipulé par des opérations spéciales. (cf. Chap. 8)

◇ **Référence au source Ada :**

Utilisé à des fins de traces et de mise au point de programme, ce registre désigne la ligne source Ada en cours d'exécution.

TP	*Task Pointer*	Numéro de tâche
SFP	*Stack Frame Pointer*	Base d'environnement d'appel
BFP	*Block Frame Pointer*	Base d'environnement de bloc
TOS	*Tos Of Stack*	Sommet de pile
CS	*Code Segment*	Numéro de segment de code
IP	*Instruction Pointer*	Pointeur d'instruction
EXR	*Exception Register*	Registre d'exception
LIN	*Line number*	Référence au source Ada

Figure 3 - Bloc des registres visibles,
avec mnémoniques et noms anglais

L'unité de traitement contient en outre des registres internes, «invisibles» ; certains sont liés à la gestion du temps et des quanta de temps alloués aux tâches [Rosen 86], d'autres sont des registres temporaires utilisés par la micro-machine au cours de l'exécution des instructions (cf. chap. 10). Deux registres servent au mécanisme d'allocation dynamique de mémoire dans le tas : la tête de la liste des «morceaux» libres, et le pointeur de fin de tas (cf. §2-7).

3. La topographie de la mémoire :

Les registres suivants ne sont pas associés à une tâche donnée et ne sont donc pas sauvegardés dans la pile associée à la tâche en cas de changement de contexte. A l'instar d'une «carte routière», ils servent à à retrouver les divers segments dans la mémoire, en faisant la traduction : numéro de segment -> adresse absolue en mémoire du début de ce segment. Un indicateur permet aussi de savoir si le segment est effectivement en mémoire ou non, provoquant son chargement sinon. Il y a trois «cartes» distinctes : une pour les segments de pile, une pour les segments de code, et une pour les segments de données, y compris un (ou plusieurs) segment particulier appelé *tas*.

Afin de garantir le «relogement» n'importe où en mémoire de n'importe quel segment, le programme utilisateur ne manipule et ne mémorise aucune adresse absolue en mémoire ; ceci implique que *tous* les accès à la mémoire se font par l'intermédiaire de ces «cartes», par une indirection, d'où l'intérêt d'avoir un accès rapide à leur contenu.

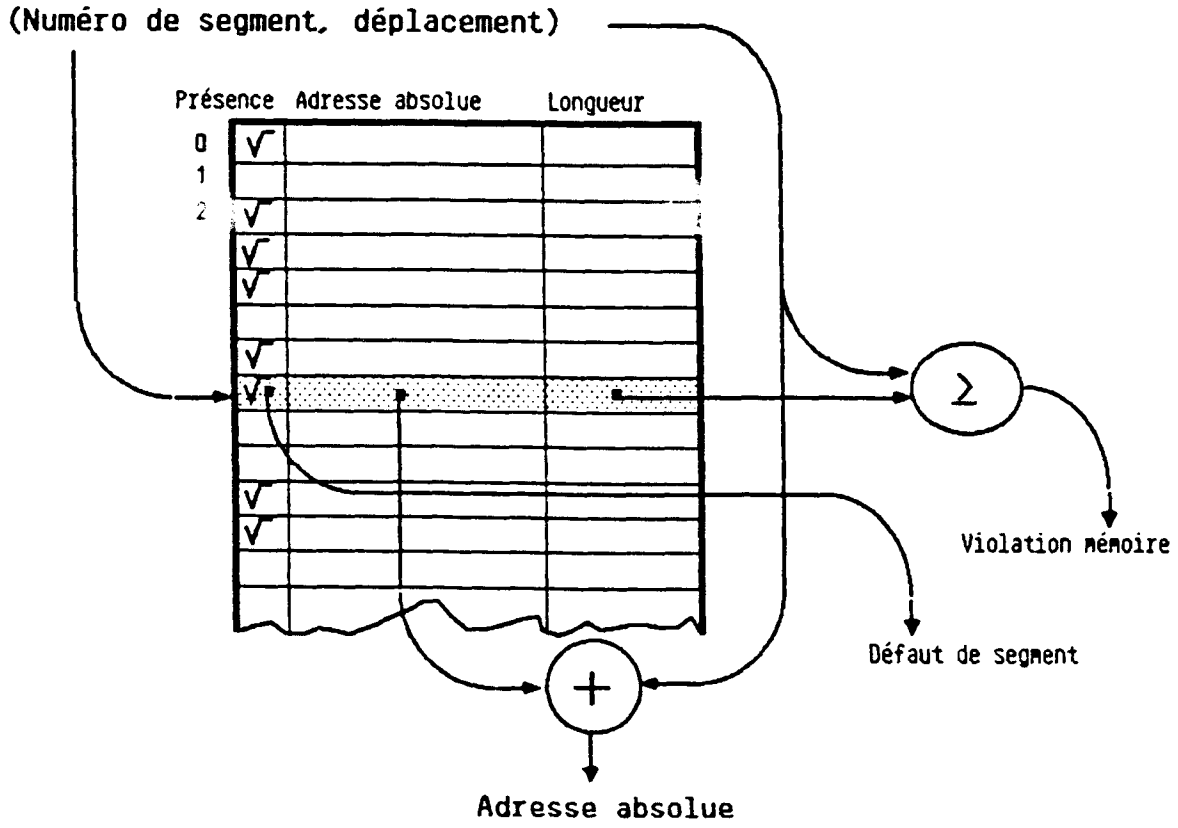


Figure 4 : Un des 3 mécanismes de topographie mémoire

4. Objets, valeurs et types

C'est au niveau de la représentation des objets et des valeurs, associée aux types de ces objets et valeurs que la machine Ada apporte des mécanismes spécifiques. Nous y avons introduit un concept de *patron de type*, objet hybride entre une valeur et un type, qui sert à la création des objets (variables, constantes Ada) et aux tests du respect lors de l'exécution des contraintes portant sur les types. C'est surtout dans le cas des objets à «géométrie variable» (articles avec discriminants, tableaux non contraints) que les patrons de type sont les plus utiles : ces objets sont représentés alors par un doublet valeur+type (cf. chap. 3 et 4).

5. Jeu d'instructions et modes d'adressage

La machine Ada met en œuvre un modèle de mémoire combinant *Piles* et *Tas*. La mémoire est segmentée. Un mécanisme original d'accès aux objets non-locaux d'un sous-programme ou d'une tâche permet de réduire à deux le nombre de modes d'adressage des données : mode "local" ou mode "global" (cf. chap. 7).

Pour permettre aux segments d'être chargés et relogés n'importe où en mémoire physique, ni les programmes, ni les données, ni les registres visibles ne contiennent d'adresses absolues ; l'adressage par l'unité de traitement utilise systématiquement les topographies pour l'accès physique à la mémoire.

Le jeu d'instructions exécutées par l'unité de traitement, s'il a été un peu inspiré par la littérature sur les RISC (Reduced Instruction Set Computer), comporte quand même 114 instructions. Deux tiers de ces instructions sont classiques et très élémentaires : opérations logiques, opérations arithmétiques (sur entiers, réels à point-flottant et réels à point-fixe), opérations de transfert entre pile et mémoire, etc. ; le dernier tiers contient au contraire des instructions très spécifiques à Ada et aux mécanismes imaginés pour la machine Ada : boucles "for", allocateurs, attributs, instructions de traitement des exceptions (cf. chap. 8). Certaines de ces instructions sont très complexes : instructions d'élaboration de type

(cf. chap. 4) ou de sous-programme (cf. chap. 7), ou encore instructions de rendez-vous [Rosen 86], par exemple.

Une part non négligeable dans la complexité des instructions spéciales est liée aux problèmes de *dépendances* : dépendance d'une tâche vis-à-vis de sa tâche mère pour son activation, vis-à-vis de son maître pour sa terminaison ; dépendance d'une variable dynamique de son maître pour sa durée de vie ; dépendance d'un traite-exception vis-à-vis du bloc qui le contient ; dépendance des contraintes d'un composant d'article vis-à-vis des discriminants de l'article, etc.

6. Modèle de la Machine Ada

La Machine Ada n'est pas un pur objet théorique sur le papier ; nous en avons réalisé un modèle complet et exécutable en utilisant le langage de très haut niveau SETL. Ce programme propose une réalisation de tous les éléments de la machine que nous avons énumérés ci-dessus : registres, mémoire, segments, topographies, etc..., et contient un *interprète* pour l'ensemble du jeu d'instructions.

Par ailleurs, pour que ce modèle puisse être utilisé, nous avons réalisé en parallèle un *générateur de code* pour la machine Ada, greffé en «dorsal» (*back-end* en anglais) sur le compilateur Ada/ED initial. Quelques uns des problèmes abordés dans cette thèse sont d'ailleurs autant des problèmes de génération de code que strictement des problèmes de structure de la machine Ada : valeurs initiales par défaut (cf. chap. 5), évaluation des agrégats (cf. chap.6). La génération de code est précédée par une passe d'*expansion* du langage intermédiaire d'Ada/ED, et enfin il existe un *relieur* (*binder* en anglais) pour l'édition des liens entre unités de compilations séparées (cf. chap. 11).

Le modèle en SETL de la machine Ada a été ensuite réécrit en C par Robert Dewar avec pour objectif de pouvoir simuler la machine sur un micro-ordinateur du genre IBM PC. L'ensemble a passé avec succès la suite de validation A.C.V.C. au printemps 1986, et la version pour «PC» devrait être prête à l'automne 86.

7. Autres réalisations similaires

Nous avons essayé chaque fois que cela était possible de confronter nos travaux avec d'autres, similaires, afin d'en avoir une vue critique. Mais il y a peu d'informations précises disponibles sur les environnements d'exécution Ada, les organisations de mémoire, les mécanismes spécifiques mis en œuvre. Il y a en effet des intérêts commerciaux énormes mis en jeu, et on ne peut reprocher aux sociétés de protéger leur secrets de fabrication. On peut classer les implémentations d'Ada en deux grandes catégories : les compilateurs «expérimentaux», en général des implémentations partielles développées sur des deniers publics, souvent dans un cadre universitaire, et les compilateurs développés par des sociétés privées, où sauf de rares exceptions, rien de très précis n'est publié.

◇ Compilateurs expérimentaux : outre Ada/ED de New York University, citons le Delft Ada Subset [Katwijk 84], l'Ada Breadboard Compiler, des «Bell Labs» [Wetherell 82, Rubine 82, Nowitz 82], le Charette Ada Compiler [Rosenberg 80, Hisgen 80, Sherman 80] et ses descendants : le Spice Ada Compiler et l'Ada+ Compiler [Barbacci 85] à Carnegie-Mellon University. Il y a aussi le compilateur de l'université de York, celui de l'université d'Irvine, dans une certaine mesure celui de l'université de Karlsruhe, à ses débuts [Goos 83], et AdaM à Stanford, qui ne nous ont guère servis.

◇ Compilateurs «commerciaux» : Il y a des informations assez détaillés sur le Dansk Datamatik Compiler [Ibsen 83, Ibsen 84], celui de Oy Softplan pour la machine Nokia MPS [Lahtinen 82] et quelques informations sur l'ALS [Kamrad 83]. Très peu d'informations sont disponibles sur les environnements d'exécution des «ténors» du domaine : Verdix, Alsys, Telesoft, Rolm+Data General, Rational. Ce dernier dispose apparemment d'une véritable Machine Ada, appelée R1000.

Toutefois, nous avons des informations fragmentaires sur toutes ces implémentations par des travaux de conseil que nous avons eu l'occasion de faire, et surtout par des discussions avec leurs auteurs lors de nombreuses réunions de SigADA (ex-AdaTEC) et d'Ada-Europe.

Chapitre 2.
La Mémoire

Ce chapitre décrit plus en détails l'organisation et la gestion de la mémoire de la Machine Ada virtuelle.

1. Position du problème

La mise au point d'un modèle de mémoire pour un langage moderne de haut niveau pose certains problèmes dès que l'on a, à côté du mécanisme d'appel récursif de sous-programmes, d'autres primitives de contrôle du flot d'exécution : coroutines (Simula-67), back-tracking ou «retour arrière» (SETL), ou processus concurrents (Mesa, Concurrent Pascal et Ada). En effet, la mécanique du «dernier arrivé, premier parti» imposée par la sémantique des appels et retours de sous-programmes se prête admirablement à une réalisation à l'aide d'une *pile*, tant que l'on n'a pas en plus l'une des autres formes de contrôle.

Les autres formes de contrôle : tâches concurrentes, retour arrière et coroutines, impliquent une certaine «rétention» des informations liées aux activations, et cette rétention a une durée de vie variable : elle ne suit pas la règle «dernier arrivé, premier parti». La pile, à moins de faire d'abondantes copies et recopies, n'est pas une bonne solution. La plupart des réalisations combinent par diverses méthodes une ou plusieurs piles avec un *tas* où l'allocation d'espace se fait sur le principe du «premier bloc libre trouvé», avec ou sans compactage.

Plusieurs auteurs se sont penchés sur les diverses manières de combiner *Pile(s)* et *Tas* pour les langages de haut-niveau en général [Kearns 83] ou pour Ada en particulier [Gupta 85].

Pour Ada en effet il y a des problèmes spécifiques de gestion de mémoire, qui découlent des points suivants du langage :

◇ l'existence de tâches qui ont des contextes d'exécution distincts, mais qui d'une part accèdent à des entités appartenant à des tâches «englobantes» et d'autre part échangent des informations entre elles par des paramètres, à l'instar des appels de sous-programmes [LRM 9] ;

◇ la durée de vie des objets créés par l'exécution d'un allocateur est en règle générale la durée de vie du bloc dans lequel le type accès correspondant est élaboré [LRM 4.8(7, 11)] ;

◇ et, plus particulièrement, un objet tâche qui est un objet ou un sous-composant d'un objet créé par l'exécution d'un allocateur dépend du bloc maître qui a élaboré le type accès correspondant [LRM 9.4] ;

◇ l'existence de traite-exception (*exception handlers*) et leur lien avec les cadres (*frames*) [LRM 11.2] ;

◇ enfin, certains objets ont des tailles et des contraintes qui ne sont pas connues statiquement dès la compilation [LRM 3.3, 3.2.1].

En outre, nous avons rajouté aux problèmes énumérées ci-dessus les contraintes suivantes :

◇ possibilité de faire de la génération de code unité de compilation par unité de compilation, au moment de leur compilation sans avoir à écrire un éditeur de lien et un chargeur complexes ;

◇ et, puisqu'à terme nous désirons obtenir des implémentations sur de petites machines, il nous faut un mécanisme où la mémoire soit segmentée, et où ces segments soient relogeables n'importe où en mémoire.

2. Unité de mémoire et segments de mémoire

La mémoire de la Machine Ada est supposée adressable au niveau de l'*unité de mémoire* [LRM 13.7(3)]. A l'instar de beaucoup de machines modernes, nous avons choisi un octet de huit éléments binaires comme unité de mémoire.

Un *segment* est une zone contiguë d'unités de mémoire ; la taille maximale d'un segment est de 2^{16} unités de mémoire.

La mémoire de la machine Ada est partagée logiquement en quatre sortes de régions :

- Les DATA_SEGMENTS, qui sont un ensemble de segments de données, un par unité de compilation, contenant toutes les entités globales ou statiques de cette unité de compilation (patrons de type et constantes statiques, variables globales).
- Le HEAP (ou tas), qui est un tableau d'unités de mémoire unique qui contient tous les objets Ada dynamiques : variables locales, constantes non statiques, patrons de types non statiques, etc.
- Les STACKS (ou piles), un ensemble de segments de pile, un par tâche Ada, qui contiennent toutes les informations appartenant en propre à la tâche correspondante.
- Les CODE_SEGMENTS, un ensemble de segments contenant du code exécutable. Il y en a à peu près un par unité de programme : sous-programme, tâche, paquetage unité de bibliothèque,... plus quelques uns pour des sous-programmes internes générés par le compilateur (tel que les procédures d'initialisation de certains types d'objet, cf. §5-2) ; mais il n'y en n'a pas pour les paquetages qui ne sont pas des unités de bibliothèque : le code correspondant est dans le segment de code de l'unité de programme englobante.

Pratiquement, les segments sont logés n'importe où dans la mémoire, et ce sont les *topographies* de la mémoire qui conservent la carte des quatre régions. Il n'y a que trois topographies, car en effet le tas est géré comme un segment de données de numéro fixé, connu.

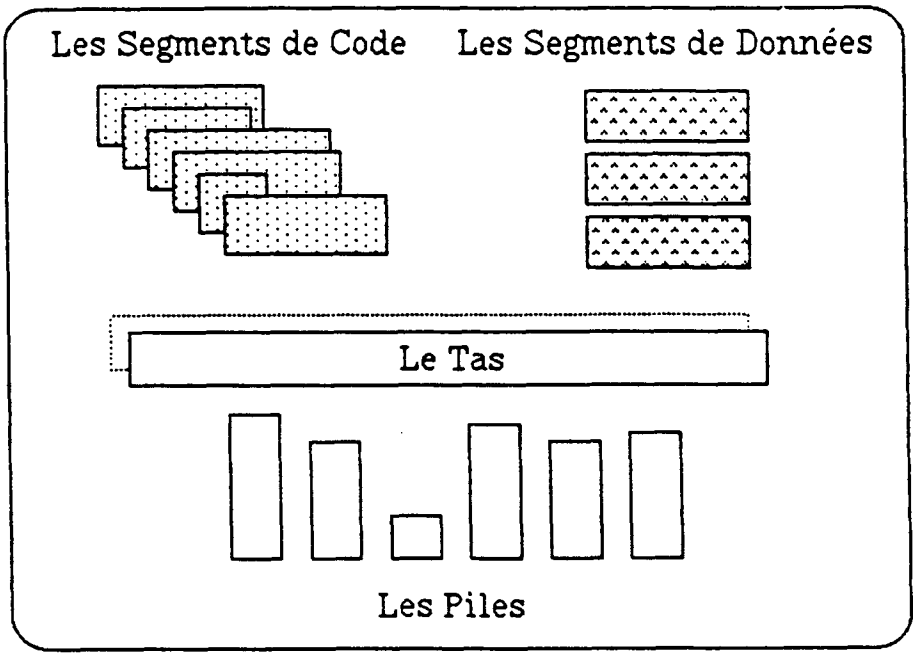


Figure 5 : les divers segments de mémoire

3. Le tas

Toutes les entités non statiques d'Ada sont logées dans le tas ; ceci inclut les objets locaux des sous-programmes, les objets globaux dont le type n'est pas connu statiquement, les corps de sous-programmes ou de tâches dont l'ensemble-relais n'est pas statique (cf. §7-3) et les patrons de type non statiques (cf. §4-2). On y trouve bien entendu aussi les objets créés par l'exécution d'un allocateur (primitive "new"). Normalement, le tas n'est constitué que d'un seul segment ; il peut être formé de plusieurs segments dans les cas suivants :

a) si pour un type accès donné il a été spécifié une taille de collection par une clause de longueur [LRM 13.2(7)], un segment de tas distinct est créé pour cette collection, dont la taille est celle spécifiée par la clause de longueur, et tous les objets désignés par des valeurs de ce type accès seront logés dans ce tas secondaire [LRM 4.8].

b) Si le tas principal déborde, c'est-à-dire s'il excède l'espace adressable, alors un tas secondaire est créé.

Ces deux cas sont prévus principalement pour permettre d'avoir plus de 2^{16} octets de données dans le tas lorsque la machine effective sous-jacente

est un micro-ordinateur à base d'Intel 8086, par exemple, si nous prenons des adresses de 16 bits au sein de chaque segment.

L'allocation et la libération de mémoire dans le tas seront examinées plus loin (cf. §2-7).

Le tas est «vu» par toutes les tâches et par tous les blocs d'une tâche. C'est ainsi que les objets sont partagés entre tâches et entre blocs : tous les objets et types, même si leur taille est connue dès la compilation, même les valeurs scalaires, tous sont logés dans le tas, et les piles des environnements de chaque tâche ne contiennent que des pointeurs vers le tas. Pour simplifier l'adressage, ceci est également vrai pour des objets qui sont purement locaux à un bloc, ainsi que pour les paramètres des sous-programmes et des entrées (cf. §§7-7 et 9-3).

4. Les piles

Les piles sont organisées en une sorte de «rateau», une pile par tâche. Chaque pile est un segment. Elle est constitué du *descripteur de tâche* ou *bloc de contrôle de tâche* (cf. [Rosen 86]), et d'une suite d' *environnements*.

A l'exécution, un *environnement d'appel* est créé au sommet de la pile pour le corps de la tâche, puis pour chaque appel de sous-programme, pour chaque instruction "accept". Les environnements d'appel se recouvrent légèrement pour permettre le passage de paramètres (cf. §7-7), comme le montre la figure 2-2. Un environnement d'appel contient un ou plusieurs *environnements de bloc*, un pour chaque bloc d'Ada : sous-programme, instruction "accept" et instruction de bloc. Il y a en outre des environnements de bloc générées par le compilateur qui ne correspondent à aucun bloc Ada mais sont nécessaires pour pouvoir générer un code plus «carré» ; voir par exemple au §8-4. Tous les environnements d'appel ont la même structure, constituée de 4 zones :

- des *données de liaison*, permettant de revenir à l'environnement de retour (l'appelant),
- la *portée de l'environnement*, c'est-à-dire des pointeurs vers l'ensemble des objets référencés par cet environnement,

- la pile d'évaluation, avec les environnements de bloc, un au minimum,
- le cas échéant, le *contexte de la tâche.* (voir figure 2-2)

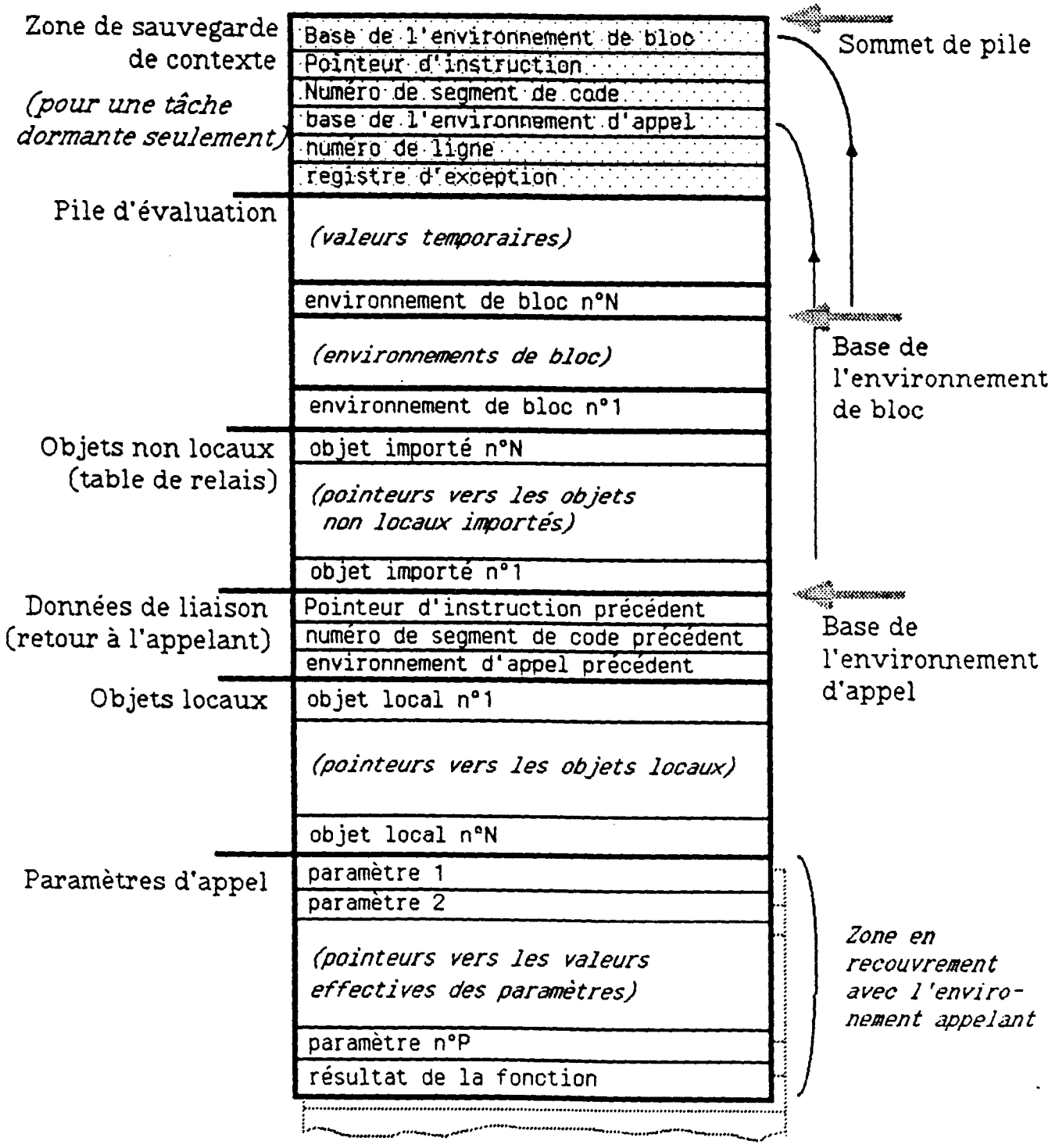


Figure 6 : Structure d'un environnement d'appel

4.1 Les données de liaison

Elles consistent en :

- l'*adresse de retour* formée du *numéro de segment de code* et d'un déplacement dans ce segment de code, le *pointeur d'intruction*
- la *base de l'environnement d'appel de retour*.

Ces informations sont sans signification pour le premier environnement d'appel d'une tâche.

4.2 La portée de l'environnement

Cette zone contient :

- un emplacement pour la *valeur retournée* par une fonction

et une liste de pointeurs vers le tas :

- des pointeurs vers les paramètres dans le tas ; cette partie est en recouvrement avec l'environnement appelant, car son évaluation est faite dans le prélude d'appel.
- des pointeurs vers les objets locaux (y compris vers les sous-programmes et les patrons de type), qui nous le rappelons sont situés dans le tas.
- des pointeurs vers les entités appartenant à des environnements extérieurs. Ceci est la table de relais du sous-programme (ou corps de tâche) courant, qui a été calculée initialement lors de l'élaboration du sous-programme, stockée dans l'objet sous-programme et recopié ici lors de l'appel pour simplifier l'adressage : les objets appartenant à des environnements extérieurs peuvent être ainsi en effet référencés comme des objets locaux.

Ce mécanisme particulier d'accès aux environnements extérieurs sera examiné au §7-2.

4.3 La pile d'évaluation

Elle contient les environnements de bloc décrits plus loin, ainsi que diverses variables temporaires utilisées au cours de l'évaluation des expressions Ada, des pointeurs vers le tas pendant des calculs d'adresses,

et les paramètres effectifs lors de l'appel d'un sous-programme ou d'une entrée.

4.4 La zone de sauvegarde du contexte

Lorsqu'une tâche n'est pas active, son contexte dynamique, c'est-à-dire l'ensemble de ses registres, est empilé au sommet de sa pile. Ceci consiste en :

- le *pointeur d'instruction* courant : une adresse de code complète : numéro de segment de code et déplacement.
- les *base d'environnement d'appel* et *base d'environnement de bloc* courants,
- le *registre d'exception* courant,
- le *numéro de ligne source* courant.

4.5 L'environnement de bloc

Un environnement de bloc contient des informations très importantes attachées à un cadre (*frame*), au sens Ada du terme [LRM 11.2(3)]:

- la *base de l'environnement de bloc précédent* utilisé lorsqu'on quitte un environnement de bloc;
- le *lien de données*: c'est la tête de la chaine de tous les «morceaux» de données dans le tas qui sont attachés à ce bloc, et qui ne pourront être libérés que lorsqu'on quittera ce bloc (cf §2-7).
- Les *tâches déclarées* : c'est la tête d'une liste chainée de tâches représentant l'ensemble des tâches qui ont été créées dans le bloc courant mais qui n'ont pas encore été activées, et qu'il faudra activer lorsque l'exécution atteindra la fin de la partie déclarative du bloc courant. Pour le mécanisme d'activation des tâches, on se reportera au §9-2 et à [Rosen 86].
- Les *sous-tâches* : c'est la tête de la liste chainée de tâches qui représente l'ensemble des tâches qui dépendent de l'environnement de bloc courant, et dont il faudra attendre la terminaison pour pouvoir quitter ce cadre et dépiler cet environnement de bloc (cf. §9-4 et [Rosen 86]).

- Le *vecteur d'exception,* qui contient l'adresse du traite-exception courant. Il désigne par défaut le traite-exception «piège» (cf. §8-3).

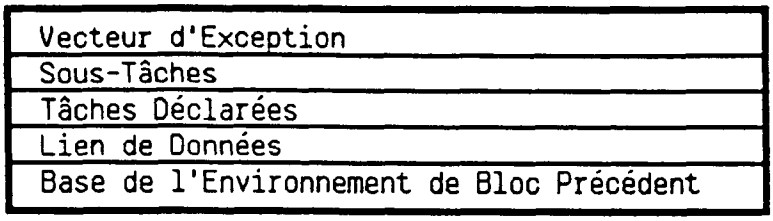


Figure 7 : Structure d'un environnement de bloc

On remarquera que les *tâches déclarées* et les *sous-tâches* sont deux ensembles distincts pour plusieurs raisons : pendant l'exécution d'un allocateur, les tâches déclarées sont sauvegardées, les tâches nouvellement créées (par l'exécution de l'allocateur) sont chaînées sur une liste initialement vide. A la fin de l'exécution de l'allocateur, leur activation a lieu, mais ces tâches sont chaînées sur la chaîne de tâches associée au bloc qui a élaboré le type accès utilisé dans l'allocateur (donc souvent dans les *sous-tâches* d'un autre environnement de bloc), puis enfin on restore les tâches déclarées du bloc courant [LRM 9.3,9.4]. En ce qui concerne les paquetages, les tâches déclarées dans la spécification de paquetage et dans le corps de paquetage doivent être réunies ensemble dans une unique liste de tâches pour leur activation [LRM 9.3(2)]. Les tâches déclarées doivent par conséquent être conservées entre l'élaboration de la spécification et la fin de l'élaboration de la partie déclarative du corps du paquetage dans une variable temporaire associée au paquetage. C'est d'ailleurs, notons-le, une des seules manifestations des paquetages au niveau de la sémantique dynamique d'Ada.

5. Les segments de code

Le code exécutable est organisé lui-aussi sous forme d'un «rateau» de segments de code. Une adresse de code complète est un couple (numéro de segment de code, déplacement); cette adresse de code complète est utilisée pour les appels de sous-programme ou d'entrée, et les retours. Pour les branchement et les boucles, ainsi que pour spécifier l'emplacement du

traite-exception actif, on utilise seulement le déplacement. Il y a à peu près un segment de code par unité de programme : sous-programme et tâche. Pour les paquetages, il faut distinguer les paquetages qui sont des unités de compilation, pour lesquels il y a un segment de code pour la spécification et un segment de code pour le corps, et les paquetages qui sont emboîtés, pour lesquels il n'y a pas lieu de générer un segment de code distinct de celui de l'unité englobante.

L'élaboration d'un sous-programme ou d'un corps de tâche consiste à calculer sa propre *table de relais* qui lui permet d'accéder à des objets non locaux (hormis les objets de l'environnement principal). Cette table est associée au segment de code et recopiée dans l'environnement d'appel chaque fois qu'on entre dans ce segment de code (cf. §7-2).

6. Espaces d'adresse

Tous les segments peuvent contenir au plus 2^{16} unités de mémoire, ce qui donne à un déplacement une largeur de 16 éléments binaires. Du point de vue des mécanismes d'adressage, il n'y a que trois sortes de segments, le tas étant un segment de données qui n'existe que lors de l'exécution : les segments de données, les segments de code, et les segments de pile. Ces trois sortes de segment sont manipulés par des instructions distinctes et il n'y a en pratique jamais besoin de pouvoir les distinguer. Nous avons ainsi trois espaces d'adresses distincts.

Le nombre de segments de données et de tas est limité à 256. Ce nombre assez bas est important, car il permet à un numéro de segment (que nous appellerons *base*) de tenir sur un octet ; et donc une adresse de donnée complète tient sur trois octet. Si l'on suppose qu'il n'y a qu'un seul segment de tas, et en réservant quelques numéros pour les paquetages prédéfinis tels que STANDARD, SYSTEM, CALENDAR, TEXT_IO..., il peut y avoir quelques 250 unités de compilation. Vu les objectifs modestes de l'implémentation, ceci a été jugé suffisant.

Mais pour les segments de code, 256 est par trop limitatif, et, codant un numéro de segment de code sur 2 octets, nous avons une limite supérieure de 2^{16} segments de code.

Pour les segments de pile, 256 serait aussi par trop limitatif, car nous pensons que l'usager typique d'un système «éducatif» voudra faire des expériences avec des programmes comprenant un grand nombre de tâches, les tâches étant une des originalités d'Ada. Le nombre maximal de segments de pile est donc également 2^{16}.

L'adressage au sein d'un segment est toujours relatif au début du segment (sa base) et par conséquent tous les segments peuvent être logés n'importe où en mémoire et relogés ailleurs, le système devant alors tenir à jour les topographies donnant pour chaque numéro de segment sa base.

La taille des segments de code et des segments de données est connue dès la compilation. Mais ce n'est pas le cas du tas, ni des piles. On leur allouera des espace mémoire arbitraires, prédéfinis, et on les relogera ailleurs, dans un espace physique plus grand s'ils débordent la place qui leur a été intitialement allouée.

Le modèle en SETL prend quelques libertés avec cette définition de la mémoire de la machine Ada, essentiellement pour des raisons de niveau sémantique du code de l'interprète et d'efficacité. Il n'y a pas de notion d'octet en SETL, mais des emplacements dans des tuples (ou séquences) ; il n'y a pas de limite à 2^{16} éléments pour un segment ; les segments ne sont pas en «mémoire», mais sont des tuples indépendants. Tous ceci n'altère en rien la validité du modèle en SETL de la machine Ada, mais élude quelques problèmes assez critique du point de vue des performances qu'il faudra résoudre au moment de la réécriture en C.

7. Allocation de mémoire

L'allocation dans le tas est fait sur la base de *morceaux* de longueur variable. Un morceau du tas est constitué de données utiles, plus des informations utilisées par le gestionnaire de mémoire et le ramasse-miettes :

- l'*état* du morceau : libre ou occupé ;
- la *taille* du morceau : un entier donnant la longueur hors-tout du morceau (informations supplémentaires comprises) ;
- le *lien de données* : un pointeur vers le morceau de mémoire suivant appartenant au même environnement de bloc (cf. ci-dessus §2-4.5).

En pratique, état du morceau et taille peuvent être codés ensemble.

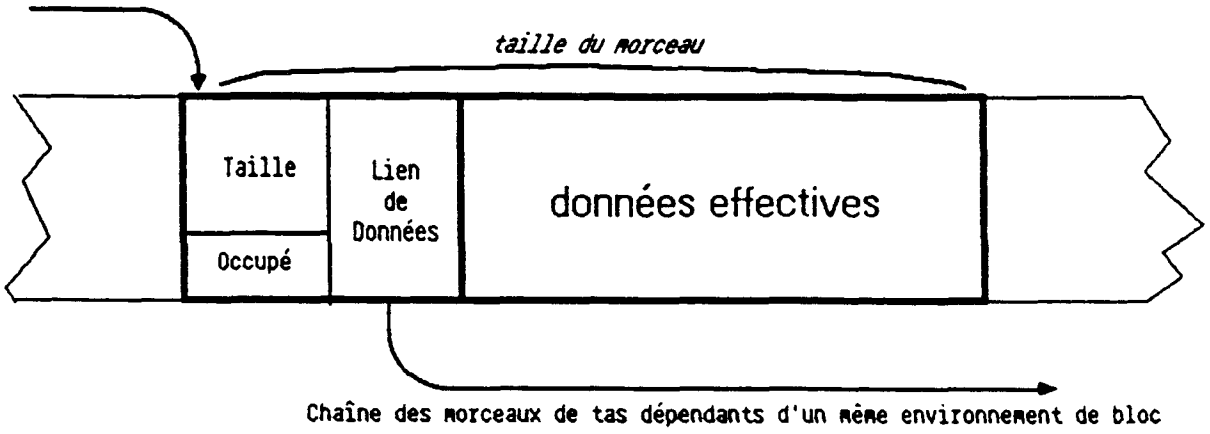


Figure 8 : Un morceau du tas

Les objets et patrons de type Ada créés durant la durée de vie d'un bloc sont libérés lorqu'on quitte ce bloc (cf. instruction *leave_block*); pour les objets qui ont été alloués par l'allocateur Ada "new", ils ne seront libérés que lorsqu'on quittera le bloc où le type accès a été élaboré [LRM 4.8(7)]. Ceci nous empêche d'utiliser le tas strictement comme une pile et tend à fragmenter le tas. La politique d'allocation adoptée est donc la suivante : on alloue un morceau au bout du tas jusqu'à ce que le tas atteigne une certaine taille maximale, et à partir de là on applique une politique de *first-fit* ou «premier trou satisfaisant». Lors de la libération d'un morceau, on vérifie s'il est au bout du tas, auquel cas le pointeur de bout du tas est ramené en arrière, plutôt que de simplement marquer le morceau comme libre. Ceci a pour effet de reculer le plus possible le moment où l'on commencera à appliquer la politique du first-fit ; en d'autres termes, autant que possible, on essaie de gérer le tas comme une pile.

Si l'on choisit de ne pas faire de ramasse-miettes sur les collections, il n'est pas nécessaire de positionner le lien de données ; dans ce cas il est possible d'avoir une instanciation de UNCHECKED_DEALLOCATION [LRM 13.10.1] pour libérer explicitement un morceau, qui consiste simplement à marquer l'*état* du morceau comme libre et de réajuster le chaînage.

8. Modes d'adressage

Ce modèle de mémoire, plus le mécanisme d'accès aux variables non locales que nous verrons au chapitre 7 nous permettent d'avoir un nombre extrêmement réduit de modes d'adressages.

- Les instructions qui font référence à la mémoire (pour lire ou écrire des données, pour accéder à un patron de type ou pour faire un appel de sous-programme) ont au plus deux modes d'adressage.

Si on appelle RELOC la topographie des segments de données, on a :

◇ le mode "donnée locale" : l'instruction contient un déplacement *d* à partir duquel on calcule :

base = STACK(SFP+d)
déplacement = PILE(SFP+d+1)
loc = RELOC(base)+déplacement

◇ le mode "donnée globale" : l'instruction contient *base* et *déplacement* :

loc = RELOC(base)+déplacement

- Quelques instructions ont un mode "immédiat", où l'opérande est contenu dans l'instruction.

- Les instructions qui font référence à des emplacements dans le code (branchements, boucles, activation de traite-exception) ont un unique mode d'adressage :

◇ le mode "code local" : le déplacement donné dans l'instruction est le déplacement absolu dans le segment désigné par le registre de numéro de segment de code.

- Les instructions utilisées pour l'élaboration d'objets locaux font référence à l'environnement d'appel courant :

◇ le mode "pile local" : l'instruction donne un déplacement au sein de l'environnement d'appel courant :

loc = SFP+d (dans STACK=STACK_SEGMENTS(TP))

Toutes les autres références faites par les instructions sont implicitement au sommet de la pile de la tâche courante.

9. Discussion

L'organisation du tas et des piles en Ada est un problème délicat qui sort du cadre des traités classiques de compilation : [Gries 71, Aho 77, Cunin 80] par exemple. Gupta et Soffa font une étude des diverses organisations possibles pour Ada, et montrent qu'il n'y a en fait que peu de salut possible hors de l'organisation des piles «en cactus», à cause des dépendances et du partage d'objets entre tâches [Gupta 85]. L'organisation en cactus semble d'ailleurs utilisée par toutes les implémentations Ada. Toutefois elle s'accompagne en général du mécanisme traditionel de chainage dynamique pour l'accès aux objets non-locaux, chaînage compliqué par le fait que les objets non locaux peuvent appartenir à d'autres tâches, donc appartennir à des environnement de piles plus proches du «tronc principal» du cactus.

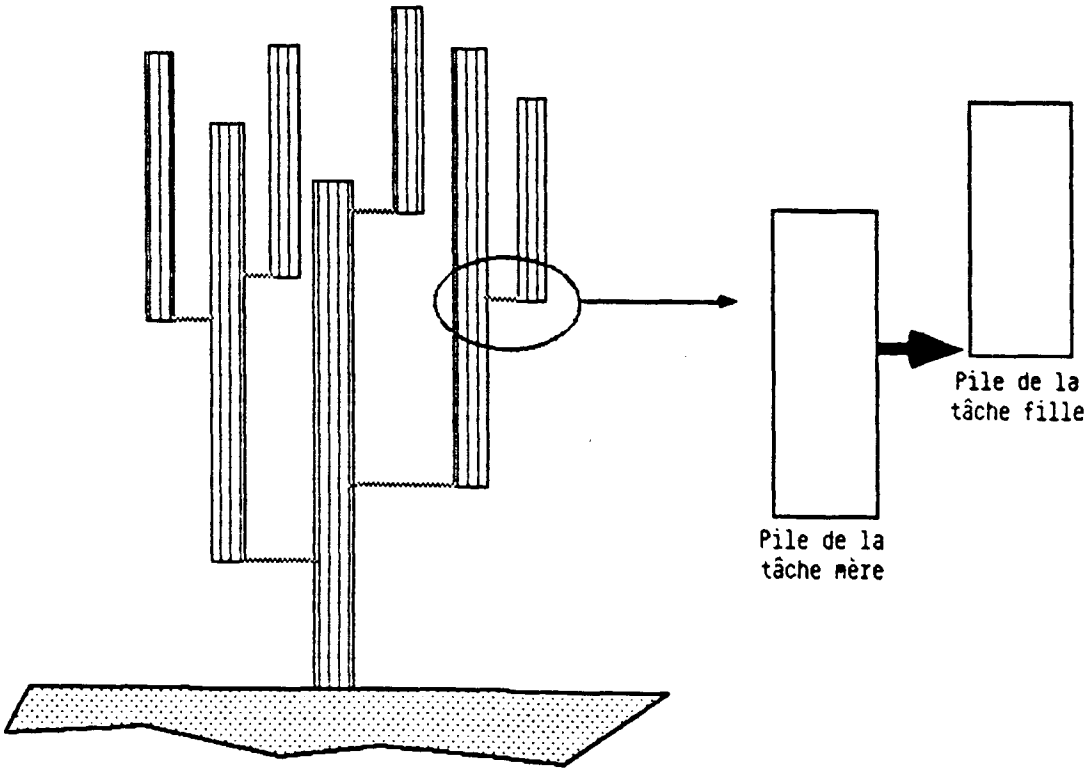


Figure 9 : Le «cactus-stack»

La structure de chaque pile est en outre compliquée par le problème de durée de vie des objets (notamment des tâches) associés aux blocs, et

l'existence de traite-exception, ce qui nous a obligé à compliquer les mécanismes traditionellement employés dans les langages à structure de bloc. En ce qui concerne la durée de vie des objets, les implémentations décrites dans la littérature, comme [Wetherell 82, Ibsen 84, Barbacci 85], ne donnent pas le mécanisme utilisé.

Chapitre 3.
Représentation des objets et valeurs

Ce chapitre décrit la manière dont sont représentés les objets et valeurs Ada dans la mémoire de la machine Ada.

1. Position du problème

La représentation des objets que nous proposons ici est un compromis qui cherche à satisfaire plusieurs contraintes parfois contradictoires.

- ◇ l'élaboration des objets, surtout les objets structurés à contraintes non statiques, peut être relativement complexe et elle inclut souvent l'élaboration d'un type ou sous-type, et l'évaluation de valeurs initiales par défaut ou explicites ;
- ◇ les objets peuvent être partagés entre plusieurs tâches, ce pourquoi nous avons décidé de les implanter tous dans le tas, et de ne conserver que des pointeurs dans les environnements d'exécution ;
- ◇ une difficulté majeure réside dans les valeurs d'un type tableau non contraint qui peuvent apparaître dans plusieurs situations : comme valeur d'une constante non statique, comme valeur retournée par une fonction, ou comme valeur désignée par une valeur accès dans un allocateur ;
- ◇ toujours pour les tableaux, la présence de *tranches*, qui peuvent être des noms (et donc figurer à gauche d'une affectation), élimine quelques solutions évidentes ;

◇ enfin la plus grande difficulté est posée par l'existence d'articles avec discriminants contenant des composants dont les contraintes, et notamment les bornes, dépendent des discriminants.

2. Les types simples de base

Une adresse, ou *pointeur* est constituée d'un couple (numéro de segment, déplacement dans le segment). A l'exécution, la pile contient des adresses d'objets, mais les valeurs temporaires sont représentées soit par leur valeur effectives pour les types simples, soit par des pointeurs vers les valeurs effectives pour les types composites. Pour un article, un seul pointeur suffit, mais pour les objets ou valeurs d'un type tableau, une référence complète est formée d'un doublet : l'adresse du tableau et l'adresse du patron de type de ce tableau.

Il y a six types internes simples dans la Machine Ada :

- *Byte* : utilisé pour les booléens, les entiers courts (Short_Integer), les numéros de champ d'article et les numéro de segment de donnée ;
- *Word* : utilisé pour les entiers (Integer), les tâches et pour un déplacement dans un segment de donnée ;
- *Addr* : utilisé pour les adresse absolues, formée d'un numéro de segment plus un déplacement ;
- *Long* : utilisé pour les entiers longs, les nombres réels à point-flottant (Float) et certains réels à point-fixe ;
- *Dble* : pour une double adresse absolue, comme une référence complète à un tableau ;
- *Xlng* : pour les nombres réels à point-flottant longs (Long_Float) et les nombres réels à point-fixe requérant une mantisse plus grande.

Remarquez que dans le prototype en SETL de la machine Ada, les mnémoniques ci-dessus (Byte, Word, Addr, Long, Dble, Xlng) ne correspondent pas de très près au modèle sous-jacent, car nous y utilisons une position de tuple par valeur, et non pas des octets. De plus SETL ne supportant pas l'arithmétique «double précision», le type Xlng ne peut être utilisé que pour les réels à point-fixe. Le tableau suivant donne

l'implémentation de ces types internes de base dans le prototype SETL et dans la version en C, plus proche de la machine Ada :

Type	Machine Ada		Prototype SETL
	en octets	en e.b.	en position de tuple
Byte	1	8	1
Word	2	16	1
Addr	3	24	2
Long	4	32	1
Dble	6	48	4
Xlng	8	64	2

3. Représentation des booléens et autres types énumérés

Les valeurs booléennes occupent un octet, mais seul le bit de poids faible est significatif et vaut 1 pour True, 0 pour False (satisfaisant ainsi la relation True > False). Les opérateurs de relation rendent des résultat booléens au sommet de la pile d'évaluation, qui sont testés par les instruction de branchement conditionnelles (cf.§10-7).

4. Représentation des nombres entiers et réels à point-flottant

La Machine Ada supporte trois types entiers prédéfinis : Short_Integer (de -128 à 127), Integer (de -32536 à 32537) et Long_Integer (de -2^{31} à 2^{31-1}).

Ils sont représentés en complément à 2 respectivement sur un Byte, un Word et un Long.

5. Représentation des nombres réels à point-fixe

La définition d'un type point-fixe comporte les éléments suivants :

- le delta,
- l'intervalle,
- le pas (small).

Ce dernier est introduit éventuellement par une clause de longueur [LRM 13.2(11)]. Nous n'autorisons dans notre implémentation que des pas qui soient une puissance de 2 (cas par défaut en Ada), soit une puissance de 5,

soit un produit de puissance de 2 et de puissance de 5, ce qui inclut les puissances de 10, dont nous pensons qu'elles seront très souhaitées par les programmeurs.

A partir de ces paramètres on choisit une représentation : mantisse et signe, plus facteur d'échelle. La mantisse et le signe sont logés, en fonction de l'intervalle et de la précision relative demandée, sur l'un des type interne de base : Word, Long, Dble ou Xlng (ou en d'autres termes, sur 2, 4, 6 ou 8 octets) ; le facteur d'échelle, qui est une caractéristique du type est formé de deux entiers, dans l'intervalle -128 à +127, indiquant les puissances de 2 et de 5 dont est formé le pas.

La sémantique de la division et de la multiplication des réels point-fixe requiert des résultats temporaires (sur la pile d'évaluation) plus grands que Xlng, et il existe une instruction *normalize* dans la machine Ada pour réduire une telle valeur à l'un des types définis [LRM 4.5.5]. Hormis ce cas, les valeurs réelles à point-fixe sont traitées comme des valeurs entières, et on utilisera les mêmes instructions que pour les entiers (cf. §10). Voir [Rosen 86] pour une discussion approfondie de l'arithmétique réelle en point-fixe.

6. Représentation des tableaux

Les objets ou composants de type tableau sont contraints ; par conséquent l'adresse du tableau effectif suffit à les représenter. Les informations quant à leur structure peuvent être déduites du patron de type associé. Mais il y a des situations dans lesquelles cette information structurelle sur le tableau n'est pas connue statiquement : les fonctions qui retournent un tableau non contraint, les constantes tableau non contraint, les valeurs d'accès qui désignent un tableau non contraint, et quelques opérations comme la concaténation. Dans ces cas, l'information structurelle doit être d'une façon ou d'une autre attachée, reliée, à la valeur du tableau.

Une solution évidente, utilisée pour maints autres langages (comme Algol 68, PL/1) est de préfixer la valeur tableau par une sorte de descripteur contenant les bornes, le *dope vector*, etc. Mais ceci n'est guère pratique en Ada, à cause des tranches [LRM 4.2.1] qui peuvent être utilisées comme *noms* et par conséquent figurer à gauche d'un opérateur d'affectation, ou comme

paramètre de mode "out". Pour éviter de faire des tranches un cas particulier de tableau, nous avons choisi une solution un peu plus complexe.

Les valeurs et objets tableau sont représentés dans la pile par un *descripteur de tableau*, formé de deux pointeurs : l'adresse du tableau et l'adresse de son patron de type. Pour les constantes non statiques et les valeurs d'accès, une fois que la valeur tableau a été évaluée et placée sur la pile, on crée un descripteur de tableau dans le tas, et le tableau est représenté par un pointeur sur ce descripteur. Dans ce cas, l'accès au tableau requiert une indirection supplémentaire.

Dans le cas d'une fonction qui retourne un tableau non contraint, on laisse sur la pile un tel descripteur de tableau. D'autres opérations sur les tableaux, comme la concaténation [LRM 4.5.3] ou l'évaluation de divers attributs comme First, Last, Range, attendent un descripteur de tableau au sommet de la pile.

Un mécanisme similaire de "doublet" est proposé par le Delft Ada Subset Compiler [Katwijk 84], mais généralisé à *tous* les objets et valeurs !

7. Représentation des articles

Les articles sans discriminants ne posent absolument aucun problème : les divers composants sont disposés en mémoire côte à côte. Dans la plupart des cas le déplacement d'un composant au sein de l'article est connu statiquement, et dans le pire des cas, il est connu lors de l'élaboration du type; on le retrouvera donc dans le patron de type correspondant.

Les articles avec discriminants sont un peu plus délicats à manipuler ; les discriminants peuvent être utilisés dans trois cas de figure :

- dans une expression initiale par défaut pour un composant (cf. §5-2),
- comme sélecteur de partie variante : nous avons choisi d'avoir les variantes mutuellement exclusives en recouvrement, et pour chaque sélection de composant, il faut vérifier avec l'aide du patron de type qu'il est présent (cf. §4-3.12).
- pour spécifier des bornes d'indices ou des discriminants de composants, ce qui nous conduits à des articles de taille variable (cf. §4-3.9,3.14).

Un article de taille variable est un article avec discriminant avec ou sans contraintes contenant un ou des sous-composants dont les bornes dépendent directement ou indirectement des discriminants.

Exemples:

```
type R(D: INTEGER) is record  -- cas simple
   S: STRING(1..D);
end record;

type R1(D: INTEGER) is record  -- dépendance au premier degré
   SA: STRING(1..D);
   SB: STRING(1..D);
end record;

type R2(D: INTEGER) is record  -- dépendance au second degré
   IR: R1(D);
end record;

type UR1(D: INTEGER:= EXPRESSION) is record  -- non contraint
   SA: STRING(1..D);
   SB: STRING(1..D);
end record;
```

Quel est exactement le problème ? Si le programmeur déclare ensuite

```
subtype SR1 is R1(10);
```

alors la disposition interne et la taille des articles de type SR1 peut être déterminée statiquement à la compilation, et la sélection de SB dans un objet de type SR1 est directe. mais s'il déclare un sous-type non statique, comme

```
subtype SFR is R1(Un_Appel_De_Fonction);
```

alors la représentation exacte des objets de type SFR ne peut être déterminée avant l'élaboration du sous-type. Lors de cette élaboration toutes les informations sur les tailles et bornes des composants peuvent être calculées et stockées dans des patrons de type. Mais les choses se compliquent encore avec des constructions telles que :

```
type ARC is access R1;
PAR : array(1..1000) of ARC;

for I in PAR'RANGE loop
   PAR(I) := new R1(I);
end loop;
```

Faut-il créer un millier de sous-types anonymes, et comment les relier à chaque objet ? Comme pour les tableaux non contraints ci-dessus ? Remarquons que les objets et valeurs de type article contiennent les discriminants comme composants, et que toute l'information structurelle

peut être reconstruite à partir des discriminants, ceci peut ne pas être que cher mais aussi terriblement redondant.

Une première solution assez brutale consiste à allouer la taille maximale possible pour chaque composant, de façon à être sûr que tous les sous-types du type peuvent avoir la même représentation, et à attacher cette représentation au patron de type. Ceci n'est guère réaliste, car le programmeur spécifiera souvent (comme c'est le cas ci-dessus) des discriminants tels que INTEGER, qui comporte un nombre très grand de valeurs.

Une autre solution est celle dite des «pointeurs cachés», utilisées dans plusieurs réalisations (Systeam, Telesoft). Au lieu de stocker les composants côte à côte linéairement dans l'article, ceux dont la taille n'est pas connue sont remplacés par un pointeur vers ce composant qui est stocké soit au bout de l'article, soit même ailleurs dans le tas. Ceci correspond à la transformation suivante :

```
type R1(D: INTEGER) is record
   SA, SB: STRING(1..D);
end record;
Q, P: R1(10);
...
if Q.SA = Q.SB then ...
```

est transformé par le compilateur en :

```
type ACC_STRING is access STRING;
type R1(D: INTEGER) is record
   SA, SB: ACC_STRING := new STRING(1..D);
end record;
Q, P: R1(10);
...
if Q.SA.all = Q.SB.all then ...
```

Remarquez l'indirection supplémentaire pour accéder au composant; ceci n'est pas une charge énorme en soi, mais cela rend l'affectation et la comparaison assez difficiles, car ce ne sont pas les valeurs accès que l'on cherche à comparer, mais les objets désignés. Ainsi

```
if Q = P then ...
```

sera transformé en :

```
if Q.D = P.D
   and then Q.SA.all = P.SA.all
   and then Q.SB.all = P.SB.all then ...
```

Nous avons en fait choisi une solution plus simple : la représentation effective pour un sous-type donné est calculée au moment de l'élaboration du sous-type, lorsque le sous-type est déclaré explicitement, ce qui permet de mettre en facteur le calcul de tous les déplacements des composants de tous les article de ce sous-type. Pour les objets créés sans déclaration explicite de sous-type (comme dans l'exemple de la boucle "for" ci-dessus), nous recalculerons l'adresse d'un composant donné à chaque accès. Le coût et la complexité de ce calcul croît avec la complexité de l'article, mais dans des cas comme celui de l'article de type R1 dans l'exemple ci-dessus, ce n'est guère plus cher que la déréférence supplémentaire de la solution des «pointeurs cachés» :

```
X: R1(Une_Expression);
```

alors

```
X. SB'ADDRESS
```

est égale à

```
    X'ADDRESS + X. D'SIZE + X. SA'SIZE + BOOLEAN'SIZE
=   X'ADDRESS + 1 +  (CHARACTER'SIZE * A. D) + 1
=   X'ADDRESS + A. D + 2
```

Mais cette solution conduit à d'autres difficultés avec des articles avec discriminants mais non contraints. Il faudrait allouer la taille maximale affectable aux objets de ce type non contraint, qui ont la propriété de «mutabilité» [Ichbiah 79b, DoD 84], c'est-à-dire de pouvoir changer leur discriminants par ré-affectation globale.

Considérons :

```
type UR1(D: INTEGER := 10) is record  -- mutable
   SA: STRING(1..D);
   SB: STRING(1..D);
end record;
subtype SR1 is UR1(10);
V: UR1;
W: SR1;
```

Si nous appliquons à SR1 le mécanisme décrit plus haut, V et W, bien que de même sous-type, n'ont pas la même représentation, et nous aurons à nouveau des difficultés lors des comparaisons, affectations, ou passages de paramètres. Pour simplifier, nous avons dans ce cas choisit la solution suivante (qui ne ravira pas le programmeur!) : tous les sous-types d'un type

article non contraint mutable ont la même représentation que le type, et donc les composant se voient affectée la taille maximale possible ; cette approche est d'ailleurs supportée par les pères du langage eux-mêmes ! [DoD 84]

Enfin pour résoudre le problème de la mutabilité de tels articles, notamment après qu'ils ont été passés en paramètres, et pour l'attribut CONSTRAINED, nous avons choisi d'ajouter un composant booléen supplémentaire dans l'article : *Constrained,* qui indique si l'objet est contraint ou non. Voir aussi § 5-2 et §7-8.

8. Représentation des tâches

Chaque tâche est représentée de façon interne par un nombre entier (un Word), qui désigne le segment de pile de la tâche. Toutes les informations liées à la tâche de trouvent dans ce segment, notamment dans le bloc de contrôle de tâche (cf. [Rosen 86]) et dans le contexte courant (cf. §2-4).

9. Représentation des fichiers

Comme pour les tâches, les fichiers (du type privé FILE_TYPE déclaré dans les paquetages prédéfinis d'entrées-sorties [LRM 14.1]) sont représentés par un article.

```
type FILE_TYPE is record
   MODE: FILE_MODE;
   NAME: STRING(1..NAME_LENGTH);
   FORM: STRING(1..NAME_LENGTH) := "";
   OPEN: BOOLEAN := TRUE;
   PAGE_NUMBER, LINE_NUMBER, VOLUM_NUMBER: COUNT := 0;
   PAGE_LENGTH, LINE_LENGTH: COUNT := UNBOUNDED;
   BUFFER: STRING(1..BUFFER_LENGTH);
end record;
```

Nota : Dans le prototype en SETL, les fichiers sont en fait représentés par un nombre entier, servant d'index dans des tuples donnant les informations liées aux fichiers :

```
UNINITIALIZED: constant := 0;
type FILE_TYPE is record
   FILE_NUM: INTEGER := UNINITIALIZED;
end record;
```

Ce choix est directement dérivé du prototype initial d'Ada/ED. Voir aussi [Rosen 86] pour une discussion des entrées-sorties.

Chapitre 4.
Elaboration des Types

Ce chapitre décrit les mécanismes de la Machine Ada impliqués dans le processus d'élaboration des types de données. Ils sont basés sur la notion de *patron de type,* entité hybride que nous introduisons ici, située entre le type Ada et l'objet.

1. Position du problème

A tout objet Ada est associé un type [LRM 3.2.1]. Mais si cette association entre un objet et son type est fixe, statique, les types eux-mêmes ou leurs sous-types ne sont pas nécessairement statiques ; certaines de leurs caractéristiques : bornes, tailles, composants, ne sont connues qu'à l'exécution. La construction d'objets de tels types requiert au préalable une activité de «construction» similaire à l'élaboration des objets, et qui prend place lors de l'*élaboration* du type ou sous-type. Dans le premier système Ada/Ed, il n'était fait aucune tentative pour élaborer statiquement les types ; toute l'élaboration était faite lors de l'exécution, et le résultat de cette élaboration était un objet, appelé *itype,* qui était stocké sous forme d'une structure de données récursive dans la *map* d'environnement (map au sens SETL, cf. annexe A) [Dewar 80, Kruchten 83]. Les objets étaient créés à partir de cet *itype,* mais contenaient eux-mêmes toutes les informations structurelles et de contraintes, ce qui était fortement redondant et rendaient l'*itype* un peu inutile.

Pour la machine Ada, nous avons dérivé de l'*itype* originel la notion de *patron de type*, au sens de «patron de couturier» (*type template* en anglais). Le premier rôle du patron de type est de mémoriser et de factoriser les informations nécessaires pour la construction des objets du type, leur taille par exemple. Mais il joue un plus grand rôle encore pour résoudre les problèmes des articles avec discriminants mutables. En effet, la sémantique d'Ada, très stricte sur ce point, requiert le test à l'exécution de la présence effective des composants sélectionnés. Comme de plus la structure de ces composants peut dépendre de ces discriminants, le patron de type va servir à conserver cette dépendance.

Pour avoir des mécanismes simples et très uniformes dans la Machine Ada, nous avons décidé d'étendre cette notion de patron de type à tous les types. Un patron se présente à l'exécution comme un objet de type article, et on peut s'en servir comme tel, c'est-à-dire le créer, le détruire, le passer en paramètre, etc. Chaque fois que cela est possible, le patron de type est calculé statiquement, lors de la génération de code. Si toutefois il était incomplet, son élaboration va consister à le compléter, et ceci est réalisé par des instructions spéciales de la machine (cf. §10-3).

Finalement, les patrons de type :

a) fournissent une définition complète, plate, non-récursive, d'un type, contenant toutes les informations nécessaires à la création d'objets, et aux tests de contraintes imposées par Ada ;

b) permettent de traiter tous les types et tous les objets de manière uniforme dans la Machine Ada.

Reprenant le titre d'un article de Donahue et Demers, nous pouvons dire que dans notre machine : *«Data types are values»* [Donahue 85].

2. Gestion des patrons de type

Nous avons vu aux chapitres 2 et 3 les options prises pour la gestion de la mémoire et la représentation des objets. Les piles ne contiennent que des adresses d'objets, jamais leur valeur effective, sauf au cours de l'évaluation d'une expression. Les valeurs sont stockées dans le tas, où l'espace est alloué à la demande. Lorsqu'on entre dans un bloc, de l'espace est réservé pour les adresses des objets locaux dans la pile (cf. fig. 6, p. 24), puis ensuite

dans le tas pour les objets proprement dit lors de leur élaboration. Ceci s'applique tant aux types structurés qu'aux types simples.

Les patrons de type seront traités de la même manière. Si le type est statique, il sera représenté par une constante globale dans un segment de données. Si non, l'élaboration consistera à construire un nouveau patron dans le tas à partir d'une constante globale incomplète, et à compléter dans cette copie les divers composants : taille, bornes, discriminants,...

La Machine Ada n'est absolument pas concernée par la notion de *type dérivé*, qui est complètement prise en charge par le frontal du compilateur. Un type dérivé et son type père se partagent donc un unique patron de type. Notons que ceci ne serait pas possible si la machine supportait des spécifications de représentation pour les types, ce qui n'est pas le cas.

Les patrons de type peuvent être passés en paramètre lors d'un appel de sous-programme ou d'entrée. C'est ainsi par exemple que l'on a implémenté les divers paquetages génériques pré-définis d'entrées-sorties : DIRECT_IO, SEQUENTIAL_IO, ainsi que ceux de TEXT_IO : ENUMERATION_IO, FIXED_IO, FLOAT_IO, INTEGER_IO [LRM 14]; le patron de type paramètre générique effectif ELEMENT_TYPE est passé comme paramètre de sous-programme supplémentaire aux routines du Noyau Exécutif.

Il y a 16 variantes de patron de type. Ceci inclut un patron de type pour les sous-programmes ; nous verrons au chapitre 7 que, pour des raisons d'uniformité, il est intéressant de les traiter comme tout autre objet (à l'instar des tâches), et qu'il y a effectivement une activité liée à leur élaboration.

3. Description des patrons de types

3.1 Types et sous-types entiers

tt_int_range : ce patron est utilisé pour des déclarations comme :

```
type ENTIER is range -100..100;
subtype ENT is INTEGER range 1..100;
```

tt_int_range
Taille
Borne supérieure
Borne Inférieure

Ce patron contient :

- la taille : 1 pour SHORT_INTEGER, 2 pour INTEGER, 4 pour LONG_INTEGER;
- une paire d'entiers : les bornes inférieure et supérieure.

Elaboration :

Dépile les bornes et les installe dans le patron.

3.2 Types énumératifs

tt_enum : ce patron est utilisé pour des déclarations comme :

```
type JOUR is (LUN, MAR, MER, JEU, VEN, SAM, DIM);
type MIXTE is ('a', 'b', RIEN);
```

tt_enum
Taille
Borne Supérieure
Borne Inférieure
Longueur 1
Littéral 1
Longueur 2
Littéral 2
...
Longueur n
Littéral n

Ce patron contient :

- la taille : 1 ou 2 ;

- une paire d'entiers : les bornes inférieure et supérieure.
- une table des littéraux correspondants, en ordre croissant, sous forme de paires (longueur de la chaîne, chaîne de caractères). Les apostrophes encadrant les caractères littéraux sont stockées dans la table.

  Ces littéraux d'énumération doivent être conservés à l'exécution pour l'attribut IMAGE et pour d'éventuelles instanciations du paquetage générique d'entrées-sorties ENUMERATION_IO [LRM 3.5.5(11) et 14.3.9]. En raison de la compilation séparée, et parce que notre implémentation n'a qu'un relieur très sommaire, il est difficile de se débarrasser de cette table s'il s'avère finalement qu'elle n'est pas nécessaire (cf. chap.11).

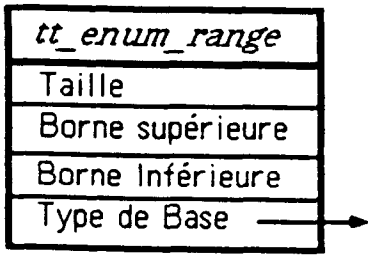
Elaboration :

Rien à faire : ce patron de type est toujours statique.

3.3 Sous-types énumératifs

tt_enum_range : ce patron est utilisé pour des déclarations comme :

```
type JOUR_DE_SEMAINE is JOUR range LUN..VEN;
```



Ce patron contient :
- la taille : 1 ou 2 ;
- une paire d'entier : les bornes inférieure et supérieure ;
- un pointeur vers le patron du type de base.

La différence avec le type énumération est qu'il n'est pas nécessaire de stocker les littéraux, mais plutôt un pointeur vers le type de base.

Elaboration :

Dépile les bornes et les installe dans le patron.

3.4 Types et sous-types réels à point-flottant

tt_float_range

tt_float_range
Taille
Borne supérieure
Borne Inférieure

Ce patron contient :

- la taille : 4 pour FLOAT, 8 pour LONG_FLOAT;
- une paire de réels : les bornes inférieure et supérieure.

Elaboration :

Dépile les bornes et les installe dans le patron.

3.5 Types et sous-types réels à point-fixe

tt_fixed

tt_fixed
Taille
Borne supérieure
Borne Inférieure
Base du pas
Position du point

Ce patron contient :

- la taille : 2, 4, 6 ou 8, selon le pas et l'intervalle.
- La base du pas : 2 ou 10
- l'emplacement du point décimal (-127 à 128)
- les bornes supérieure et inférieure

La machine Ada ne supporte pas les clause de longueur pour le pas d'un nombre réel à point-fixe dans toute sa généralité. Seules les puissances de 2 et de 10, les premières car elles sont imposées par le langage [LRM 3.9.5], les secondes car il nous a semblé qu'elles seraient très utiles au programmeur ; en fait, le pas est exprimé sous forme d'un produit d'une puissance de 2 et

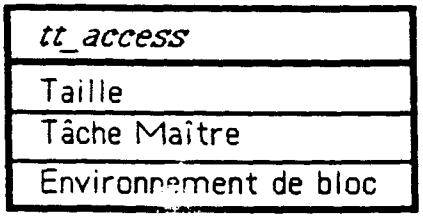
d'une puissance de 5. (Voir [Rosen 86] pour plus de précisions sur l'arithmétique réelle à point-fixe).

Elaboration :

Pour le type : les caractéristiques sont déterminées statiquement. Pour un sous-type : on dépile les bornes et les installe dans le patron.

3.6 Types accès

tt_access



Ce patron contient :

- la taille : 4
- le numéro de la tâche «maître» et
- la base de l'environnement de bloc «maître».

Un type accès doit être relié à l'environnement de bloc actif au moment de son élaboration, afin de pouvoir ultérieurement relier à cet environnement la collection des objets désignés par des valeurs de ce type [LRM 4.8(7)], ainsi que les tâches dépendantes [LRM 9.4(2)].

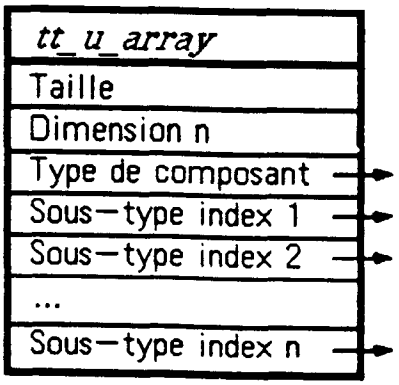
Elaboration :

Recopie le contenu des registres Pointeur de tâche et Base d'environnement de bloc dans le patron.

3.7 Types tableau non contraint

tt_u_array : ce patron est utilisé pour des déclarations comme :

```
type ARR is array(INTEGER range <>) of INTEGER;
```



Ce patron contient :

- la taille ;
- le nombre de dimensions du tableau *n*,
- un pointeur vers le patron du type du composant,
- et *n* pointeurs vers les patrons des sous-types d'indice; notez qu pointeurs sont stockés en ordre inverse, pour des raisons d'efficac lors d'une opération d'indexation.

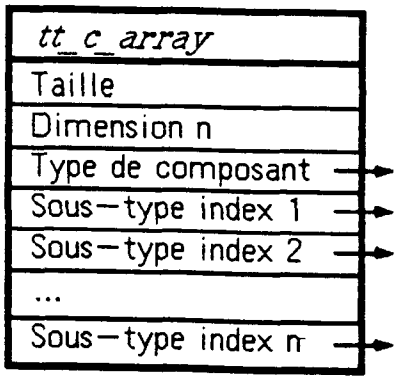
Elaboration :

Aucun objet ne pouvant être directement construit à partir de ce patron, il n'y a rien à effectuer lors de l'élaboration. La mise à jour des sous-types d'indice et du sous-type du composant sera faite pour le sous-type (contraint) uniquement.

3.8 Types tableau contraint

tt_c_array : ce patron est utilisé pour des déclarations comme :

```
subtype ARR10 is ARR(1..10);
```



Ce patron a une structure identique à celui d'un type tableau non contraint (cf. ci-dessus), la seule différence étant qu'il contient des pointeurs vers les sous-types du composant et des indices.

Elaboration :

Met à jour les pointeurs vers les patrons des sous-types; utilise les tailles respectives de ces patrons pour calculer la taille effective du tableau.

Note :

Même dans le cas d'une déclaration de la forme :

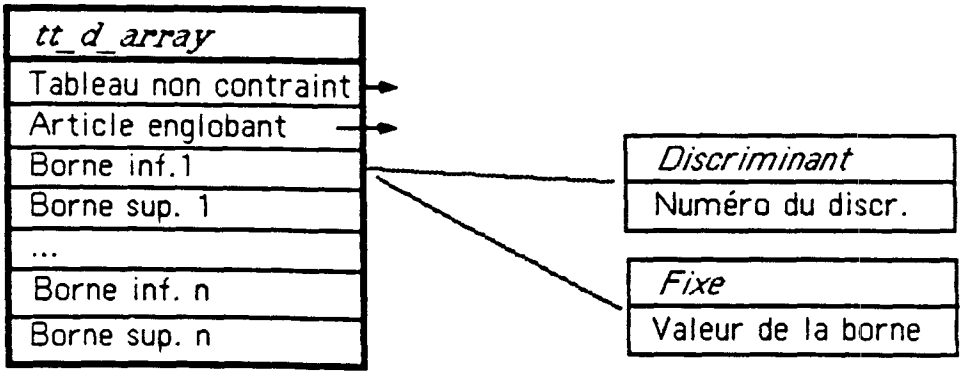
```
type ARR10 is array(1..10) of INTEGER;
```

on construit un patron non contraint et un patron contraint, car le frontal du compilateur applique à la lettre l'équivalence donnée dans le Manuel de référence [LRM 3.6(5)] en créant un type anonyme intermédiaire.

3.9 Types tableau dont les bornes dépendent de discriminants

tt_d_array : ce patron est utilisé pour des déclarations comme :

```
type REC(D: INTEGER) is
   record
      A: ARR(1..D); -- ce composant
   end record;
```



Ce patron contient :

- un pointeur vers le patron du type tableau non contraint (ARR),
- un pointeur vers le patron du type article englobant (REC),
- et, pour chaque dimension, et pour chaque borne (inférieure et supérieure),

- si la borne dépend d'un discriminant, le numéro de ce discriminant,
- si la borne ne dépend pas d'un discriminant, sa valeur effective.

Ceci est le patron d'un sous-type d'un composant d'un article dont au moins une des bornes dépend d'un discriminant de l'article. Il n'est pas possible dans ce cas d'utiliser un patron *tt_c_array*, car les bornes doivent être déduites du patron de l'article. Toutefois un patron *tt_c_array* est construit si l'on accède à ce composant pour une autre raison que d'y faire une indexation.

Elaboration :

Si l'une au moins des expressions données dans l'indication de sous-type tableau n'est pas un discriminant, alors toutes les bornes qui ne sont pas des discriminants se trouvent sur la pile. Cf [LRM 3.7(8)].

3.10 Types tableau unidimensionnels

tt_s_array : ce patron est construit par quelques opérations de base, comme la concaténation, et les opérations prédéfinies retournant des chaines de caractères (type STRING).

tt_s_array
Taille
Taille composant
Taille index
Borne Inférieure
Borne Supérieure

Ce patron contient :

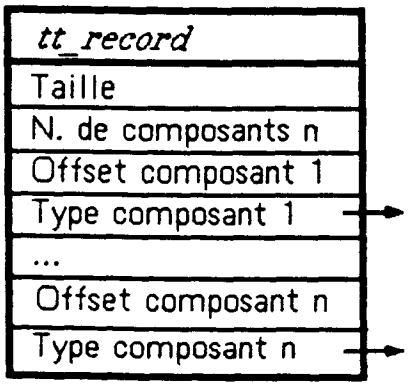
- la taille ,
- la taille du composant la taille de l'indice,
- une paire d'entiers : les bornes inférieure et supérieure.

Elaboration :

Il n'est jamais élaboré explicitement ; il est soit rendu par l'évaluation de certaines opérations de base ou prédéfinies, soit construit statiquement pour les littéraux de chaines de caractères.

3.11 Types article sans discriminant

tt_record



Ce patron contient :

- la taille ,
- le nombre total de composants,
- pour chaque composant, un couple (déplacement dans l'article du début du composant, pointeur vers le patron de type du composant).

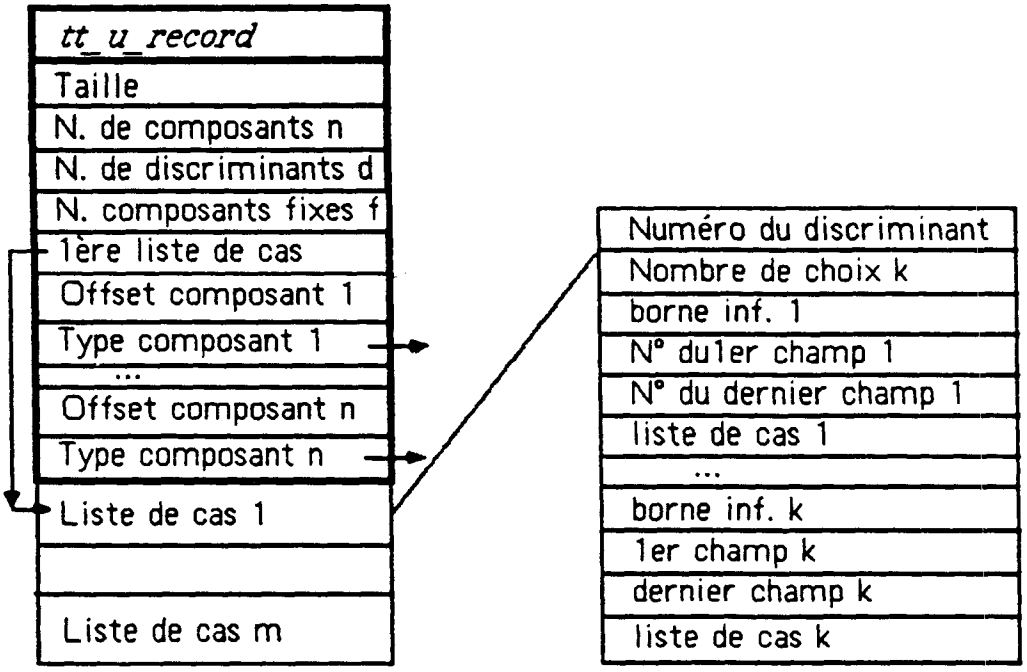
Si la taille d'un composant n'est pas connue statiquement, alors les déplacements de tous les composants qui suivent ne peuvent pas être déterminés statiquement; ils seront calculés lors de l'élaboration et on utilisera l'instruction *select* pour accéder à ces composants.

Elaboration :

Met à jour les pointeurs vers les patrons des composants non statiques ; utilisant leur tailles, calcule les déplacements manquants, puis la taille totale de l'article.

3.12 Types article non contraints avec discriminants

tt_u_record



Ce patron contient :

- la taille ;
- le nombre total de composants (toutes variantes cumulées),
- le nombre de discriminants,
- le nombre de champs de la partie fixe,
- le déplacement (dans le patron de type) de la première liste de cas, représentant une variante (0 s'il n'y en a pas).
- enfin pour chaque composant, un couple (déplacement dans l'article du début du composant, pointeur vers le patron de type du composant).

Une liste de cas est composée de :

- numéro du discriminant gouvernant cette variante,
- le nombre de choix dans la variante,
- pour chaque choix, le quadruplet (borne inférieure, premier composant, dernier composant, liste de cas emboîtée).

Les remarques faites pour l'article sans discriminant s'appliquent aussi ici. Les déplacements des composants de partie variante sont tels que des variantes mutuellement exclusives se recouvrent en mémoire.

On rajoute un composant dans la partie fixe de l'article, le composant booléen *Constrained* qui est utilisé pour déclencher des tests lors de l'affectation à l'article (cas des article «mutables»), notamment lorsque l'objet article a été passé en paramètre de mode "out".

Les choix de la table de variante sont triés en ordre croissant de borne inférieure et recouvrent tout l'intervalle du sous-type du discriminant. Cette structure est identique à celle utilisée pour l'instruction *case*.

Elaboration :

Met à jour les pointeurs vers les patrons de type des composants, calcule les déplacements manquants, utilise les tailles des composants pour déterminer la taille de l'article, en fait la taille de la plus longue variante.

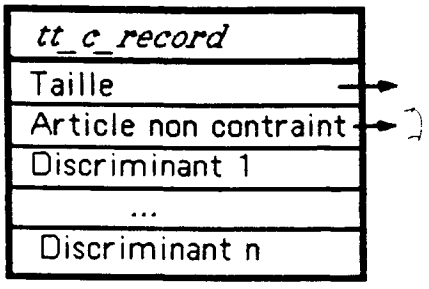
Note :

Contrairement au prototype Ada/ED, il n'y a pas de composant spécial "field_present" indiquant quel composants sont effectivement présents, et l'instruction *select* utilise le patron de type et les valeurs effectives des discriminants pour déterminer la présence d'un composant. Cette solution est encombrante ou requiert de la Machine des possibilités de manipulation de bits. Le Delft Ada Subset propose de générer une fonction de test optimisée [Katwijk 84].

3.13 Types article contraint avec discriminants

tt_c_record : ce patron est utilisé pour des déclarations comme :

```
subtype REC10 is REC(10);
```



Ce patron contient :

- la taille de l'article;
- un pointeur vers le patron du type non constraint,

- la valeur effective de chaque discriminant.

Le pointeur vers le patron d'article non contraint permet à l'instruction *select* d'y accéder sans avoir à dupliquer les tables des déplacements et les tables de variantes.

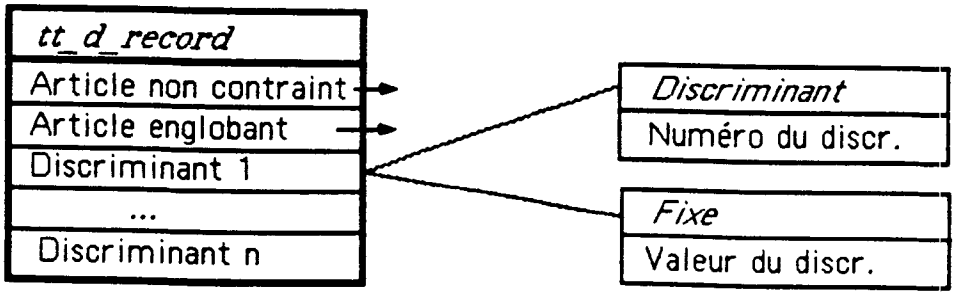
Elaboration :

Si l'une au moins des expressions donnant les discriminant n'est pas statique, alors tous les discriminants sont dépilés et rangés dans le patron [LRM 3.7.2(13)].

3.14 Type article dont les discriminants dépendent de discriminants

tt_id_record : ce patron est utilisé pour des déclarations comme :

```
type R(D: INTEGER) is
   record
      C: REC(D); -- sous-type dépendant d'un discriminant
   end record;
```



Ce patron est similaire à *tt_d_array* à ceci près qu'il donne les dépendances ou les valeurs des discriminants et non pas des bornes.

Elaboration :

Si l'une au moins des expressions de l'indication de sous-type n'est pas statique, alors les valeurs de tous les discriminants qui ne dépendent pas des discriminants de l'article englobant se trouvent sur la pile, et sont recopiées dans le patron [LRM 3.7(8)].

3.15 Types tâches

tt_task

Nous avons vu (cf. §3-8) qu'une tâche est représentée en interne par un entier : le numéro de son segment de pile, qui contient tous les renseignements qui lui sont associés.

Son patron contient :

- la priorité statique (donnée par un pragma PRIORITY),
- un pointeur vers le corps, qui est un objet sous-programme dans notre machine (cf. chap. 7),
- le nombre total d'entrées,
- le nombre total de familles d'entrées,
- la table des entrées : une par famille d'entrée, une entrée simple étant assimilée à une famille d'entrées d'indice : INTEGER **range** 1..1 .
  + si le type tâche est statique, la taille de chaque famille et son déplacement dans la table des entrées.
  + si le type n'est pas statique, l'adresse du sous-type d'indice de la famille.

Il faut en effet conserver à l'exécution suffisamment d'information pour réaliser les calculs d'entrée et notamment les opérations d'indexation et de test de contrainte. Une expression d'entrée est évaluée comme un triplet (numéro de tâche, numéro de famille d'entrée, index), où l'index vaut 1 par convention pour une entrée simple. Chaque tâche conserve un lien en arrière vers son patron de type. La table des entrées est également utilisée lors de la création de tâche pour dimensionner la table des listes d'attente sur chaque entrée dans le Bloc de Contrôle de Tâche, cf. [Rosen 86].

Elaboration :

Si l'un au moins des sous-type d'indice des familles d'entrées n'est pas statique, alors toute la table des entrées est calculée lors de l'élaboration, après avoir mis à jour les pointeurs sur les patrons des sous-types d'indice.

3.16 Sous-programmes

Le patron de sous-programme contient :

- le numéro de segment de code correspondant,
- un indicateur d'élaboration,
- la table de relais avec sa longueur,
- et le nombre d'objets locaux.

Voir chapitre 7 pour élaboration et usage de ce patron.

4. Exemple de mise en œuvre

Nous allons considérer un fragment de partie déclarative contenant plusieurs déclarations de type, et montrer la structure de données correspondante après élaboration.

```
type COULEUR is (BLEU, BLANC, ROUGE);
subtype COUL2 is COULEUR range BLEU..BLANC;
type INT is new INTEGER range 1..5;
type TABLE is array(INT range <>, INT range <>) of COUL2;
type ARTICLE(DISCR: INT) is
   record
      T : TABLE(1..DISCR, 1..DISCR);
      C : COUL2;
   end record;
subtype ARTICLE_3 is ARTICLE(DISCR => 3);
```

La structure de données qui en résulte est montrée à la figure 10 page suivante ; ARTICLE.T est le patron du sous-type (anonyme) du composant T de l'article ARTICLE.

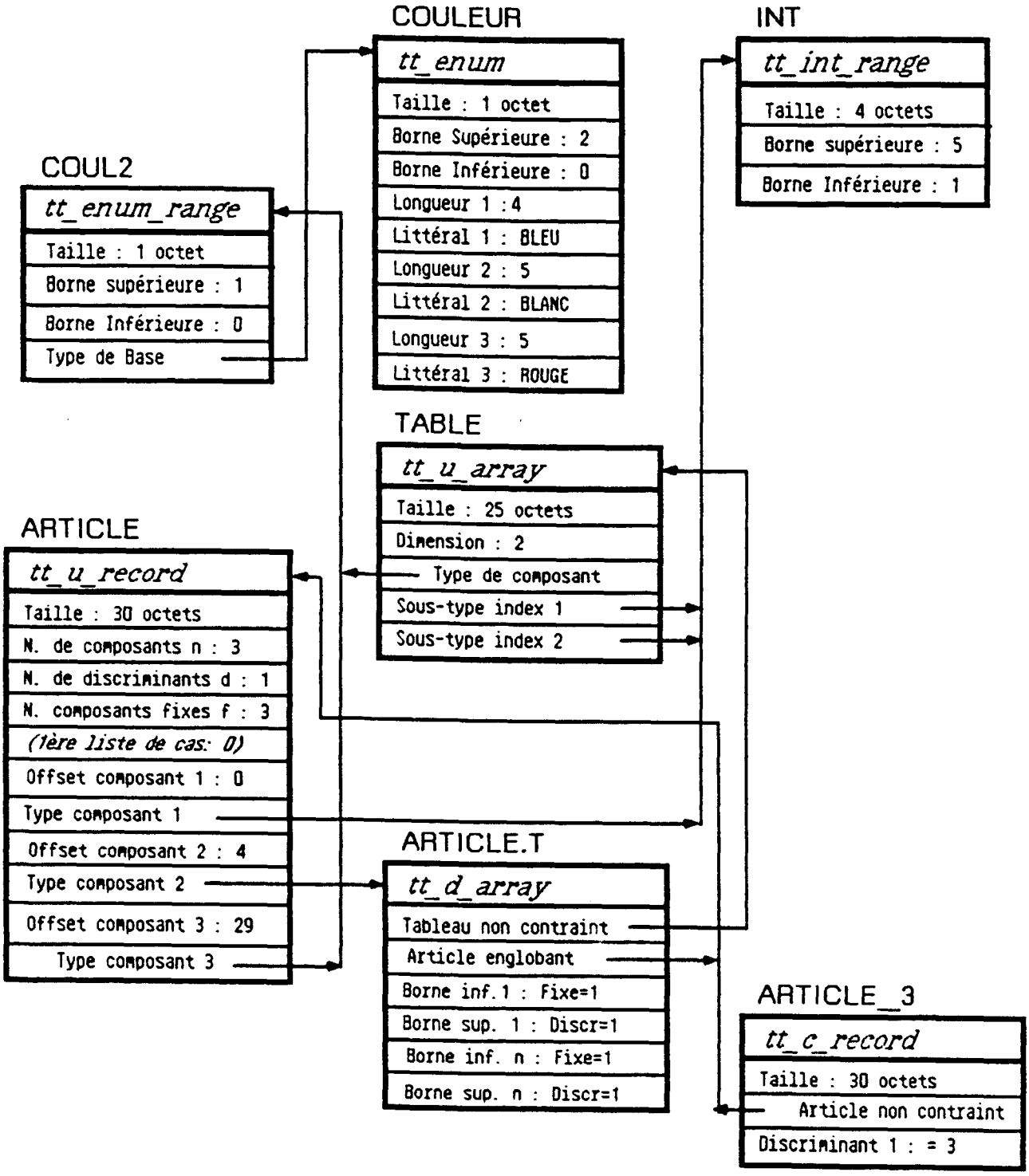


Figure 10 - Exemple de patrons de type

5. Discussion

La conception de la notion de patron de type a été faite à N.Y.U. à une époque où nous ignorions tout des études sur le Ada Breadboard Compiler aux Bell Laboratories, de l'autre côté de l'Hudson [Wetherell 82]. Il se trouve que notre mécanisme ressemble de façon frappante à ce qu'ils proposent, jusqu'aux noms mêmes donnés aux patrons de type [Rubine 82] ! Il semble

toutefois que les concepteurs du Ada Breadboard Compiler n'aient pas vu quelques problèmes délicats, comme les articles avec variantes, les types tâche et les dépendances entre les blocs, les tâches et les types accès ; il est clair que leur conception ne permet pas de passer en paramètre de mode "out" un composant d'article qui serait lui-même un article dont les discriminants dépendent de l'article englobant. Il faut dire à leur décharge qu'ils sont partis du Manuel de Référence du Langage, alors que nous avions déjà un traducteur complet du langage, qui allait être validé quelques mois plus tard. Un point à mettre à l'actif du prototypage de logiciel.

Le Delft Ada Subset Compiler [Katwijk 84] présente aussi un modèle assez proche du nôtre, dans lequel le cas des composants de type article dont les bornes ou discriminants dépendent des discriminants de l'article englobant est traité (utilisation d'un *call block*). Ils proposent un mécanisme différent pour le test d'existence lors de la sélection un composant d'article : ils génèrent une *path function* pour chaque article avec discriminant; dans notre cas, cette fonction est implicitement contenue dans la table de variantes du patron *tt_u_record* (cf. §3.12 ci-dessus), qui est utilisé par les instructions *select* et *qual_discr* (cf. § 10-6). Par ailleurs, l'inconvénient du Delft Ada Subset est qu'il généralise la notion de "doublet" à tous les objets et valeurs.

Les auteurs du Charette Ada Compiler de Carnegie-Mellon proposent un mécanisme similaire au nôtre, mais ils utilisent des pointeurs cachés ; ils ne linéarisent pas les variantes, mais les considèrent comme des articles emboités ; n'ayant pas de tâches, ils ignorent les problèmes de dépendance du maitre, et ils ne partagent pas les descripteurs. Ce dernier point semble évoluer avec ses descendants : le Spice compiler et l'Ada+ compiler. On trouve un «constrained bit» dans les articles comme nous le proposons.

Les autres publications disponibles sur des traducteurs Ada ne décrivent rien de ressemblant aux patrons de type. Ils y a des «descripteurs de type» dans l'A.L.S., sans plus de précision [Kamrad 83]. Ils y a des descripteurs de type pour les tests d'intervalle dans la A-Machine [Ibsen 83, Ibsen 84], dans le deuxième compilateur de Telesoft [Fisher 83], et dans l'implémentation de Karlsruhe, mais rien n'est dit sur le traitement des types composites complexes, et notamment sur les discriminants.

Chapitre 5.
Elaboration des Objets

Ce chapitre décrit l'élaboration des objets dans la Machine Ada, et plus particulièrement la manière dont sont traitées les valeurs initiales par défaut. Puis on y discute de l'association entre valeur et type.

1. Position du problème

L'élaboration d'une déclaration d'objet se déroule en quatre phases [LRM 3.2.1(4-8)] :

- ◇ le sous-type de l'objet est élaboré,
- ◇ une valeur initiale explicite, ou à défaut implicite, de l'objet est évaluée,
- ◇ l'objet est créé,
- ◇ la valeur initiale est affectée à l'objet.

La valeur initiale implicite est pour un type accès la valeur "null", pour un type tâche, la tâche associée. Mais elle peut aussi, dans le cas des articles être donnée par des expressions pour ses composants, expressions qui doivent être évaluées à *chaque* création d'objet, et non pas juste lors de l'élaboration du type (comme c'était le cas en Ada millésime 1980 [DoD 80]).

L'élaboration du sous-type, s'il n'est pas statique, correspond au calcul du patron de type correspondant (cf. §3-2). La création de l'objet est immédiate, disposant de sa taille dans le patron de type, à moins que l'objet, étant statique n'existe déjà dans le segment de données global. L'affectation est similaire en tous points à ce qui se passe lors de l'opération d'affectation.

Seul le calcul des valeurs initiales par défaut nous a posé un problème, plus au niveau de la *génération de code* qu'au niveau de la conception de la Machine Ada proprement dite.

Pour illustrer au cours de ce chapitre les divers problèmes posés par l'élaboration des objets, nous considérerons le fragment de programme suivant :

```
function F return INTEGER; -- pour avoir des valeurs non
                           -- statiques
task type TYPE_TACHE;
type REC is
  record
    I : INTEGER := 50;
    J : INTEGER := F;
  end record;
type ACC is access REC;
type REC1 (D : INTEGER) is
  record
    V : INTEGER := F + D;
  end record;
subtype SREC1 is REC1(F);
type REC2 (D : INTEGER := 2*F+3 ) is  -- mutable
  record
    T : TYPE_TACHE;
    R : REC1(D);
  end record;
subtype SREC2 is REC2(F);
type ARR is array(INTEGER range <>) of REC2;
subtype ARR10 is ARR(1..10);
A : TYPE_TACHE;
B : REC;
C : ACC;
D : REC := (43,28);
E : SREC1;
G : REC2;
H : SREC2;
K : ARR10;
```

2. Procédures internes d'initialisation

Nous laisserons de côté les cas évidents, comme celui où tous les composants d'un article ont pour valeurs initiales par défaut des constantes statiques, auquel cas il suffit de calculer un agrégat lors de la compilation, et de le recopier dans l'objet créé.

2.1 Types tâche et accès

Pour les types tâche, nous considérerons que la déclaration d'objet (ou de composant) a une expression initiale consistant en un appel à une fonction interne qui retourne une valeur de ce type tâche, après avoir créé une telle tâche. Ainsi à :

```
task type TYPE_TACHE;
A : TYPE_TACHE;
```

correspondrait (en simili-Ada) :

```
generic
   type UN_TYPE_TACHE is private;
function CREATION_DE_TACHE_GENERIQUE return UN_TYPE_TACHE;
function CREER_TYPE_TACHE is
     new CREATION_DE_TACHE_GENERIQUE (TYPE_TACHE);
A : TYPE_TACHE := CREER_TYPE_TACHE;
```

Pour une tâche singulière, nous nous contenterons d'appliquer l'équivalence donnée dans le Manuel de Référence [LRM 9.1(2)], en créant un type tâche anonyme.

Pour les types accès, de manière similaire, nous considérerons que la déclaration a une expression par défaut, formée d'une constante du type accès. Ainsi, on transformera :

```
type ACC is access REC;
C : ACC;
```

en :

```
function CAST is
    new UNCHECKED_CONVERSION ( SYSTEM.ADDRESS, ACC );
ACC_NULL : constant ACC
     := CAST(SYSTEM.ADDRESS'( SEGMENT_TYPE'LAST,
                              OFFSET_TYPE'LAST));
C : ACC := ACC_NULL;
```

2.2 Type article

Pour les types article sans discriminant, on construit une procédure interne avec un paramètre de mode "in out" du type, procédure qui réalisera l'initialisation des composants un à un. Ainsi au code :

```
type REC is
   record
      I : INTEGER := 50;
      J : INTEGER := F;
   end record;
B : REC;
```

correspond (en simili-Ada) :

```
procedure REC_INIT(PARAM : in out REC) is
begin
   PARAM.I := 50;
   PARAM.J := F;
end REC_INIT;
B : REC;
REC_INIT(PARAM => B);
```

en supposant bien entendu que dans notre simili-Ada «B : REC;» n'effectue plus aucune initialisation.

2.2.1 Initialisation secondaire

Pour un article avec discriminant sans valeurs par défaut pour les discriminants, nous construisons une procédure similaire. Notons qu'aucun objet ne peut être construit sans donner une indication de sous-type. Et donc aux déclarations :

```
type REC1( D : INTEGER ) is
   record
      V : INTEGER := F + D ;
   end record;
```

nous ferons correspondre une procédure interne, dite *d'initialisation secondaire* :

```
procedure REC1_INIT2 ( PARAM : in out REC1) is
begin
   PARAM. CONSTRAINED := TRUE;
   PARAM. V := F + PARAM. D;
end REC1_INIT2;
```

2.2.2 Initialisation primaire

Pour les sous-types de types article avec discriminants, on construit une autre procédure interne, dite d'*initialisation primaire*, qui initialise les discriminants et qui est appelée avant l'initialisation secondaire.

```
subtype SREC1 is REC1(F);
D : SREC1;
```

sera transformé en :

```
SREC1_D : constant INTEGER := F;
procedure SREC1_INIT(PARAM : in out REC1) is
begin
   PARAM.D := SREC1_D;
end;
D : SREC1;
SREC1_INIT(PARAM => D);   -- init. primaire (sous-type)
REC1_INIT2(PARAM => D);   -- init. secondaire (type)
```

Le discriminant est en pratique évalué une seule fois, lors de l'élaboration du sous-type SREC1 et stocké dans le patron de type *tt_c_record* associé à SREC1 (cf. §4-3.12).

Note : nous supposons bien entendu qu'en simili-Ada, une telle affectation au discriminant est autorisée.

2.2.3 Cas des articles mutables

Pour un article avec discriminants qui a des valeurs par défaut pour les discriminants, on pourra créer soit des objets mutables (auquel cas il faut évaluer les valeurs par défaut des discriminants) soit des objets non mutables, auquel cas on aura créé un sous type. Nous construirons donc deux procédures d'initialisation, une initialisation primaire pour les discriminants, et une initialisation secondaire pour les autres composants. Pour un sous-type, on est ramené au cas mentionné ci-dessus. Ainsi à :

```
type REC2 (D : INTEGER := 2*F+3 ) is  -- mutable
   record
      T : TYPE_TACHE;
      R : REC1(D);
   end record;
subtype SREC2 is REC2(F);
G : REC2;
H : SREC2;
```

correspond en simili-Ada :

```
procedure REC2_INIT2 (PARAM : in out REC2) is
begin
   PARAM.T := CREER_TYPE_TACHE;
   PARAM.R.D := D;
   REC1_INIT2(PARAM.R);
end REC2_INIT2;

procedure REC2_INIT(PARAM : in out REC2) is
begin
   PARAM.CONSTRAINED := FALSE;
   PARAM.D := 2*F+3;
end REC2_INIT;

SREC2_D : constant INTEGER := F;
procedure SREC2_INIT(PARAM : in out REC2) is
begin
   PARAM.CONSTRAINED := TRUE;
   PARAM.D := SREC2_D;
end SREC2_INIT;

G : REC2;
REC2_INIT(G);                    -- primaire
REC2_INIT2(G);                   -- secondaire

H : SREC2;
SREC2_INIT(H);                   -- primaire
REC2_INIT(H);                    -- secondaire
```

2.2.4 Traitement des parties variantes

Pour un article avec discriminants et parties variantes, la procédure d'initialisation secondaire tire parti de la similitude qui existe en Ada entre la sémantique de la partie variante et celle de l'instruction "case" ; ainsi à :

```
type VARIANT_REC(D1, D2 : INTEGER) is
   record
      B: BOOLEAN := TRUE;
      case D1 is
         when 1 | 6 | 35..65 =>
            U1: SREC1;
            U2: INTEGER := 3*F+30;
         when 5 =>
            V1 : TYPE_TACHE;
            case D2 is
               when 1 | 5 => null;
               when others =>
                  X : ACC;
            end case;
         when others =>
            W : INTEGER := 1;
      end case;
   end record;
```

correspond en simili-Ada :

```
procedure VARIANT_REC_INIT_2(PARAM: in out VARIANT_REC) is
  begin
    PARAM.CONSTRAINED := TRUE;
    PARAM.B := TRUE;
    case D1 is
      when 1 | 6 | 35..65 =>
        SREC1_INIT(PARAM.U1);
        PARAM.U2 := 3*F+30;
      when 5 =>
        PARAM.V1 := CREER_TYPE_TACHE;
        case D2 is
          when 1 | 5 => null;
          when others =>
            PARAM.X := NULL_ACC;
        end case;
      when others =>
        PARAM.W := 1;
    end case;
  end VARIANT_REC_INIT_2;
```

2.3 Type tableau

Pour les types tableau, c'est la boucle "for" qui permet d'obtenir l'effet désiré. Il n'est pas nécessaire de distinguer entre tableaux contraints et non contraints car on ne peut en fait créer que des tableaux contraints, et d'autre part pour une procédure qui aurait un paramètre de mode "out" de type tableau non contraint, le sous-type du paramètre effectif est passé en paramètre supplémentaire (cf §3-6).

```
type ARR  is array(INTEGER range <>) of REC1;
subtype ARR10 is ARR(1..10);
Q: ARR10;
```

donne :

```
procedure  ARR_INIT(PARAM: in out ARR) is
  begin
    for I in PARAM'RANGE loop
       REC1_INIT(PARAM(I));
    end loop;
  end ARR_INIT;
Q: ARR10;
ARR_INIT(PARAM => Q);
```

Si le tableau est multidimensionnel, la procédure d'initialisation contient des boucles imbriquées.

2.4. Particularités des procédures internes d'initialisation

Les procédures internes d'initialisation que nous venons de présenter ne sont pas tout à fait à mettre sur le même plan que les procédures Ada écrites par le programmeur, une fois de plus à cause du problème des liens de dépendance entre les objets et les blocs où sont déclarés ces objets (ou le type accès, pour un allocateur).

En effet, ces procédures sont susceptibles de créer des tâches, tâches qui bien sûr ne doivent pas dépendre de la procédure d'initialisation qui les a créées, mais du bloc qui a invoqué directement ou indirectement cette procédure d'initialisation. En pratique les procédures d'initialisation n'ont pas de partie déclarative, et jamais d'instruction *activate* à la fin de cette partie déclarative.

Les tâches créées par les procédures du genre CREER_TACHE (cf. §2.1 ci-dessus) sont normalement chaînées sur la liste des *tâches_déclarées* du bloc courant, mais lorsqu'on quitte un bloc par l'instruction *leave_block*, les tâches déclarées mais non activées dans le bloc qu'on quitte sont rajoutées à la liste des tâches déclarées du bloc englobant.

Par contre, pour l'évaluation d'un allocateur (primitive "new") qui crée un objet dont un composant ou un sous-composant est une tâche, l'activation est faite à la fin de l'évaluation de l'allocateur, donc juste après l'évaluation de la procédure d'initialisation ; le maître dans ce cas n'est pas le bloc courant, mais le bloc qui a élaboré le type accès (cf. 4-3.6), et ce ne peut pas être une procédure interne d'initialisation.

3. Une architecture partiellement «taggée»

Les chapitres 3, 4 et 5 ont présenté les types, objets et valeurs manipulés dans la machine Ada et nous allons ici conclure sur cette partie.

Le *tagging* (ou marquage, ou typage) consiste à rajouter aux données d'exécution une quelconque forme d'auto-description, dans des buts de contrôle de type, de gestion de mémoire ou de direction du flot de contrôle. Ce concept a été considérablement développé surtout pour les langages à allocation dynamique de mémoire (LISP, APL,...), pour les langages où le lien entre objet et type est dynamique (Smalltalk-80), et des architectures matérielles ont été proposées intégrant ce marquage de façon très systématique [Myers 78]; nous l'avions proposé pour la machine Arcade [Kruchten 78], il existe des propositions pour Pascal [Schulthess 82], et même pour Ada [Bishop 80].

Mais en fait, comme l'ont montré plus récemment Van Vliet et Gladney, le marquage n'est réellement utile que pour un nombre limité de valeurs [Vliet 85], et d'autre part l'association systématique des marques aux valeurs et aux objets comme le proposent le Delft Ada Subset, avec leur "doublet model" [Katwijk 84] ou l'Ada Breadboard Compiler [Wetherell 82] est extrêmement gourmande en mémoire.

Dans la machine Ada, les "tags" ou marquages que nous proposont sont bien entendu les patrons de type, mais :

(a) ces patrons de type sont mis en facteur pour tous les objets d'un même type ou sous-type, types dérivés confondus ;

(b) ces patrons de type ne sont associés *systématiquement* qu'aux objets ou valeurs de types tableau non contraints ;

(c) les patrons de type ne servent que pour un nombre limité d'opérations bien précises :

- élaboration ou allocation d'un objet,
- sélection dans un objet (tableau et parfois article),
- qualification,
- conversion de type.

Les instructions réalisant ces opérations ont alors quelque chose de «générique», leur exécution étant dirigée par le patron de type. Le

reste du jeu d'instructions de la machine Ada n'a pas à se soucier des marquages, des patrons de type ; il travaille sur des objets des types simples de base (cf. §3-2).

Nous avons donc une architecture «partiellement taggée», pour reprendre l'expression de Van Vliet, qui nous paraît réaliser un bon compromis entre la complexité de la compilation, l'encombrement mémoire, la complexité du jeu d'instruction et l'efficacité à l'exécution.

Les descendants du Charette compiler : Spice Ada Compiler et Ada+ semblent avoir évolué dans cette voie [Barbacci 85].

Chapitre 6.
Évaluation des expressions

L'évaluation des expressions arithmétiques et logiques à l'aide d'une pile est un mécanisme simple et bien connu sur lequel nous ne nous étendrons pas. Dans ce chapitre, nous détaillerons un point qui nous a donné bien du fil à retordre : l *'évaluation des agrégats*. Par ailleurs l'évaluation des expressions arithmétiques réelles à point-fixe est détaillée dans [Rosen 86].

1. Agrégats article

Si l'article a des discriminants, ces derniers sont évalués en premier afin de déterminer le sous-type exact de la valeur. Notons que le langage impose des discriminants statiques lorsqu'ils gouvernent des parties variantes [LRM 4.3.1(2)]. Un objet temporaire de ce sous-type est créé, initialisé avec la procédure d'initialisation secondaire associée à ce sous-type (cf. §5-2), puis enfin les composants sont garnis par des affectations individuelles. Illustrons ceci par la transformation suivante :

```
type REC;
type ACC is access REC;
type REC(D: INTEGER) is
   record
      S: STRING(1..D);
      U,V: ACC;
   end record;
TROIS : INTEGER := 3;
V: constant REC(TROIS, S=>(1..TROIS=>' '), U | V=> new REC(0));
```

qui donne :

```
subtype RECTROIS is REC(TROIS);
V: RECTROIS;
REC_TROIS_INIT(V); -- cf. chap.5
V.S := (1..TROIS => ' '); -- voir ci-dessous, agrégats tableau
V.U := new REC(0);
V.V := new REC(0);
```

2. Agrégats tableau

Ils sont plus complexes à évaluer. Le Manuel de Référence donne l'ordre suivant d'élaboration (et ce qui n'y est pas dit explicitement doit être lu entre les lignes, ou déduit de la suite de validation !) :

(1) Evaluer tous les choix, y compris les choix des sous-agrégats; ne pas vérifier si les bornes appartiennent au sous-type d'indice ou non.

(2) pour un tableau multidimensionnel, vérifier que tous les sous-agrégats ont les mêmes bornes et lever CONSTRAINT_ERROR sinon.

(3) calculer la taille du tableau résultant, par élaboration d'un sous-type tableau anonyme ;

(4) si, et seulement si ce n'est pas un tableau vide (null array), lever CONSTRAINT_ERROR si les bornes des indices n'appartiennent pas aux sous-types d'indices ;

(5) créer le tableau et évaluer les composants en évaluant les expressions autant de fois qu'il y a de composants imposés par les choix.

Tous les agrégats tableau ne passent pas par toutes ces étapes :

- si le tableau est unidimensionnel, le point (2) est sans objet;
- si les bornes sont statiques ou contiennent un choix "others", les points (2) et (3) peuvent être réalisés à la compilation ;
- si l'expression du composant est statique ou sans effet de bord, le point (5) peut être optimisé.

Les points (1), (2) et (3) sont heureusement restreints par les règles suivantes [LRM 4.3.2(3)] :

- outre un choix "others" final, le reste de l'agrégat doit être soit positionnel, soit nommé ;

- une association nommée n'est autorisée à avoir un choix non statique (ou un choix d'intervalle vide) que si l'agrégat n'a qu'une seule association de composant, et que cette association de composant n'a qu'un seul choix.
- Le choix "others" est statique si le sous-type d'indice est statique.

Voyons sur un exemple, en utilisant les mécanismes de transformation que nous avons déjà utilisés, quel sorte de code doit être généré pour le cas difficile. Les variables UN, DEUX, TROIS etc, sont là pour rendre des choix non statiques.

Soient les déclarations suivantes :

```
type A is array(  INTEGER range <>,
                  INTEGER range <>,
                  INTEGER range <>  ) of INTEGER;

ONE, UN, ICHI: constant INTEGER:=1;
THREE, TROIS, SAN: constant INTEGER := 3;
subtype SINT is INTEGER range UN..TROIS;
subtype AS is A(1..3,SINT,SINT);

procedure P(Q: in AS);  -- pour éviter d'être dans le cas de [LRM 4.3.2(6)]
function F return INTEGER; -- pour avoir des expressions non statiques
```

Voyons comment évaluer l'agrégat suivant :

```
P(
   (  1 =>
         (  -- positionnel
            (ONE..THREE =>F),  -- nommé, non statique
            (1|2=>F,3=>0),     -- nommé, statique
            (UN..TROIS => F)   -- nommé, non statique
         ),
      others =>
         (  -- nommé, statique
            1..2 => (ICHI..SAN => F),
            3 => (F, UN, F)    -- positionnel
         )
   )
);
```

◇ L'étape (1) calcule les choix et détermine les bornes, selon les règles de [LRM 4.2.3(9)] :

```
subtype bornes is INTEGER range 1..3;  -- de la définition de AS

subtype bornes1 is SINT'BASE range SINT'FIRST..SINT'FIRST+2;
subtype bornes2 is SINT'BASE range 1..3;

subtype bornes1_1 is SINT'BASE range ONE..THREE;
subtype bornes1_2 is SINT'BASE range 1..3;
subtype bornes1_3 is SINT'BASE range UN..TROIS;
subtype bornes2_1 is SINT'BASE range ICHI..SAN;
subtype bornes2_3 is SINT'BASE range SINT'FIRST..SINT'FIRST+2;
```

◇ L'étape (2) s'applique, le tableau étant multidimensionnel; on vérifie que tous les sous-agrégats ont les mêmes bornes et on lève CONSTRAINT_ERROR sinon :

```
-- seconde dimension : bornes2 est égal statiquement à 1..3
-- troisième dimension: bornes1_2 est égal statiquement à 1..3, d'où
if bornes1'FIRST /= 1 or else bornes1'LAST /=3 or else
   bornes1_1'FIRST /= 1 or else bornes1_1'LAST /=3 or else
   bornes1_3'FIRST /= 1 or else bornes1_3'LAST /=3 or else
   bornes2_1'FIRST /= 1 or else bornes2_1'LAST /=3 or else
   bornes2_3'FIRST /= 1 or else bornes2_3'LAST /=3
then
   raise CONSTRAINT_ERROR;
end if;
```

◇ L'étape (3) détermine le sous-type résultant :

```
subtype A_Anonyme is A(bornes, bornes1, bornes1_1);
```

◇ L'étape (4), si on n'a pas à faire à un tableau vide, vérifie que les bornes appartiennent bien aux sous-types d'indice :

```
if A_Anonyme'SIZE > 0 and then (
   bornes'FIRST notin 1..3 or else bornes'LAST notin 1..3 or else
   bornes1'FIRST notin SINT or else bornes1'LAST notin SINT or else
   bornes1_1'FIRST notin SINT or else bornes1_1'LAST notin SINT )
then
   raise CONSTRAINT_ERROR;
end if;
```

Certaines sous-expressions booléennes peuvent s'évaluer statiquement :

- Un tableau à n dimensions est parcouru par n boucles "for" imbriquées :

```
for i1 in temp'RANGE(1) loop
   for i2 in temp'RANGE(2) loop
      ...
   end loop;
end loop;
```

- Dans la boucle d'un agrégat nommé la sélection de l'association est faite assez naturellement à l'aide d'une instruction "case" s'il y a assez de choix, ou d'une cascade de "if" sinon; par exemple :

```
(1|5..8|12 => F, 3|9..11 => 0, others => G)
```

```
for i in temp'RANGE loop
   case i is
      when 1|5..8|12 => temp(i) :=F;
      when 3|9..11 => temp(i) := 0;
      when others => temp(i) := G;
   end case;
end loop;
```

ou, autrement :

```
for i in temp'RANGE loop
   if i = 1 or else i in 5..8 or else i=12 then
      temp(i) := F;
   elsif i=3 or else i in 9..11 then
      temp(i):=0;
   else
      temp(i):=G;
   end if;
end loop;
```

- Lorsque l'expression du composant est statique ou n'a pas d'effet de bord, et si elle est gardée par un choix "others" ou par un intervalle simple, alors on évalue cette expression une seule fois et on l'«étale» sur l'ensemble du tableau avec une instruction spécifique *array_init*. On évalue l'expression de ce composant «majoritaire» et on place sa valeur dans le premier composant du tableau en mémoire ; *array_init* recopie alors cette valeur dans l'ensemble du tableau. Ceci permet d'optimiser les cas très fréquents du style :

```
S: STRING(1..DISCR) := (1..DISCR =>' '); -- dans une définition d'article
A: MATRICE := (others =>(others=>0.0));
B: VECTEUR := (1|10 =>1, others => VECTEUR'LENGTH);
```

```
if A_Anonyme'SIZE > 0 and then (
    bornes1'LAST notin SINT or else
   bornes1_1'FIRST notin SINT or else bornes1_1'LAST notin SINT )
then
   raise CONSTRAINT_ERROR;
end if;
```

◇ L'étape (5) peut alors enfin créer un objet temporaire et l'initialiser avec les valeurs adéquates :

```
temp : A_Anonyme;
j,k:SINT;
if A_Anonyme'SIZE > 0 then
   for i in bornes loop
      if i=1 then
         j := bornes1'FIRST;
         for k in bornes1_1 loop
            temp(i,j,k):=F;
         end loop;
         J := bornes1'SUCC(j);
         for k in bornes2 loop
            if k=1 or else k=2 then
               temp(i,j,k) := F;
            elsif k=3 then
               temp(i,j,k):= 0;
            end if;
         end loop;
         J := bornes1'SUCC(j);
         for k in bornes1_3 loop
            temp(i,j,k) := F;
         end loop;
      else          -- choix "others"
         for j in bornes1 loop
            if j in 1..2 then    -- choix intervalle
               for k in bornes2 loop
                  temp(i,j,k) := F;
               end loop;
            elsif j=3 then
               k:=bornes2'FIRST; -- agrégat positionnel
               temp(i,j,k):=F;
               k:=bornes2'SUCC(k);
               temp(i,j,k):=UN;
               k:=bornes2'SUCC(k);
               temp(i,j,k):=F;
            end if;
         end loop;
      end if;
   end loop;
end if;
```

Ce morceau de simili-Ada illustre les transformations appliquées à un agrégat tableau non statique pour calculer sa valeur.

- Enfin il y a des cas où les boucles "for" ou les "case" peuvent être «déroulés» pour produire une succession d'affectations à l'instar de l'agrégat article. Ceci est assez flagrant sur notre exemple, où l'agrégat à évaluer n'avait finalement que 9 éléments en tout.

En conclusion, les agrégats tableau ne posent pas de problèmes particuliers au niveau de la machine Ada. Mais ils requièrent beaucoup de code au frontal pour tester leur légalité, et beaucoup de code dans le dorsal pour effectuer les transformations évoquées ci-dessus. Une première tentative dans Ada/ED avait été de considérer les tableaux multi-dimensionnels comme des tableaux de tableaux ; ceci constituait un modèle plus simple, mais ne conduisait pas à un traitement correct des agrégats multidimensionnels et des tableaux vides.

Chapitre 7.
Sous-programmes

Ce chapitre décrit la façon dont se déroulent les appels aux procédures et fonctions dans la Machine Ada. En particulier, on y introduit un mécanisme original pour l'accès aux objets non locaux : les *tables de relais*.

1. Position du problème

Les points délicats dans la sémantique dynamique d'Ada liés aux sous-programmes sont les suivants :

(1) Dans le texte-source d'un programme Ada, un appel à un sous-programme peut apparaître dès que la spécification du sous-programme a été compilée et donc avant que le corps correspondant ait été compilé.

(2) Les paramètres de type simple doivent être passés par copie, ceux de types composites peuvent être passés par référence ou par copie.

(3) Un paramètre effectif de mode "out" ou "in out" peut être une *conversion* appliquée à un objet, auquel cas le résultat doit être reconverti dans le type originel [LRM 6.4.1].

(4) Une fonction peut retourner une valeur d'un type composite non contraint. Cette valeur doit être recopiée au retour, quel que soit son type.

(5) La procédure appelée peut faire référence aux discriminants et aux bornes d'un paramètre de mode "out".

(6) L'appel à un sous-programme non encore élaboré doit lever une exception PROGRAM_ERROR, et de même pour l'activation d'une tâche dont le corps n'est pas élaboré. [LRM 11.1(7)].

(7) Ada est un langage à structure de blocs avec un lien *statique* strict entre les identificateurs et les objets qu'ils nomment, suivant la règle du bloc englobant *statique* le plus proche (lexicalement).

Le point (1) implique que l'objet procédure devrait être créé par l'élaboration de la spécification, à un point où toutes les caractéristiques du sous-programme ne sont pas encore connues, et notamment les objets non locaux auxquels il fait référence. Nous discuterons de ce point au §7-5. Pour le point 2, nous choisissons de passer les objets de type composite par référence ; ce n'est pas gênant pour le programmeur, et bien plus économique en mémoire, surtout sur un micro-ordinateur. De plus, cela va tout à fait dans le sens de notre choix d'avoir seulement des adresses dans la pile et les objets effectifs dans le tas. La conséquence du point (3) et de la décision précédente est que l'objet que l'on recopie au retour peut avoir une taille ou une structure différente de la variable dans laquelle on est supposé mettre le résultat final après conversion. Les conséquences de (5) sont qu'il se peut qu'on ne sache pas lors de la compilation quelle sera la taille et la structure de la valeur retournée et donc que la valeur retournée ne peut pas être traitée comme un simple paramètre de mode "out". Le point (6) indique bien que l'élaboration de sous-programme n'est pas une opération «neutre» à l'exécution. Quant au point (7), nous allons ici proposer un mécanisme pour le résoudre.

2. Références aux objets non locaux

La plupart des traducteurs pour les langages à structure de blocs utilisent le mécanisme des chaînages statiques ou dynamiques (*static or dynamic link*) ou des des vecteurs d'accès (*display*) pour accéder facilement aux objets non locaux au sous-programme en cours d'exécution [Dijsktra 68, Gries 71, Griffiths 76, Hill 76, Aho 78, Cunin 80]. Ici le problème est légèrement compliqué par le fait que les objets non locaux peuvent appartenir à une autre tâche, et donc ne figurent pas dans la même

pile. Bien que ces mécanismes de chainage ou de vecteurs d'accès puissent, avec quelques aménagements, être étendus au cas d'Ada, ils requièrent des modes d'adressages supplémentaires ou des registres spéciaux, ce qui compliquerait la conception du code d'ordre de la Machine Ada. Aussi avons-nous abordé le problème sous un tout autre angle.

Considérons qu'il y a trois classes d'objets dans un programme Ada :

a) Les objets *globaux* : tous les objets qui sont déclarés au niveau le plus externe, juste dans le paquetage STANDARD, c'est-à-dire : le programme principal (s'il est non réentrant), et tous les paquetages de bibliothèque. De plus nous allons promouvoir à ce niveau global toutes les constantes statiques, les patrons de type statiques, les patrons de types non statiques avant leur élaboration, et les sous-programmes statiques (cf. §7-4 ci-dessous).

Si leur taille est statique, on accède à ces objets globaux, situés dans les segments de données, par le mode d'adressage *global* avec une adresse absolue formée d'un couple (n° de segment, déplacement) déterminé à la compilation.

b) Les objets *locaux* : tous les objets déclarés immédiatement à l'intérieur du sous-programme ou corps de tâche, ou corps de paquetage. Ils sont logés dans le tas, et on y accède au moyen de pointeurs dans l'environnement d'appel courant. Les paramètres formels sont considérés classiquement comme des objets locaux.

On accède à ces objets locaux, et aux objets globaux de taille non statique, par le mode d'adressage *local*, avec un déplacement statique dans l'environnement d'appel (voir fig. 6, p.24).

c) Les objets *«semi-globaux»* : une façon simple de les définir serait de dire : «tous les objets qui ne sont ni globaux ni locaux»; ce sont des objets appartenants aux portées englobantes, hormis la portée la plus externe.

Dans la Machine Ada, *tous les objets semi-globaux référencés par un sous-programme lui sont passés en paramètre* comme s'il s'agissait de paramètres de mode "in out" passés par référence. Par conséquent, on y accède ensuite par le mode d'adressage local. Nous parvenons ainsi à n'avoir que deux modes d'adressage : global et local.

Cette notion de semi-globalité est transitive, comme le montre l'exemple suivant :

```
procedure P1 is
   A : UN_TYPE;
   procedure P2 is
      procedure P3 is
      begin
         A := ...; -- référence à A
      end P3;
   begin
      P3;
   end P2;
begin
   P2;
end P1;
```

A est semi-global à P3 qui y fait référence; il doit donc être passé comme paramètre à P3 par P2. Ceci en fait un objet semi-global référencé par P2, et donc P1 doit le passer en paramètre à P2. Enfin A est local à P1. En fait, ceci correspond à transformer le programme ci-dessus en :

```
procedure P1 is
   A : UN_TYPE;
   procedure P2 (X: in out UN_TYPE) is
      procedure P3 (Y: in out UN_TYPE) is
      begin
         Y := ...; -- référence à A
      end P3;
   begin
      P3(X);
   end P2;
begin
   P2(A);
end P1;
```

Examinons maintenant comment ce principe est mis en œuvre concrètement dans la Machine Ada.

Notons tout d'abord un point important : tous les mécanismes que nous allons décrire dans les paragraphes suivants pour des *sous-programmes* s'appliquent de la même manière aux *corps de tâches* et aux *paquetages de bibliothèque*.

Lors de la génération de code, on construit pour chaque sous-programme l'ensemble des objets semi-globaux auxquels il fait accès, soit directement, explicitement, soit indirectement parce que certains de ses sous-programmes internes y font référence. Nous appellerons cet ensemble l'*ensemble de relais* et, lors de l'exécution, nous installerons cet ensemble dans l'environnement d'appel sous forme d'une table : la *table de relais*, après l'avoir actualisée avec les adresses effectives des objets semi-globaux.

A première vue, ce calcul des adresses des éléments de la table de relais à passer en paramètre à chaque sous-programme peut sembler une tâche colossale. En fait des règles du langage font qu'il n'est nécessaire de calculer les adresses de cette table qu'une seule fois : lors de l'élaboration du sous-programme.

Dans la Machine Ada, les sous-programmes sont considérés comme des objets ayant un type, généralisant ainsi ce qui est fait pour les tâches, comme le suggère P. Hilfinger dans sa thèse [Hilfinger 83]. Par conséquent le mécanisme de patron de type leur est appliqué (cf. chap. 4). Un objet sous-programme contient les adresses globales effectives des objets semi-globaux auxquels le sous-programme est susceptible de faire référence. Un patron de sous-programme contient une table de relais constituée des adresses *locales* c'est-à-dire dans l'environnement d'appel du bloc dans lequel le sous-programme est élaboré; l'élaboration du sous-programme consiste alors à remplacer ces adresses locales par les adresses globales effectives des objets référencés. Un appel à un sous-programme fait référence à un objet sous-programme qui contient d'une part le numéro du segment de code et d'autre part la table de relais actualisée, table qui est recopiée dans l'environnement d'appel.

Ce mécanisme nous a été suggéré par R. Dewar. Il ne nous a pas été possible de le trouver mentionné dans la littérature, mais R. Dewar se souvient qu'un membre du W.G. 2.1 de l'IFIP lui en avait fait part il y a une dizaine d'années comme solution potentielle pour un compilateur Algol-68.

3. La table de relais

Il est important de ne construire que la table de relais minimale pour chaque sous-programme, afin de ne pas avoir à trainer des références inutiles.

Pour une procédure donnée p, soit S_p son ensemble de relais, E_p l'ensemble des sous-programme ou corps de tâche déclarés immédiatement dans p, R_p l'ensemble des objets auquel p fait référence, L_p l'ensemble des objets déclarés immédiatement dans p ; soit enfin G l'ensemble des objets globaux statiques. L'ensemble de relais S_p de p peut être défini comme :

$$S_p = \bigcup_{q \, in \, E_p} S_q - L_p + R_p - G \qquad (1)$$

Il y a quelques cas particuliers intéressants à signaler :

a) Si le sous-programme p est déclaré au niveau le plus exte e, c'est-à-dire si p est une unité de compilation, alors $S_p = \emptyset$. De plus le sous-programme p qui fait office de programme principal est non-réentrant, alors pour tout $q \in E_p$, on a $S_p = \emptyset$.

b) Si le sous-programme p est récursif, et n'est pas au niveau le plus externe, alors p in S_p.

De l'équation (1) ci-dessus, nous pouvons déduire l'algorithme utilisé pour construire la table de relais lors de la génération de code d'un sous-programme p, sans avoir à faire une passe supplémentaire sur le code intermédiaire :

a) On suppose au départ que l'ensemble de relais est vide : $S_p = \emptyset$.

b) Pendant la compilation de p, on tient à jour une table des objets locaux (y compris les patrons de types, les paramètres et les sous-programme

c) Chaque fois qu'il est fait référence à un objet qui n'est ni local, ni global, celui-ci est ajouté à la table de relais et est considéré dorénavant comme local.

d) Lorsqu'un sous-programme interne q in E_p est compilé, après avoir ajouté q à L_p, l'objet procédure qui contient S_q est construit en faisant

des références explicites à chacun de ses éléments, ce qui, par application du point c) ci-dessus, réalise l'opération :

$$S_{p,i+1} = S_{p,i} + (S_q - L_p) \qquad (2)$$

Bien que L_p puisse ne pas encore être complet au moment de la génération de code de *p*, les règles du langage sont telles que *p* ne peut pas faire référence à des objets déclarés plus loin que son propre corps [LRM 8.2(2)].

4. Elaboration des sous-programmes

L'élaboration d'une spécification de sous-programme consiste en la création de l'*objet* sous-programme. La table de relais du sous-programme peut ne pas être connue complètement à ce moment (le corps correspondant peut faire référence à des objets qui figurent plus loin et qui n'ont donc pas encore été élaborés), et par conséquent seul le numéro de segment de code est placé dans l'objet sous-programme, qui est marqué pour signaler que le corps correspondant n'a pas encore été élaboré. Cette élaboration est faite comme une élaboration de type (cf. §4-2) mais en utilisant un patron de sous-programme au lieu d'un patron de type.

Un sous-programme dont la table de relais est vide est considéré comme *statique*. L'élaboration d'un corps de sous-programme statique ne requiert aucune action (autre que de le marquer comme «élaboré»), et l'objet correspondant est global. Un sous-programme récursif *p* dont la table de relais ne contiendrait que *p* est aussi considéré comme statique.

L'élaboration du *corps* de sous-programme (ou de tâche) consiste à calculer dans l'environnement d'appel courant la table de relais effective à partir du patron de sous-programme et de stocker cette table effective dans l'objet créé lors de l'élaboration de la spécification. Ceci est réalisé par une instruction spéciale appelée *subprogram* avec le patron comme paramètre. L'objet qui en résulte est le sous-programme auquel il est fait référence par les instructions *call*. L'objet est marqué comme étant élaboré, et donc appelable. Il est intéressant de noter que ceci n'est possible que parce que les règles du langage font qu'il est interdit d'appeler un sous-programme (ou d'activer une tâche) avant que son corps n'ait été élaboré. Si ceci devait se produire (et il existe des cas où le compilateur ne peut le repérer

statiquement), l'instruction *call* détecte que l'objet est marqué «non élaboré» et lève l'exception PROGRAM_ERROR [LRM 11.1(7)].

Il faut enfin noter qu'il est indispensable de construire un nouvel objet sous-programme à chaque élaboration du sous-programme. Considérez le code suivant :

```
procedure P is
   X: INTEGER;
   procedure Q is
      Y: INTEGER: =X;  -- référence au semi-global X
   begin
   ...
   end Q;
begin
   ...
   P;  -- appel récursif à P
end P;
```

une nouvelle table de relais doit être construite pour Q à chaque exécution résursive de P, car X désigne un objet différent à chaque appel.

5. Tables de relais et compilation séparée

Ada offre plusieurs mécanismes permettant de faire de la compilation séparée, et en particulier celui des *corps souches* (ou *stubs* en anglais), qui permet de différer l'écriture d'un corps de sous-programme, de tâche ou de paquetage. En fonction de la stratégie de compilation adoptée, le mécanisme que nous venons de présenter peut éventuellement poser un problème majeur.

En effet, si un compilateur ne réalise la génération de code qu'après avoir «relié» toutes les unités de compilation requises par la désignation d'un sous-programme «principal», alors les corps proprement dits peuvent être inséré dans le code à leur place, et la construction de leur table de relais (ainsi que celle des sous-programmes englobants) pourrait avoir lieu à ce moment sans problème.

Mais si l'on souhaite effectuer la génération de code unité de compilation par unité de compilation, au moment même où elles sont soumises au compilateur, alors on ne sait pas construire la table de relais d'un sous-programme q pour lequel on n'a qu'une souche (*body stub*). Si q appartient à E_p, alors la table de relais de p ne peut pas non plus être

construite et ainsi de suite, transitivement. Aucune des deux solutions évidentes ne nous paraît satisfaisante :

- repousser la génération de code des unités qui contiennent des stubs jusqu'au moment où le corps vrai apparaît,
- ou décider arbitrairement que tous les objets non globaux visibles au point où le stub est déclaré font partie de sa table de relais, et donc de la table de relais du sous-programme englobant, etc.

Il nous a toutefois été possible de trouver une meilleure solution. Une fois encore nous allons donner des explications en termes de sous-programmes, mais rappelons que tout ceci est valable également pour les tâches et les paquetages de bibliothèque.

Soulignons tout d'abord quelques points :

(1) Les valeurs effectives des éléments de la table de relais ne sont pas utiles au moment de la compilation ; seule la taille de cette table serait utile, car ses éléments sont désormais des variables locales, et il nous faut connaître leur déplacement dans l'environnement d'appel pour générer le code qui y fait référence.

(2) Une unité de compilation qui n'est pas une sous-unité a une table de relais vide, puisque les seuls objets auxquels elle peut faire référence sont d'autres unités de compilation ou des entités déclarées dans la partie visible de paquetages de bibliothèque.

(3) Les *stubs* ne peuvent figurer que dans la partie déclarative la plus externe d'une unité de compilation [LRM 10.2(3)] ; il ne peuvent donc pas entrainer d'autres absences de table de relais.

Notre premier objectif est de rendre la génération de code possible sans connaître la table de relais ; or (1) signifie que le contenu de la table importe peu. Nous pouvons nous rendre indépendant de la taille de la table de relais en plaçant la table de relais au *sommet* de la pile, mais au-dessous par rapport au pointeur d'environnement d'appel, et en plaçant les autres variables locales au-dessus (voir figure 6, p.24). Par conséquent aucune adresse de variable locale ne dépend de la taille de la table de relais : simplement les éléments de la table de relais ont un déplacement négatif par rapport au registre de *Base de l'environnement d'appel* (SFP).

Puis il nous faut décider comment implémenter les patrons de sous-programme et les objets sous-programme pour les sous-programmes de bibliothèque et les sous-unités. Si un sous-programme n'est pas une sous-unité, il a une table de relais vide ; l'objet procédure est statique et est implanté en tête du segment de données associé à l'unité. Si le sous-programme est une sous-unité, alors l'objet sous-programme appartient à son unité mère, et il a été normalement créé lors de l'élaboration de la spécification.

Mais l'élaboration d'un objet sous-programme requiert un patron de sous-programme. Ce patron de sous-programme ne peut pas contenir la table de relais, puisque la sous-unité n'a pas encore été compilée (ou pas même été écrite). Le patron de sous-programme contient donc un champ : le *numéro de relais* qui est normalement à 0 lorsque le patron contient la table de relais, mais dans lequel on stocke le numéro de segment de code alloué au sous-programme dans le cas d'un *stub*. Nous savons que ce numéro est unique pour une unité de programme ; nous l'allouons donc à l'avance, et il nous servira à retrouver ultérieurement la table de relais.

Pour chaque unité de compilation contenant un stub le compilateur sauvegarde sa table de références locales dans la bibliothèque de programme : l'*environnement de stub*. Le point (3) ci-dessus garantit qu'il ne peut y avoir de tables de références locales imbriquées. Pour chaque sous-unité, on rajoutera à son environnement de stub sa propre table de relais (qui ne contient pas encore d'éventuels éléments hérités de sous-unités emboîtés).

Avec ces données dans la bibliothèque de programme, le relieur est désormais capable de construire les tables de relais manquantes. Ces tables sont placées dans le segment de donnée de l'«unité de reliure», par convention le segment n°1 (cf. chap.11). Puisque l'unité de reliure dépend de toutes les unités de compilation nécessaires pour un programme, toute recompilation d'une sous-unité susceptible d'affecter les tables de relais rendra périmée l'unité de reliure, donc forcera à recalculer les tables de relais lors de la prochaine activation du relieur.

Il subsiste un petit problème pour les paquetages dont le corps est séparé. Supposons le cas de figure suivant :

```
procedure MAIN is
   package PACK is
      procedure PROC;
   end PACK;
   package body PACK is separate;
begin
   PACK. PROC;
end MAIN;
```

A la fin de l'unité de compilation MAIN, nous traiterons en fait tous les sous-programmes dont nous avons la spécification mais pas le corps comme s'ils étaient explicitement séparés. Lorsque le corps proprement dit du paquetage sera compilé, deux cas peuvent se produire :

◇ Soit on y trouve le corps de PROC ; sa table de relais peut être calculée, mais on ne peut pas l'ajouter au patron de sous-programme, car nous sommes dans une autre unité. La table de relais est donc gardée «en attente» dans l'environnement de stub de l'unité de compilation et c'est comme plus haut le relieur qui se chargera de l'ajouter, à la différence près qu'il n'est plus besoin d'y apporter des modifications : il est complet, car il ne peut pas contenir de stub d'après le point (3).

◇ Soit le corps de PROC est lui-même séparé. Il ne se passe rien, on retombe dans le cas normal, le relieur trouvera sa table de relais comme pour toute sous-unité.

En conclusion nous avons un mécanisme qui nous permet de conserver une table de relais minimale, de générer immédiatement le code d'une unité de compilation, tout en gardant modeste le travail à réaliser par le relieur.

6. Discussion de l'efficacité du mécanisme de table de relais

Comparé aux mécanismes traditionels de vecteurs d'accès ou de chaînages, on est en droit de poser la question de l'*efficacité* du mécanisme de relais que nous avons proposé, dont la mise en œuvre semble à première vue bien plus lourde.

Considérons la finalité des paquetages d'Ada ; le Manuel de référence dit qu' «ils servent à spécifier des déclarations d'objets et de types mises en commun» [LRM 7.2] et également à «spécifier des groupes d'entités logiquement reliées, comprenant entre autres des sous-programmes qui peuvent être appelés de l'extérieur du paquetage» [LRM 7.2, 7.3(1)]. Ces considérations, confirmées par ce que nous avons pu constater dans l'usage effectif qu'il est fait par les programmeurs des paquetages de bibliothèque, tendent à réduire considérablement l'usage des objets semi-globaux si l'on compare avec un langage comme Pascal, par exemple. Les divers constituants d'un programme Ada tendent, grâce aux paquetages, à se partager des objets au niveau global, un peu à l'instar des *COMMONs* de Fortran, mais avec une meilleure encapsulation de ces objets. Tout ceci contribue en fait à rendre la structure générale des programmes Ada beaucoup plus plate, et donc les tables de relais seront la plupart du temps inexistantes ou très petites.

A la lumière des mécanismes d'encapsulation de données d'Ada, la plupart des arguments développés lors d'une récente polémique sur «Variables locales contre variables globales» sont caducs pour ce langage [FisherDL 83, Er 85], mais on peut en retenir une chose, c'est qu'il n'y a plus guère de place pour les variables semi-globales ; d'ailleurs plusieurs langages de programmation moderne n'offrent plus que le choix qu'entre local et global : C, BCPL, SETL et son fils spirituel B, PL/M-86.

Le coût de la recopie de la table de relais dans l'environnement d'appel est en fait négligeable, et il est contrebalancé par la simplicité du mode d'adressage local utilisé ultérieurement pour accéder aux objets semi-globaux.

Notons que les patrons de type étant des *objets* dans la Machine Ada, ils passent aussi par les tables de relais.

Enfin, et ce n'est pas un mince avantage, ce mécanisme offre le partage d'objets entre tâches emboîtées sans presqu'aucun sur-coût.

Concluons sur les ensembles de relais en donnant un exemple de programme Ada dont les sous-programmes et tâches sont «décorés» en commentaire de leurs ensembles de relais :

```
package PK is
   type PT is array(1..10) of INTEGER;
   PTAB: PT;
   procedure PK_PROC;  -- { } (ensemble vide, car niveau = 1)
end PK;

with PK; use PK;
procedure Q is  -- { } (ensemble vide, car niveau=1)
   type QT is record null; end record;
   REC: QT;
   procedure P1;
   procedure P2 is  -- { QT, P1 } (hérités du corps de tâche TT)
      task TT;
      U: INTEGER;
      task body TT is  -- { QT, P1, U }
         TREC: QT;
      begin
         U := U+1;
         P1;
      end TT;
   begin
      null;
   end P2;
   procedure P1 is  -- { REC, QT, P2 }
      PB: PT;
      REC1: QT;
   begin
      PTAB := PB;
      REC1 := REC;
      P2;
   end P1;
begin
   P1;
   P2;
end Q;
```

7. Appels de sous-programmes et retours

Récapitulons le mécanisme exact d'appel de sous-programme et de retour dans la Machine Ada.

L'instruction *call* a un paramètre qui désigne l'objet sous-programme. Celui-ci contient : la table de relais et sa taille, un booléen indiquant si le sous-programme a été élaboré, le numéro de segment de code qui contient le code effectif du sous-programme, le nombre d'objets locaux du sous-programme.

Si le sous-programme n'a pas été élaboré, l'exception PROGRAM_ERROR est levée au point de l'appel [LRM 11.1(7)] et l'instruction est abandonnée.

Un nouvel environnement d'appel est construit (cf. § 2-4) ; la table de relais y est recopiée, puis on y empile les données de liaison : adresse de

retour et Base d'environnement d'appel de l'appelant, le registre de Base d'environnement d'appel est alors chargé avec l'adresse courante du sommet de pile, puis de la place est réservée pour les objets locaux.

Enfin le contrôle est passé à la troisième instruction du segment de code du sous-programme appelé (qui sera très probablement un *enter_block*), sautant ainsi le traite-exception «piège» (cf. § 8-3).

L'instruction *return* n'est utilisée que pour une fonction, pour recopier le résultat de la fonction du sommet de la pile où il a été évalué jusque dans l'emplacement réservé sous les paramètres effectifs. Le retour effectif à l'environnement appelant se fait par une instruction *leave_block* après avoir attendu que les sous-tâches dépendantes soient terminées et après avoir libéré les ressources mémoires associées au bloc. Le registre d'environnement de bloc est alors restauré et, constatant que c'est le tout premier bloc de l'environnement, les registres d'environnement et de compteur programme sont restaurés avec les données de liaison.

Si l'instruction *return* n'est pas dans le bloc le plus externe du sous-programme, elle doit être précédée par un nombre adéquat de *leave_block*.

Pour le cas d'un corps de fonction qui retournerait à l'appelant sans passer par une instruction Ada "return", voir le §8-5.

Toute l'activité liée au passage de paramètres est réalisé par du code généré avant et après l'instruction *call*, dans ce que nous appellerons le prélude et le «post-lude» d'appel, qui se déroulent dans l'environnement de l'appelant.

8. Passage de paramètres

Il faut noter que tout ce que nous allons décrire pour des paramètres de sous-programmes est également valable à très peu de choses près pour des paramètres d'instructions "accept", utilisées pour les rendez-vous entre tâches (cf. §9-3 et [Rosen 86]).

Pour les paramètres de types scalaires (plus les types tâches) le langage impose un passage par valeur [LRM 6.2(6)]. Dans le prélude un objet est créé dans lequel la valeur "in" est copiée après évaluation. Pour les paramètres

"out" et "in out", deux pointeurs sont placés sur la pile : une référence au paramètre effectif, plus une référence à un objet temporaire dans lequel la valeur "in" est recopiée. Dans le postlude, pour les paramètres "out" ou "in out", le contenu de l'objet temporaire est recopié dans l'objet effectif, après une éventuelle conversion.

Pour les paramètres de type accès, c'est très similaire, mais dans le cas de contraintes différentes entre les objets désignés par le paramètre effectif et le paramètres formel, des tests sont insérés dans le prélude et le postlude pour qualifier l'objet désigné par la valeur accès [LRM 6.4.1].

Pour les paramètres de type composite, tableaux et articles, le langage autorise le passage par référence [LRM 6.2(7)], qui est moins propre du point de vue formel que le passage par valeur (problèmes dits de l'*aliasing*), mais nettement plus réaliste du point de vue des contraintes de temps et de mémoire. Cette solution du passsage par référence cadre parfaitement avec notre politique de gestion des piles et du tas, et permet de régler de façon simple le problème de l'accès aux bornes et discriminants des paramètres formels de mode "out" et "in out".

Un test d'index ou de discriminant est inséré dans le prélude si le paramètre formel est contraint et l'effectif ne l'est pas, ou bien si les deux sont contraints et qu'il n'est pas possible de déterminer statiquement si les contraintes sont compatibles [LRM 6.4.1].

Pour les tableaux, deux pointeurs sont passés dans la pile : un pointeur vers l'objet et un pointeur vers le patron. Pour les articles, comme ils contiennent leurs discriminants, il n'est pas besoin de passer d'autre information structurelle. Toutefois lorsqu'un article avec discriminant non contraint («mutable») est associé à un paramètre formel contraint, le passage ne peut pas se faire par une simple référence, car cela conduirait à une situation étrange, illustrée par le programme suivant :

```
type REC(D: INTEGER := 10) is record ...; end record;
subtype REC10 is REC(10);
U: REC;

procedure PU(K: out REC) is
begin
   if not K'CONSTRAINED then      -- comme le paramètre formel n'est
   then                           -- pas contraint, l'attribut déduit son
      raise NON_CONTRAINT;        -- résultat du paramètre effectif.
   end if;
end PU;
```

```
procedure PC(I: out REC10) is
begin
   PU(I);        -- le paramètre formel est contraint, quel que soit
end PC;          -- l'état du paramètre effectif.
   ...
PC(U);
```

Si le passage se faisait par référence, l'appel de procédure PC(U) lèverait l'exception NON_CONTRAINT, puisque le paramètre formel K, associé en fait à U, a son composant CONSTRAINED faux. Mais les règles du langage spédifient que la contrainte est celle du paramètre effectif, qui ici est I (dans PC), qui lui est contraint [LRM 6.2(9) et 3.7.4(3)]. Dans ce cas, et ce cas seulement, nous réaliserons le passage par valeur, en recopiant la valeur du paramètre effectif dans une variable temporaire contrainte. Cette recopie permet en même temps de faire le test de discriminant.

Notons que la solution naïve envisagée initialement, à savoir modifier le composant CONSTRAINED de U lors du prélude et du postlude de PC, n'est pas satisfaisante, car U doit conserver son état originel ; en effet, PU pourrait y accéder directement (par exemple en évaluant l'attribut U'CONSTRAINED).

Enfin, il ne devrait pas être trop gênant pour le programmeur que certains articles soient passés par valeur, car, comme dit le Manuel de référence : «l'exécution d'un programme est erronée si son effet dépend du mécanisme choisi par l'implémentation» [LRM 6.2(7)].

Ce cas spécial découle de notre choix initial d'inclure le champ *Constrained* dans l'article (cf. §4-3.12). Une autre solution serait de passer cette indication comme un paramètre supplémentaire, comme ceci :

```
type UREC(D: INTEGER := 0) is record ...; end record;
procedure P(A: in out UREC) is
begin
   if A'CONSTRAINED then ...; end if;
   A := UNE_VALEUR;
end P;
```

Que l'on pourrait transformer dans le compilateur en :

```
procedure P(A: in out UREC; A_CONTRAINT: in BOOLEAN) is
begin
   if A_CONTRAINT then ...; end if;
   if A_CONTRAINT and then A.D /= UNE_VALEUR.D
   then
      raise CONSTRAINT_ERROR;
   else
      A := UNE_VALEUR; -- sans test de discriminant
   end if;
end P;
```

9. Résultat d'une fonction

Finalement décrivons ce qui se passe pour la valeur retournée par une fonction. Pour les types simples, le résultat est laissé au sommet de la pile de l'appelant, afin de pouvoir être utilisé dans l'évaluation d'expression. Si le type composite du résultat est contraint, l'appelant crée un objet dans lequel la fonction appelée copie la valeur retournée. Si le résultat est d'un type composite non contraint, l'appelant ne peut pas créer un tel objet temporaire, dont il ne connaît ni la taille ni la structure. La fonction appelée, après évaluation de la valeur à retourner doit créer un objet temporaire de taille adéquate, mais doit lier cet objet au lien de données du bloc de l'appelant (cf. §2-7), sinon l'objet serait libéré par l'instruction *leave_block* qui suit normalement le *return*.

Ce problème est évoqué par Kamrad, pour l'Ada Language System et il propose une solution similaire : un type accès interne désignant le type retourné est créé et la fonction rend en pratique une valeur accès [Kamrad 83].

Chapitre 8.
Exceptions.

Ce chapitre décrit le mécanisme mis en œuvre dans la Machine Ada pour traiter les exceptions.

1. Position du problème

On considère généralement que la présence des exceptions ne doit rajouter aucune charge à l'exécution d'un programme tant qu'aucune exception n'est levée. La sémantique des exceptions est décrite dans le chapitre 11 du manuel de référence ; voici les points qui à notre avis sont les plus importants ou les plus délicats :

(1) Le «traite-exception» (*exception handler* en anglais) est associé au bloc (*frame*), et le traite-exception d'un bloc ne devient *actif* que lorsque l'exécution atteint le "begin" du bloc.

(2) Une exception est levée explicitement par une instruction "raise", ou implicitement par l'évaluation d'une expression par exemple [LRM 11.1].

(3) Lorsqu'une exception est levée, le contrôle est transféré au traite-exception le plus récemment activé (pour la tâche) qui soit capable de traiter l'exception.

(4) Un traite-exception peut re-lever la même exception, sans même savoir son nom.

La conséquence de (3) et des règles de dépendance des tâches est que lorsqu'une exception est levée et qu'il n'y a pas de trraite-exception pour la traiter dans le bloc courant, il faut dépiler l'environnement de bloc courant, après avoir attendu que les sous-tâches qui en dépendent soient terminées et après avoir libéré les ressources mémoire du bloc. Par conséquent, la prise en compte et le traitement d'une exception peut dans certains cas ne pas être aussi immédiat qu'on pourrait le penser à première vue ; la tâche peut être endormie plusieurs fois avant d'atteindre finalement le point où l'exception sera effectivement traitée.

2. Principes du traitement des exceptions

Un bloc en Ada a la forme suivante :

```
declare                                                    (1)
   déclarations
begin
   instructions
traite-exception
end;
```

où soit le *traite-exception* est absent, auquel cas il est équivalent à :

```
exception
   when others => raise;
```

soit il a pour forme :

```
exception
   when EXi | ... | EXj => action_A;
   when EXk | ... | EXl => action_B;
     ...
   when others => autre_action ;
```

Nous pouvons traduire ce code, dans le compilateur, en :

```
goto FIN_TRAITE_EXCEPTION;
<< DEBUT_TRAITE_EXCEPTION>>
case exception_courante is
   when EXi | ... | EXj => action_A;
   when EXk | ... | EXl => action_B;
     ...
   when others => autre_action ;
end case;
LEAVE_BLOCK;
raise;
<< FIN_TRAITE_EXCEPTION>>
```

Cette instruction "case" est très proche du code original, mais nous pensons que dans un traite-exception typique, il y aura soit très peu d'alternatives, soit l'ensemble des exceptions traitées sera un très petit sous-ensemble de l'ensemble de toutes les exceptions déclarées. Il est donc préférable de traduire ceci par une cascade de "if"... "elsif"... :

```
goto FIN_TRAITE_EXCEPTION;
<< DEBUT_TRAITE_EXCEPTION>>
if exception_courante = EXi
   or else...
   or else exception_courante = EXj
then
   action_A;
elsif exception_courante = EXk
   or else...
   or else exception_courante = EXl
then
   action_B;
elsif
   ....
else
   autre_action ;
end if;
LEAVE_BLOCK;
raise;
<< FIN_TRAITE_EXCEPTION>>
```

Maintenant notre bloc initial (1) peut se réécrire :

```
ENTER_BLOCK; -- declare
déclarations
ACTIVATE(DEBUT_TRAITE_EXCEPTION); -- begin
   instructions
traite-exception
LEAVE_BLOCK; -- end
```

et par conséquent un bloc sans traite-exception peut s'écrire :

```
ENTER_BLOCK; -- declare
déclarations
ACTIVATE(DEBUT_TRAITE_EXCEPTION); -- begin
   instructions
goto FIN_TRAITE_EXCEPTION;
<< DEBUT_TRAITE_EXCEPTION>>
LEAVE_BLOCK;
raise;
<< FIN_TRAITE_EXCEPTION>>
LEAVE_BLOCK; -- end
```

Lever une exception implicite ou explicite consiste à exécuter les actions suivantes : si le traite-exception du bloc courant a été activé, alors on le désactive et on exécute un **goto** DEBUT_TRAITE_EXCEPTION ; sinon, on exécute un **goto** FIN_TRAITE_EXCEPTION;

Mais nous pensons que la plupart des blocs n'ont pas de traite-exception, qui reste précisément exceptionnel, et nous avons souhaité factoriser toutes ces séquences

```
LEAVE_BLOCK;
raise;
```

d'un segment de code en un fragment de code que nous appellerons le traite-exception «piège». Pour simplifier l'activation de ce traite-exception piège et notamment lorsqu'une exception est levée par une instruction de la machine Ada ou le Noyau Exécutif, le piège est implanté en un endroit fixe, conventionnel : au début du segment de code. Ceci permet des optimisations intéressantes ; considérez le programme suivant :

```
BLOC1:
   begin
      instruction_1;
      BLOC2:
         declare
            déclaration_2 ;
         begin
            instruction_2 ;
         end BLOC2;
      BLOC3:
         declare
            déclaration_3 ;
         end BLOC3;
   end BLOC1;
```

Avec une transformation naïve, utilisant les équivalences données plus haut, on obtiendrait :

```
<<BLOC1>>
   ENTER_BLOCK;
   ACTIVATE(DEBUT_TRAITE-EXCEPTION_1);
   instruction_1;
<<BLOC2>>
   ENTER_BLOCK;
   déclaration_2;
   ACTIVATE(DEBUT_TRAITE-EXCEPTION_2);
   instruction_2;
   goto FIN_TRAITE_EXCEPTION_2;
<<DEBUT_TRAITE_EXCEPTION_2>>
```

```
      LEAVE_BLOCK;
      raise;
   <<FIN_TRAITE_EXCEPTION_2>>
      LEAVE_BLOCK, -- end BLOC2
   <<BLOC3>>
      ENTER_BLOCK;
      déclaration_3;
      ACTIVATE(DEBUT_TRAITE-EXCEPTION_3);
      instruction_3;
      goto FIN_TRAITE_EXCEPTION_3;
      LEAVE_BLOCK;
      raise;
   <<FIN_TRAITE_EXCEPTION_3>>
      LEAVE_BLOCK; -- end BLOC3
      goto FIN_TRAITE_EXEPTION_1
   <<DEBUT_TRAITE_EXCEPTION_1>>
      LEAVE_BLOCK;
      raise;
   <<FIN_TRAITE_EXCEPTION_1>>
      LEAVE_BLOCK; -- end BLOC1
```

qui en fait peut être replié sous la forme plus concise :

```
   <<TRAITE_EXCEPTION_PIEGE>>
      LEAVE_BLOCK;
      raise;
   <<BLOC1>>
      ENTER_BLOCK;
      instruction_1;
   <<BLOC2>>
      ENTER_BLOCK;
      déclaration_2;
      instruction_2;
      LEAVE_BLOCK; -- end BLOC2
   <<BLOC3>>
      ENTER_BLOCK;
      déclaration_3;
      instruction_3;
      LEAVE_BLOCK; -- end BLOC3
      LEAVE_BLOCK; -- end BLOC1
```

On note, en outre, que lorsqu'il n'y a pas de traite-exception, l'activation du traite-exception à la fin de la partie déclarative peut également disparaître, puisque par convention «pas de traite-exception» signifie par défaut le traite-exception piège. Voyons maintenant concrètement comment ceci est réalisé dans la Machine Ada.

2. Exceptions dans la Machine Ada

En interne, une exception est une valeur d'un type entier :

```
type EXCEPTION_TYPE is range 0..255;
NO_EXCEPTION: constant EXCEPTION_TYPE := 0;
```

La valeur 0 signifie par convention qu'il n'y a pas d'exception. Les valeurs 1 à 15 représentent les exceptions prédéfinies du langage :

- 1 à 5 les exceptions standard (CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, TASKING_ERROR) [LRM 11.1],
- 6 l'exception définie par notre implémentation SYSTEM_ERROR,
- 7 à 14 les exceptions prédéfinies dans le paquetage IO_EXCEPTIONS (STATUS_ERROR, MODE_ERROR, NAME_ERROR, USE_ERROR, DEVICE_ERROR, END_ERROR, DATA_ERROR, LAYOUT_ERROR) [LRM 14.5],
- 15 l'exception TIME_ERROR du paquetage prédéfini CALENDAR, [LRM 9.6],
- et enfin de 16 à 255 pour les exceptions définies par le programmeur.

L'affectation d'une valeur à une exception est faite par le compilateur globalement pour une bibliothèque de programme, par un mécanisme similaire à celui de l'affectation des numéro de segment de code aux unités de programme, ou aux segments de données aux unités de compilation. La raison en est que chaque exception doit être identifiée de manière unique dans un programme, car elle peut se propager en-dehors de la portée de son identificateur, vers d'autres tâches et d'autres unités de compilation.

Nota :

Dans le modèle en SETL de la machine, les exceptions prédéfinies du langage sont en plus «décorées» de leur cause exacte : ainsi pour Constraint_error, le programmeur saura s'il s'agit d'une "déréférence sur une valeur accès nulle", ou un "indice de tableau hors borne", etc. Ceci, ajouté au numéro de ligne où a été levée l'exception, s'est avéré extrêmement utile pour le programmeur Ada débutant, et répond partiellement aux critiques qui ont été faites du peu de «subtilité» des exceptions prédéfinies.

3. Traitement d'une exception

Le contexte de chaque tâche contient un registre d'exception dont le contenu est par défaut 0 (NO_EXCEPTION), et que l'on charge avec la valeur de l'exception avant de partir «en chasse» d'un traite-exception approprié. Le chargement de ce registre est fait soit par certaines instructions de la machine Ada qui détectent une situation d'exception (erreur de contrainte, accès avant élaboration, erreur numérique,...) soit explicitement par l'instruction *load_exception_register*.

Chaque environnement de bloc sur une pile contient un vecteur d'exception qui contient l'adresse du traite-exception couramment actif, c'est-à-dire le déplacement dans le segment de code courant du début du code du traite-exception. Le vecteur d'exception est à une position fixe par rapport au registre Base d'environnement de bloc.

Chaque segment de code (correspondant à une unité de programme) commence par un traite-exception piège formé des deux instructions *leave_block* et *raise* qui est utilisé dans deux circonstances :

- soit comme traite-exception par défaut lorsqu'il n'y en a pas d'autre actif dans l'environnement de bloc,
- soit pour les exceptions levées par le système, auquel cas seule l'instruction *raise* est utilisée.

Le déroulement du traitement est alors le suivant :

- *Pour activer le traite-exception :*

L'instruction *enter_block* construit un environnement de bloc dont le vecteur d'exception pointe sur le traite-exception piège. L'instruction *install_handler* a un paramètre qui est l'adresse du traite-exception effectivement attaché à ce bloc. Cette instruction est générée à la fin de la partie déclarative s'il y a un traite-exception. L'adresse est recopiée dans le vecteur d'exception de l'environnement de bloc courant, remplaçant ainsi le traite-exception piège par le traite-exception effectif.

- *Pour lever une exception :*

L'instruction *load_exception_register* a comme paramètre immédiat la valeur de l'exception. L'instruction *raise* va tenter de traiter l'exception

désignée par le registre d'exception. Cette instruction est donc utilisée également pour propager un exception dans un traite-exception.

Les instructions de la Machine Ada qui détectent une exception procèdent de la façon suivante : elles chargent le registre d'exception, puis effectuent le *raise* en positionnant le compteur programme à l'adresse 1, désignant ainsi comme prochaine instruction le *raise* du traite-exception piège.

- *Pour traiter une exception :*

L'instruction *raise* effectue les étapes suivantes : après avoir mémorisé le vecteur d'exception courant, elle le désactive en le remettant à zéro afin qu'il désigne à nouveau le traite-exception piège ; on évite ainsi les boucles au cas où une exception serait re-levée dans le traite-exception. Puis elle saute au traite-exception désigné par le vecteur d'exception. Si c'était le piège, on y trouvera l'instruction *leave_block* qui quittera le bloc courant (après avoir attendu la fin des sous-tâches et libéré les ressources mémoire), puis on trouvera une nouvelle instruction *raise*, et on sautera ainsi d'environnement de bloc en environnement de bloc jusqu'à trouver éventuellement un traite-exception actif. Un traite-exception actif est un morceau de code normal, qui se termine par une instruction *leave_block* et éventuellement par un *raise* si l'exception doit être propagée.

4. Exceptions et tâches

Une exception levée dans un rendez-vous doit être progagée dans la tâche appelée et dans la tâche appelante [LRM 11.5]. Ceci est traité par notre mécanisme en rajoutant un traite-exception spécial dans le corps de l'instruction "accept". Il n'est pas nécessaire de rajouter ce traite-exception lorsque le corps de l'"accept" est vide, car aucune exception ne peut alors y être levée. La transformation est la suivante :

```
accept ENTRY do
   instructions
end ENTRY;
```

devient :

```
accept ENTRY do
   begin
      instructions
   exception
      when others =>
         PROPAGER_VERS_L_APPELANT( exception_courante );
         raise;
   end;
end ENTRY;
```

Une exception levée pendant l'élaboration d'une partie déclarative ne peut être traitée que dans un bloc englobant, car le traite-exception n'est activé qu'à la fin de la partie déclarative. Mais il y a une exception pour une exception levée dans la partie déclarative d'une tâche (donc avant la fin de son activation) : il faut lever dans ce cas l'exception TASKING_ERROR dans la tâche-mère au point d'activation [LRM 9.3(3,7)]. D'autre part, si une exception est levée dans un corps de tâche, mais après la fin de l'activation, et qu'il n'y a pas de traite-exception, alors la tâche se termine, sans autre forme de procès. Pour réaliser ceci, on place dans un corps de tâche un traite-exception piège spécial, le *piège de tâche* dont le rôle est de terminer la tâche. Toutefois au début de l'activation, on fait désigner au vecteur d'exception un traite-exception spécial dont le rôle est de signaler à la tâche-mère l'échec de l'activation. Ainsi le corps de tâche suivant :

```
task body T is
   déclarations
begin
   instructions
 traite-exception
end T;
```

devient, après cette transformation :

```
task body T is
begin
   begin
      declare
         déclarations
         SIGNAL_END_ACTIVATION(OK);
      begin
         begin
            instructions
          traite-exception
         end;
      exception
         when others =>
            TERMINATE_TASK_QUIETLY;  -- piège de tâche
      end;
   exception
      when others =>
         SIGNAL_END_ACTIVATION(ERROR);
         raise;
   end;
exception
   when others =>
      TERMINATE_TASK_QUIETLY;  -- piège de tâche
end T;
```

En donnant une adresse conventionnelle fixe au piège de tâche (2), le code généré est relativement simple, similaire à ce que nous avons montré pour le piège ci-dessus.

Pour la tâche principale, le piège de tâche a pour effet de terminer inconditionnellement le programme. Quant à la tâche *principale*, qui a pour rôle d'élaborer les paquetages de bibliothèque, le piège de tâche en avorte l'exécution avec un message d'erreur approprié. Remarquez que bien qu'il n'y ait aucun moyen de l'atteindre, pour des raisons de simplicité, on a conservé un traite-exception piège dans les tâches *principale* et *de fond* au même emplacement que dans les autres tâches (cf. §9-6).

5. Cas des fonctions

D'après le Manuel de Référence, lorsqu'on quitte le corps d'une fonction autrement que par une instruction "return", il faut lever l'exception PROGRAM_ERROR [LRM 6.5(2)]. Mais, comme nous avons eu l'occasion de le relever dès la parution de la norme [DoD 83], il n'est pas indiqué très clairement *où* l'exception doit être levée. Pour le père du langage, J. D. Ichbiah, il est évident que l'exception est levée à la fin du corps, là où l'instruction "return" est absente. Mais d'autres spécialistes du langage comme J. G. Goodenough et G. A. Fisher, Jr., pensent que c'est au point d'appel, afin d'éviter le cas étrange où le corps de la fonction contiendrait un traite-exception pour PROGRAM_ERROR précisément. Ainsi dans l'appel à la fonction suivante :

```
function F return INTEGER is
   UNE_CONDITION: BOOLEAN := FALSE;
begin
   if UNE_CONDITION then
      return 5;  -- juste pour satisfaire l'analyse sémantique
   end if;
exception
   when PROGRAM_ERROR => raise UNE_AUTRE_ERREUR;
end F;
```

leur interprétation lèverait PROGRAM_ERROR, alors que pour Ichbiah ce serait UNE_AUTRE_ERREUR.

En attendant que le problème exégétique soit résolu par les sages, et pour pouvoir passer la suite de validation, nous avons adopté l'interprétation de Goodenough, qui nous a semblé assez bonne. Si donc la dernière instruction d'une fonction n'est pas un "return", le générateur de code émet du code pour désactiver le traite-exception courant, réactivant ainsi le piège, puis pour lever PROGRAM_ERROR. Le corps de la fonction ressemble à ceci :

```
<<TRAITE_EXCEPTION_PIEGE>>
   LEAVE_BLOCK;
   raise;
<<FONCTION>>
   ENTER_BLOCK;
   déclarations
   ACTIVATE_HANDLER(TRAITE_EXCEPTION);
   instructions
   goto <<FIN_TRAITE_EXCEPTION>>
<<DEBUT_TRAITE_EXCEPTION>>
   traite-exception
```

```
<<FIN_TRAITE_EXCEPTION>>
  ACTIVATE_HANDLER(PIEGE);
  raise PROGRAM_ERROR;
```

Si finalement l'interprétation de Jean Ichbiah l'emportait, il suffirait de retirer l'avant-dernière instruction qui réactive le piège.

6. Discussion

Pour un programme Ada sans traite-exception, et où aucune exception n'est levée, le sur-coût de notre mécanisme est minime :

- ◇ 2 instructions d'un octet par segment de code : le piège,
- ◇ 4 instructions d'un octet par tâche : le piège et le piège de tâche,
- ◇ 1 mot pour le vecteur d'exception dans chaque environnement de bloc,
- ◇ aucune exécution supplémentaire, sauf dans deux cas : lors d'une activation de tâche et dans le corps d'un "accept", où on exécute les 3 instructions *enter_block*, *install_handler* et *leave_block* (correspondant *grosso modo* aux "declare", "begin" et "end" du bloc fictif qu'on y rajoute).

Tous les auteurs qui décrivent une implémentation d'Ada fixent comme objectif de minimiser le sur-coût dû aux exceptions, mais très peu indiquent les mécanismes exacts mis en œuvre. La "Portable Virtual Ada Machine" de Ibsen semble avoir un vecteur d'exception associé à l'environnement de bloc comme nous le proposons, contenant par défaut une adresse de traite-exception fictif [Ibsen 84]. Knudsen fait une étude exhaustive des mécanismes de traitement des exceptions, mais commet une erreur majeure à notre avis en affirmant que le lien entre une exception et son nom est dynamique en Ada, extrapolant sur le fait qu'une exception peut se propager hors de la portée de son nom [Knudsen 84].

Chapitre 9.
Les tâches.

Ce chapitre récapitule sommairement la façon dont sont gérées les tâches dans la Machine Ada. Les mécanismes mis en œuvre ont été imaginés et réalisés par Jean-Pierre Rosen [Rosen 83] et sont développés dans sa thèse [Rosen 86].

1. Position du problème

Les points les plus délicats à réaliser sont à notre avis les suivants :

(1) Les tâches en Ada sont créées et terminées dynamiquement.

(2) Les tâches ont un *parent* et un *maître* [LRM 9.4]. La création et l'activation d'une tâche se fait en relation avec son parent, et la terminaison d'une tâche en relation avec son maître.

(3) L'existence de l'instruction "abort" complique considérablement le problème de la terminaison des tâches.

(4) Les rendez-vous peuvent être emboîtés.

(5) Les tâches sont sensibles à des événements externes (interruptions).

2. Création et activation des tâches

Les types tâche sont élaborés comme n'importe quel autre type de la machine Ada en complétant éventuellement un *patron de type tâche* (cf. §4-3.15). Les seuls éléments qui puissent éventuellement être non statiques dans un patron de type tâche sont les sous-type d'indice de familles d'entrée. Au corps de tâche correspond, comme pour toute autre unité de programme, un objet procédure ; son élaboration consiste éventuellement à calculer sa *table de relais* (cf. §7-3).

Une tâche est créée par l'élaboration d'une déclaration d'un objet de type tâche ou l'évaluation d'un allocateur pour un type accès désignant un type tâche (de même pour un type dont un *composant* est de type tâche). Dans le cas de l'élaboration d'une déclaration, le maître est le bloc contenant la partie déclarative, dans le cas de l'évaluation d'un allocateur, c'est la partie déclarative où le type accès a été élaboré.

La création d'une tâche consiste principalement en la création d'une pile de tâche, et l'initialisation de cette pile avec des données regroupées en un *bloc de contrôle de tâche*, et la création d'un *contexte de tâche*, c'est-à-dire l'ensemble des registres. Toutes les tâches ainsi créées et qui doivent être activées simultanément à la fin de la partie déclarative sont regroupées en une liste chaînée dont l'origine se trouve dans le tas, désignée par l'élément *tâches_déclarées* de l'environnement de bloc courant (cf. §2-4). C'est ce pointeur, sauvegardé dans une variable locale, qui permet de réunir des tâches déclarées dans une spécification de paquetage avec celles déclarées dans le corps pour leur activation simultanée. (Nous avons vu au §5-2.4 le cas particulier des tâches créées par les procédures internes d'initialisation.)

L'activation des tâches d'une liste de tâches déclarées consiste à les relier à leur maître, représenté par le couple (TP, BFP) : numéro de tâche et le pointeur d'environnement de bloc, et à leur autoriser l'accès à l'unité de traitement. A la fin de leur activation, elles signaleront à la tâche parente soit une exception TASKING_ERROR (cf. §8-4) soit leur succès.

3. Synchronisation entre tâches

La synchronisation entre tâches se fait au moyen d'un mécanisme de *rendez-vous*. Rosen a montré que les différentes sortes de rendez-vous [LRM 9.5-9.7] pouvaient se ramener par des transformations simples à deux formes : l'attente sélective temporisée et l'appel d'entrée temporisé [Rosen 83] ; il suffit pour cela d'introduire les notions de *délai infini* et de *délai nul*.

Le bloc de contrôle de tâche contient pour chaque entrée une file de tâches en attente.

Le mécanisme de passage de paramètres est identique à ce que nous avons décrit pour les sous-programmes (cf. §7-7). La seule différence est que les pointeurs de paramètres évalués sur la pile de l'appelant sont *recopiés* sur la pile de la tâche appelée. Il n'est pas nécessaire de faire de recopie au retour, les paramètres effectifs étant dans le tas. Toutefois l'appel d'entrée temporisé se comporte comme une fonction à résultat booléen et laisse sur la pile de l'appelant la valeur vrai si le rendez-vous a effectivement eu lieu, et faux sinon.

4. Terminaison des tâches

Pour quitter un environnement de bloc, il faut que toutes les tâches qui en dépendent (chaîne *Sous-tâches*) soient terminées. Il y a donc là une attente multiple, en même temps qu'une signalisation aux tâches «terminables».

Lorsqu'une tâche est terminée, l'espace qu'elle occupe : sa pile, est rendu au système. L'ordre dans lequel cette libération se déroule est assez délicat, car une tâche peut éventuellement s'avorter elle-même ou avorter son maître.

5. Principes de réalisation

Toute la réalisation au niveau microscopique est basé sur :

- trois *chaînages de tâches* : un pour les rendez-vous, un pour l'horloge et un pour les dépendances;

- la notion d'*État* d'une tâche,
- la notion d'*Événement*,
- et deux primitives : *Signal* et *Wait*.

La signalisation d'un événement à une ou plusieurs tâches destinataires produit, en fonction de leur état, une action, action dans laquelle l'une où l'autre des chaînes de tâches est parcourue et/ou modifiée.

L'ordonnancement des tâches est fait selon la technique des tourniquets, un par niveau de priorité. L'allocation de l'unité de traitement est faite, par défaut, aux seuls points de synchronisation. De façon optionnelle, on peut décider d'allouer des quanta de temps aux tâches.

Les interruptions sont des événements externes forçant un changement de contexte. Les sections critiques (notamment la primitive WAIT) sont protégées des interruptions par la primitive LOCK et son pendant UNLOCK ; elles peuvent être emboîtées, chaque tâche ayant un *niveau de verrouillage.*

6. Les tâche anonymes

Tout programme Ada dans la machine Ada comporte au moins deux tâches anonymes : la tâche *principale*, celle dans laquelle sont élaborés les unités de bibliothèques et qui appelle le sous-programme désigné comme «principal», et la tâche *de fond*, qui déclare la tâche principale et qui est activée (par le jeu d'une priorité inférieure à toutes les autres) lorsque toutes les autres tâches sont dormantes.

L'introduction de ces tâches anonymes est une notion très «unifiante» qui ramène un certain nombre de cas particulier au cas général. Cela permet par exemple de résoudre élégamment le problème de l'élaboration des unités de bibliothèque, notamment lorsqu'il s'agit de paquetages contenant des tâches, en donnant à ces unités un environnment et une tâche parente. Il y a une suggestion analogue dans l'ALS [Kamrad 83], mais nous avons trouvé pour notre part qu'il n'est pas possible de faire élaborer les unités de bibliothèques par la tâche de fond, car il se peut que celles-ci exécutent des intructions "delay", pendant lesquelles on n'aurait pas de tâche de fond à activer.

Voici esquissé en simili-Ada ce à quoi ressemblent ces deux tâches :

```
task TACHE_DE_FOND is
   pragma PRIORITY(0);
end TACHE_DE_FOND;
task body TACHE_DE_FOND is
   package STANDARD is
      ...
   end STANDARD;
   package SYSTEM is
      ...
   end SYSTEM;
   task TACHE_PRINCIPALE;
   task body TACHE_PRINCIPALE is
   begin
      declare
         -- ici déclaration des unités de bibliothèque
         package BIBLIOTHEQUE is ...
      begin
         SOUS_PROGRAMME_PRINCIPAL;
         STOP("Fin normale");
      exception
         when others =>
            STOP("Exception non traitée dans le programme principal");
      end;
   exception
      when others =>
         STOP("Exception dans l'élaboration d'une unité de bibliothèque");
   end TACHE_PRINCIPALE;
begin -- tache-de-fond
   loop
      null;
   end loop;
end TACHE_DE_FOND;
```

où STOP est une procédure qui arrête le programme en indiquant la raison de son arrêt.

Chapitre 10.
Jeu d'instructions

Nous terminons la présentation de la Machine Ada en passant en revue son jeu d'instructions.

1. Format des instructions

La Machine Ada a un jeu de 114 instructions. La plupart des instructions n'ont pas de paramètres immédiat dans le segment de code, mais trouvent leurs arguments sur la pile de la tâche courante, dans l'environnement de bloc, voire pour quelques rares instructions dans le bloc de contrôle de tâche ou les registres. Mais à certaines instructions, notamment les instructions arithmétiques et les instructions de transfert sont attachés :

◇ un *type simple de base* manipulé : *Byte, Word, Long, Dble* ou *XIng.* (cf. §3-2)

◇ un mode d'adressage : local, global, immédiat, spécial,... (cf. §2-8)

Tout combiné, ceci donne environ 261 codes opératoires.

Beaucoup d'instructions sont fort simples et similaires à ce qu'on trouve dans tout ordinateur : opérations arithmétiques et logiques entre un opérande au sommet de pile et un opérande en mémoire, transferts entre sommet de pile et mémoire, etc... D'autres sont à la fois plus sophistiquées et directement liée à Ada et aux mécanismes que nous avons décrits dans les

chapitres précédents : opérations d'élaboration des types et objets, de synchronisation entre tâches, etc...

La sémantique de chaque instruction est très précisément décrite par le code du prototype d'interprète en SETL. Cette description a l'énorme avantage d'être à la fois concise et lisible, mais également exécutable et donc vérifiable. Nous en donnerons des extraits en exemple dans les paragraphes suivants (les numéros dans la marge renvoient au source de l'interprète).

La mémoire est représentée par la séquence MEMORY, la topographie par RELOC, la pile courante par CUR_STACK, le code en cours d'exécution par CUR_CODE et on a défini les macro-instructions suivantes :

```
472   macro ADDR(bse,off);
          RELOC(bse)+off
      endm;

476   macro MEM(bse,off);
          MEMORY(ADDR(bse,off))
      endm;

484   macro POP(val);
          val frome CUR_STACK    -- fin de la pile courante (PUSH est similaire)
      endm;

514   macro GET_LOCAL_ADDR(BSE,OFF);
          SP := CUR_CODE(IP)+SFP;  -- numéro de variable en paramètre
          BSE:=CUR_STACK(SP);  -- mais la pile ne contient que l'adresse
          OFF:=CUR_STACK(SP+1);-- de l'objet, pas l'objet
          IP +:= 1;
      endm;
```

2. Instructions de gestion d'environnement

Enter_block et Leave_block créent et libèrent les environnements de bloc. A la fin de la partie déclarative, on trouve le cas échéant *Install_handler* qui met en place le vecteur d'exception (cf. §8-3) et/ou une activation des tâches de la chaîne des *tâches_déclarées*

Exemple : instruction *enter_block* (cf. fig.6, p.24)

```
2670  (i_enter_block) :
          PUSH(BFP);              -- sauvegarde de l'ancien environnement de bloc
          BFP := #CUR_STACK;      -- nouveau = sommet de la pile courante
          PUSH(0);                -- chaîne : lien de données
          PUSH(0);                -- chaîne: tâches déclarées
          PUSH(0);                -- chaîne: sous-tâches
2676      PUSH(1);                -- vecteur d'exception (désignant le piège)
```

3. Instructions d'élaboration

Les patrons de type sont élaborés avec *Type_local* et *Type_global*. *Subprogram* élabore un objet sous-programme, un corps de tâche ou de paquetage de bibliothèque. Les instructions *Allocate_xxx* réalisent les allocateurs ("new"), alors que les instructions *Create_xxx* gèrent le tas pour les objets locaux.

Exemple : instruction *i_create_b* : création d'une variable de type Byte

```
1416  (i_create_b) :
1417        CREATE(1, BSE, OFF, PTR); -- 1 unité de mémoire, résultat dans
                                      -- les registres invisibles BSE et OFF, adresse
                                      -- absolue dans registre invisible PTR
```

où CREATE est la macro :

```
537   macro CREATE(size, BSE, OFF, PTR);
          if THP + 2 + (size max 0) > MAX_MEM then -- THP = Top of Heap Pointer
              RAISE(STORAGE_ERROR,'Object creation');
          else  -- construction d'un morceau (cf. §2-7)
              MEMORY(THP +:= 1) := size+2;
              MEMORY(THP +:= 1) := BLOCK_FRAME(data_link); -- chaînage
              BLOCK_FRAME(data_link) := THP-1;
              PTR := THP+1;  -- calcul adresses absolue et virtuelle
              OFF := PTR - HEAP_ADDR;
              BSE := HEAP_BASE;
              THP := (size max 0);
          end if;
549   endm;
```

4. Instructions de sélection

Deref, Select, Subscript et *Array_slice* correspondent aux opérations de déréférence (ou indirection), à la sélection dans un article, à l'indexation et à la tranche. Elles font référence aux patrons de type pour effectuer leur calcul d'adresse et les tests de contrainte éventuels.

Exemple : instruction *i_deref_w* déréférence portant sur un Word

```
1508  (i_deref_w):
          POP_ADDR(BSE,OFF); -- dépile 2 éléments dans des registres invisibles
          if BSE=255 then
              RAISE(constraint_error,'Null access value'); -- macro RAISE
          else
              VALUE := MEM(BSE,OFF); -- lecture mémoire
              PUSH(VALUE);
1515      end if;
```

5. Instructions de synchronisation entre tâches

Activate et *End_activation* encadrent la phase d'activation, la tâche activée renvoyant la balle à la tâche parente. *Entry_call, Timed_entry_call, Selective_wait* et *End_rendezvous* réalisent les rendez-vous. *Wait, Abort* et *Terminate* implémentent le reste. (cf. description dans [Rosen 86])

6. Instructions de test

Qual_discr, Qual_index, Qual_range et *Qual_sub* réalisent les tests de contraintes. Elles ont deux paramètres : un objet sur la pile et un patron de type. Elles lèvent une exception en cas d'échec, laissent l'objet sur la pile en cas de succès. Par contre les comparaisons sont faites par les instructions *Compare*, dont le résultat composite laissé au sommet de pile est soit exploité par les *Is_equal, Is_greater,...* pour le transformer en booléen, soit par les *Jump_if_equal, Jump_if_greater,...* pour effectuer un branchement conditionnel.

Exemple : instruction *i_compare_w* portant sur deux mots en pile

```
1483  (i_compare_w):
          POP(VAL1);
          POP(VAL2);
          VALUE := ADA_BOOL(VAL1=VAL2) + 2*ADA_BOOL(VAL1<VAL2);
1487      PUSH(VALUE);

609   macro ADA_BOOL(X);
          (if X then 1 else 0 end)
611   endm;
```

7. Instructions de branchement

Outre les *Jump_xxx* que nous venons de mentionner, on a les instructions de boucle *For_Loop*, *Case* et *Raise*, dont la destination est au sein du même segment de code ; on a par contre *Call* qui fait référence à un objet sous-programme et construit un environnement d'appel, avant de changer de segment de code. L'instruction *Case* mérite un mot : elle est suivie d'une table de branchement «dense», formée de couples (borne inférieure de l'intervalle, adresse de destination) ordonnés par borne inférieure croissante ; la recherche de l'adresse de branchement se fait par recherche dichotomique dans cette table. [Atkinson 82, Bernstein 80]

Exemple : instruction *i_call_1* appel d'un sous-programme local

```
2756  (i_call_1):
          GET_LOCAL_ADDR(BSE,OFF); -- objet procédure
          PTR := ADDR(BSE, OFF);
          VALUE := MEMORY(PTR); -- numéro de segment de code
          if VALUE<0 then  -- accès avant élaboration
              RAISE(program_error,'Access before elaboration to ' +
                  GET_NAME(abs(VALUE),CODE_SLOTS));
          else
              OLD_CS := CS;
              CS := VALUE;
              CUR_CODE := CODE_SEGMENTS(CS);
          -- réserver de l'espace pour les variables locales
              VAL1 := CUR_CODE(#CUR_CODE);
              loop for i in [1..VAL1] do
                  PUSH(0);
              end loop;
          -- données de liaison
              PUSH(SFP);
              PUSH(OLD_CS);
              PUSH(IP);
          -- nouveau contexte
              SFP := #CUR_STACK+1;
              IP := 3; -- on évite le "piège"
          -- recopie de l'ensemble de relais (s'il existe)
              VAL2 := MEMORY(PTR+:=1)*2; -- longueur
              loop for i in [1..VAL2] do
                  PUSH(MEMORY(PTR +:= 1));
              end loop;
2787      end if;
```

8. Instructions arithmétiques, logiques et de transfert

Outre les instructions usuelles de cette catégorie (*Push, Pop, Add, Sub, Mul, Div,...*), on remarque : des opérations sur des tableaux de booléens, la concaténation et les copies d'objets structurés (qui font référence à des patrons de type), et quelques opérations portant sur des registres spéciaux : *Load_exception_register, Current_task, Statement* (pour la mise au point), etc...

Table des instructions

NAME	DATA MODE	ADDRESS MODE	OP CODE
nop	none	none	0
activate	none	none	7
activate_new	none	global, local	8...9
end_activation	none	immediate	92
selective_wait	none	immediate	234
raise_in_caller	none	none	218
end_rendezvous	none	none	99
entry_call	none	immediate	101
timed_entry_call	none	immediate	252
wait	none	none	258
abort	none	immediate	1
terminate	none	immediate	250
abs	byte, word, long, dble, xlng	none	2...6
neg	byte, word, long, dble, xlng	none	165...169
add	byte, word, long, dble, xlng	none	10...14
sub	byte, word, long, dble, xlng	none	243...247
mul	byte, word, long	none	162...164
div	byte, word, long	none	83...85
mod	byte, word, long	none	153...155
rem	byte, word, long	none	221...223
pow	byte, word, long	none	186...188
add_immediate	byte, word, long, dble, xlng	immediate	15...19
fix_mul	none	none	102
fix_div	none	none	103
float_neg	long, xlng	none	112...113
float_add	long, xlng	none	104...105
float_sub	long, xlng	none	116...117
float_mul	long, xlng	none	110...111
float_div	long, xlng	none	108...109
float_pow	long, xlng	none	114...115
not	none	none	170
and	none	none	20
or	none	none	171
xor	none	none	259
array_not	none	none	27
array_and	none	none	24
array_or	none	none	28
array_xor	none	none	30
array_catenate	none	none	25
allocate	none	none	21
allocate_copy	none	global, local	22...23
deallocate	none	none	64
declare	byte, word, addr, long, dble, xlng	local	65...70
create	byte, word, addr, long, dble, xlng	none	47...52
create_copy	byte, word, addr, long, dble, xlng	none	53...58
create_copy_struc	none	none	59
create_task	none	global, local	60...61
create_struc	none	none	62
uncreate	none	local	255
type_global	none	global	253
type_local	none	global	254
update	none	local	256

Table des instructions (suite)

NAME	DATA MODE	ADDRESS MODE	OP CODE
update_and_discard	none	local	257
discard	byte, word, addr, long, dble, xlng	none	77...82
duplicate	byte, word, addr, long, dble, xlng	none	86...91
subprogram	none	local	248
attribute	special	attribute	31
case	byte, word, long	none	35...37
start_for_loop	byte, word, long	code	235...238
start_forrev_loop	byte, word, long	code	239...241
end_for_loop	byte, word, long	code	93...95
end_forrev_loop	byte, word, long	code	96...98
call	none	global, local	32...33
call_predef	none	predefined	34
return	byte, word, addr, long, dble, xlng	local	225...230
return_struc	none	local	231
jump	none	code	142
jump_if_false	none	code	143
jump_if_greater	none	code	144
jump_if_greater_or_equal	none	code	145
jump_if_less	none	code	146
jump_if_less_or_equal	none	code	147
jump_if_true	none	code	148
raise	none	none	217
enter_block	none	none	100
leave_block	none	none	149
install_handler	none	code	136
link_tasks_declared	none	none	150
pop_tasks_declared	none	global, local	184...185
move	byte, word, addr, long, dble, xlng	none	156...161
array_move	none	none	26
record_move	none	global, local	219...220
indirect_move	byte, word, addr, long, dble, xlng	none	118...123
push	byte, word, addr, long, dble, xlng	global, local	189...200
push_immediate	byte, word, addr, long, dble, xlng	immediate	203...208
push_effective_address	none	global, local	201...202
pop	byte, word, addr, long, dble, xlng	global, local	172...183
indirect_pop	byte, word, addr, long, dble, xlng	global, local	124...135
compare	byte, word, addr, long, dble, xlng	none	38...43
compare_struc	none	none	44
float_compare	long, xlng	none	106...107
is_equal	none	none	137
is_less	none	none	138
is_greater	none	none	139
is_less_or_equal	none	none	140
is_greater_or_equal	none	none	141
membership	none	none	152
test_exception_register	none	exception	251
convert_to	none	global, local	45...46
qual_discr	none	global, local	209...210
qual_index	none	global, local	211...212
qual_range	none	global, local	213...214
qual_sub	none	global, local	215...216

Table des instructions (suite et fin)

NAME	DATA MODE	ADDRESS MODE	OP CODE
current_task	none	none	63
stmt	none	immediate	242
save_stack_pointer	none	local	232
restore_stack_pointer	none	local	224
load_exception_register	none	exception	151
deref	byte, word, addr, long, dble, xlng	none	71...76
select	none	immediate	233
subscript	none	none	249
array_slice	none	none	29
data	none	data	260
set_discr	none	none	235

Chapitre 11.
Le "Dorsal" d'Ada/ED et l'interprète

Ce chapitre décrit le dorsal (*back-end* en anglais) du compilateur Ada/ED et l'interprète qui simule et modélise la Machine Ada.

1. Le dorsal

Le dorsal est la partie du système Ada/ED dépendant de la machine. Il transforme la représentation intermédiaire AIS (Ada Intermediate Source) issue de l'analyse sémantique en code exécutable par la machine cible. Il est possible d'attacher différent dorsaux au même frontal, et c'est la technique que nous avons adoptée pour la mise au point de l'ensemble du système : le dorsal que nous décrirons ici se substitue exactement au dorsal initial, dit «de haut niveau», du système Ada/ED validé en Avril 83 [Kruchten 83].

Le dorsal que nous décrivons ici génère du code exécutable par la machine Ada que nous avons décrite dans les chapitres précédents. La génération s'effectue unité de compilation par unité de compilation, au fur et à mesure de leur soumission au compilateur. Le code produit n'a pas besoin d'édition des liens, car toutes les adresses sont relatives aux segments de code et aux segments de données associés respectivement aux unités de programme et aux unités de compilation. Un relieur très primitif sert à rassembler toutes les unité de compilation qui dépendent du sous-programme désigné comme programme principal et à compléter les informations manquantes dans le cas de *stubs* : leurs tables de relais (cf.§7-5).

Un tout premier prototype tentait de réaliser la génération de code directement à partir du langage intermédiaire produit par le frontal. Le décalage sémantique s'est avéré trop important et nous avons divisé le dorsal en deux passes entrelacées dans un même programme. Le dorsal consiste donc en trois parties :

- l'expanseur,
- le générateur de code proprement dit,
- le relieur.

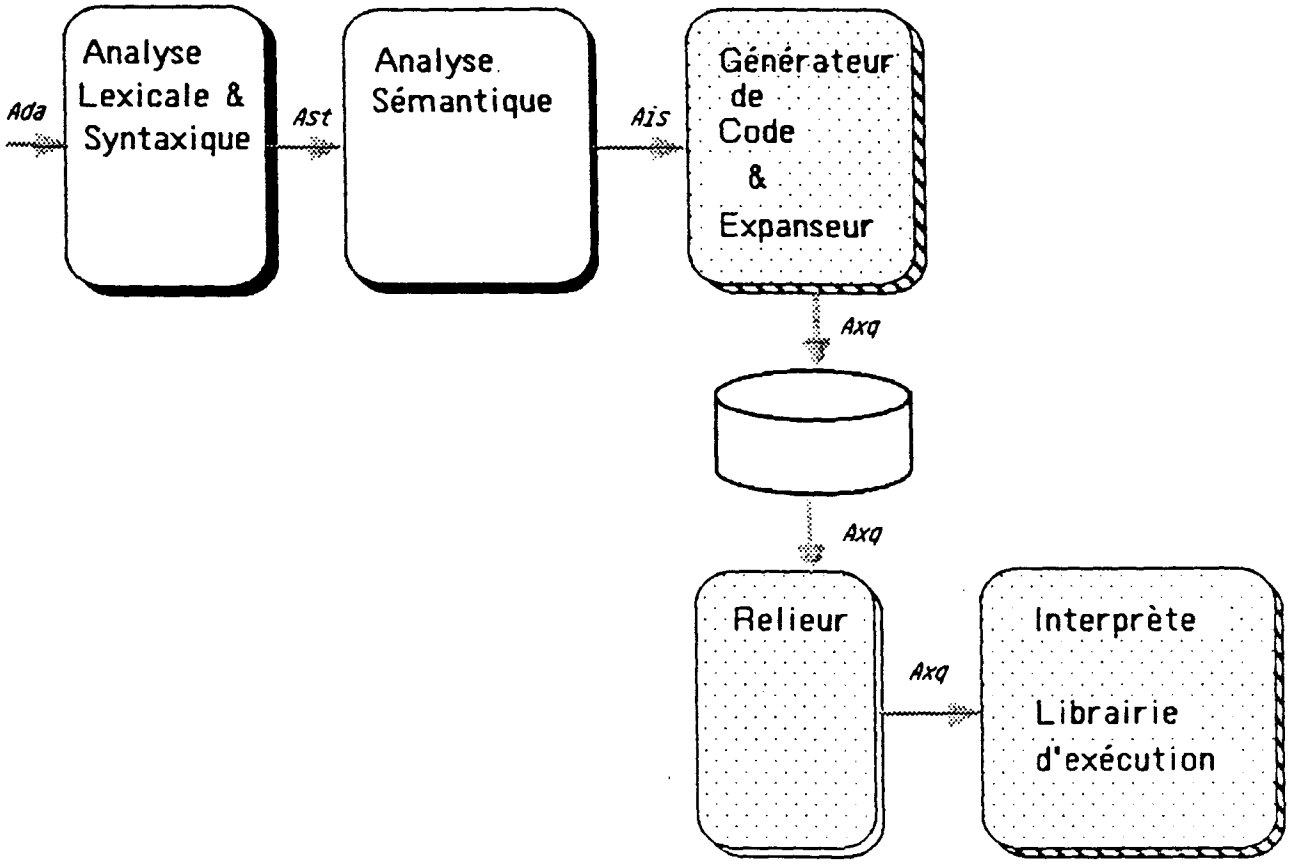


Figure 11 - Le dorsal et l'interprète

Plusieurs langages intermédiaires sont utilisés par le système Ada/ED qui sont plus ou moins des sous-ensembles les uns des autres.

- L'AST (Ada abstract Syntax Tree) entre l'analyse lexicale et l'analyse sémantique. Ils s'agit de séquences SETL imbriquées donnant une représentation «à la LISP» du programme source Ada (à ceci près que les parenthèses sont remplacées par des crochets carrés !).

- L'AIS (Ada Intermediate Source) issu de l'analyse sémantique ressemble à l'AST, mais tous les identificateurs et opérateurs ont été «désambigüés», et il font référence à une table des symboles associée à cet AST, sous-forme de relation SETL : (identificateur -> attributs). Les types dérivés ont disparu, et les instanciations génériques ont été «expansées». C'est à partir de l'AIS que travaille le dorsal.

- L'AXQ (Ada executable code) est produit par le générateur de code. Il contient les segments de code et les segments de données générés, avec un index des segments, plus les informations sur les *stubs* destinées au relieur.

Ce dorsal consiste en à peu près 10 000 lignes de source en SETL, dont 8 100 pour expanseur et générateur de code ensemble.

2. L'expanseur

L'expanseur reçoit du frontal un AIS, formé d'un AST et d'une table des symboles. La résolution des noms et des types a été faite, ainsi que l'expansion des génériques. L'expanseur prépare le travail du générateur de code en effectuant sur l'AST un certain nombre de transformations, dont la plupart ont été décrites dans les chapitres précédents.

(a) Construction des sous-programmes internes d'initialisation des objets de type composites, s'il l'un au moins des composants a une valeur initale par défaut, ou une valeur initale implicite. Cf. Chap.5.

(b) Construction de suites d'instructions pour l'évaluation des agrégats dont la valeur ne peut pas être calculée statiquement ; ceci comprend l'évaluation du sous-type de l'agrégat. Cf. Chap. 6.

(c) Insertion dans les déclarations d'objets des valeurs initiales implicites (pour les tâches et les types accès) ou des appels aux sous-programmes construits en (a).

(d) Calcul des expressions statiques (du moins celles qui n'ont pas été calculées par le frontal en application d'une des règles du LRM). Ceci est en fait plus une compression du code qu'une expansion !

(e) Synthèse d'attributs supplémentaires pour permettre des optimisations simples lors de la génération effective du code : *has_no_side_effect*

par exemple qui est un attribut d'un nom (au sens Ada) ou d'une expression, ou *tasks_declared* qui est un attribut d'une partie déclarative.

(f) Ajout aux parties déclaratives de la déclaration effective des variables de contrôle des boucles "for" et des sous-types utilisés dans les tests d'apprtenance "in", "notin" et les allocateurs "new".

(g) Extension du code d'un corps de tâche, pour y rajouter les traite-exception spéciaux : piège de tâche et traite-exception d'activation (cf. §8-4).

(h) Expansion des instructions de rendez-vous, suivant les équivalences de Rosen [Rosen 83].

(i) Expansion des traite-exception, suivant les transformations définies au §8-2.

(j) Transformation des renommages d'objets en déclarations d'objets de type accès et transformation des références à ces objets en déréférence par rapport à l'objet de type accès [LRM 8].

(k) Ajout de conversions et/ou qualifications explicites dans certains cas de passage de paramètres (cf. §7-7 et §10-6).

(l) Supression de qualifications et conversions dans des cas simples (tests d'intervalle et de discriminants).

(m) Suppression du code «mort» dans des cas simples.

Cette passe procède par une descente récursive dans l'AST, construisant de nouveaux nœuds dans l'arbre pendant la descente (expansion) et effectuant des simplifications, calculs d'expression statiques et synthèse d'attributs pendant la remontée. Le résultat est un arbre plus simple et souvent plus gros que l'arbre originel, dans lequel certains types de nœuds n'existent plus ('aggregate', 'renames', par exemple), et une table des symboles augmentée.

Il serait possible également de réaliser dans cette même passe l'expansion des sous-programme pour lesquels le programmeur a spécifié un pragma INLINE.

3. Le générateur de code

Le générateur de code reçoit de l'expanseur un AST simplifié et une table des symboles augmentée. Il produit un fichier AXQ qui contient :

- Un segment de donnée, tableau d'octets contenant toutes les données statiques de cette unité de compilation : constantes statiques, patrons de types, tables de relais etc.

- Un ensemble de segments de code.

- Une relation entre les numéros de segments de code et les noms uniques des sous-programmes qu'ils contiennent (pour l'aide à la mise au point).

- Une relation entre les numéros d'exception et leurs noms uniques (pour l'aide à la mise au point).

Le générateur de code est organisé en plusieurs modules :

(a) La *gestion des portées* : qui s'occupe des sous-programmes, blocs, corps de tâches, instructions "accept", paquetages etc. On y préserve l'environnement du bloc englobant : références locales, compteurs d'adresses locales,... et on calcule les ensembles de relais (cf. §7-3). On y trouve aussi les procédures qui génèrent les prélude et postlude d'appel de sous-programme et de rendez-vous (cf. §7-7).

(b) Le *traitement des types* : qui construit les patrons de types et l'élaboration des types (cf. §4-3). La plus grosse partie est liée à la linéarisation des *itype* des types articles, notamment en cas de variantes imbriquées (cf. §4-3.12).

(c) Les *évaluateurs* : qui génèrent le code pour le calcul d'adresse et le calcul des expressions. Un part importante est liée au calcul des agrégats (cf. Chap. 6).

(d) La *sélection de code* : processus piloté par des tables pour déterminer les codes opératoires à utilisés, en fonction des types de données de base (cf. §3-2) et des modes d'adressage (cf. §10-1).

Le générateur de code procède par descente récursive dans l'AST, orientant le travail sur les différents modules en fonction du type de nœud et détruisant l'arbre au fur et à mesure.

La table des symboles est complétée par plusieurs relations : la relation des références, la relation des tailles de type, et quelques attributs supplémentaires.

La relation des références donne pour chaque objet l'adresse qui lui a été affectée : adresse locale à l'environnement d'appel ou globale dans le segment de données de l'unité de compilation courante, selon le cas. Pour les types, il s'agit de l'adresse du patron de type, pour les sous-programme, l'adresse de la table de relais. Pour les noms des composants d'un article, il s'agit du déplacement au sein de l'article si celui-ci est connu statiquement. Pour le nom d'un paquetage, on trouve le nom de l'objet interne généré pour contenir la liste des tâches déclarées par la spécification du paquetage et qui ne devront être activées qu'à la fin de la partie déclarative du corps correspondant.

La relation des tailles de type contient pour chaque type ou sous-type sa taille en unité de mémoire, si elle est connue statiquement.

4. L'interprète

L'interprète consiste en un programme d'environ 11 000 lignes sources en SETL. Il est constitué de trois parties :

- l'interprète proprement dit, hormis l'implémentation du *tasking*, mais incluant le chargeur et des procédures d'aide à la mise au point,
- le Noyau Exécutif, réalisant le *tasking*, (cf. [Rosen 86])
- la librairie d'exécution, contenant le code des paquetages prédéfinis.

La structure de l'interprète est extrêmement simple ; en dehors de quelques déclarations représentant la structure de mémoire et les registres de la machine Ada, l'essentiel du code est une boucle contenant un grand "case" sur le code de l'instruction. La plupart des instructions sont implémentées en quelques lignes. Seules quelques opérations très spéciales, comme l'élaboration de type, font l'objet de procédures à part. On a utilisé abondamment le langage de représentation des données de SETL [Dewar 79], pour préciser la structure et le contenu des séquences et variables simples utilisées, et pour gagner en efficacité.

La librairie d'exécution a été reprise en grande partie du prototype Ada/ED de haut-niveau.

Conclusion

De nombreuses architectures de «machines-langage» ont été proposées ces vingt dernières années qui tentent d'intégrer au matériel des opérations spécifiques à un langage de programmation afin d'obtenir un meilleur compromis entre les phases de traduction et d'exécution [Lecussan 86]. Dans le cas des langages procéduraux à structure de bloc, ces machines ont une architecture relativement simple : P-machine de l'U.C.S.D , machine O.P.A. pour Pascal [Schulthess 82], le stade ultime dans cette direction étant les architectures "RISC" (Reduced Instruction Set Computer). Nous avons présenté dans cette thèse une architecture de machine adaptée à l'exécution de programmes écrits en Ada. Elle tente de réaliser un compromis entre une architecture très simple, où les instructions offrent des *primitives* très générales, et une "architecture orientée langage", où les instructions offrent plutôt des *solutions*.

Ce qui fait donc l'originalité de cette Machine Ada, c'est qu'elle intègre, au-dessus d'une structure assez classique : segments, piles et jeu d'instructions limité sur des types de base simples, des mécanismes spécifiques pour traiter les nombreux *liens de dépendance* entre entités Ada qui subsistent encore après la phase de traduction.

(a) Liens entre valeurs et types, en offrant une architecture partiellement «taggée» par les *patrons de type*.

(b) Liens de dépendance des tâches, des traite-exception et des variables dynamiques vis-à-vis d'un bloc.

(c) Liens entre environnements d'appel de sous-programme et de tâches par notre proposition d'un mécanisme de *tables de relais*.

Enfin la Machine Ada intègre des mécanismes tout à fait spécifiques à Ada pour la gestion des tâches : activation, synchronisation et terminaison, pour le traitement des exceptions (avec le traite-exception «piège») et pour l'élaboration des objets et types.

L'architecture proposée permet en outre, par le jeu d'une segmentation très stricte, une grande indépendance entre unités de compilation Ada et une implantation aisée sur de petites machines.

Nous avons écrit une spécification formelle, complète et exécutable de cette machine, accompagnée d'un générateur de code pour l'utiliser.

Le succès de ce projet est à mettre à l'actif d'une technique : le *prototypage rapide de logiciel*. Cette méthode commence à se tailler une place majeure parmi les outils du génie logiciel, pour réaliser des systèmes informatiques corrects et fiables [Budde 83, Kruchten 85]. Les méthodes de vérification formelle n'ont pas encore rempli leur promesses, du moins pas à l'échelle des systèmes industriels (ou de tout programme de plus de mille lignes de code) et les concepteurs ressentent de plus en plus le besoin de *spécifications exécutables*, c'est-à-dire de maquettes qui montrent les caractéristiques opérationnelles du système visé (sauf en terme d'efficacité, de performance) et qui puissent être produites en une toute petite fraction du temps et du coût du produit final. Ce besoin est tout particulièrement aigu lorsque le cahier des charges définissant le système est incomplet, susceptible d'être modifié, voire même d'évoluer en parallèle avec la réalisation. Dans ce dernier cas, il n'est plus question de formaliser le cahier des charges, vu qu'il n'y a souvent rien encore à formaliser et on a besoin d'une technique qui permette de faire des *esquisses* du système dont la fonctionnalité puisse être démontrée aux usagers potentiels et qui soient assez souples pour être modifiées très vite lorsque l'expression des besoins évolue. Des logiciels écrits dans des langages classiques (PL/I, Pascal, Algol etc.) n'offrent pas cette malléabilité ; on peut dire d'eux qu'ils sont résilients : si on tente de les plier, ils se cassent. Les langages de très haut niveau, surtout s'ils sont riches en primitives abstraites, semblent plus adaptés à la réalisation de tels prototypes ; c'est le cas de SETL, que nous présentons sommairement à l'annexe A.

La machine Ada a été esquissée dans ses grandes lignes en SETL, sans aucun souci initial d'efficacité, dans un style très abstrait, en parallèle avec son générateur de code, puis alors qu'elle commençait déjà à pouvoir exécuter des programmes élémentaires, nous avons pu profiter du vaste spectre sémantique offert par SETL pour expérimenter et choisir d'autres structures de donnée, plus efficaces, et pour «typer» certaines de ces structures de données, utilisant le sous-langage de représentation des données. Les «trous» ont été peu à peu comblés, soit dans un style très abstrait lorsque le temps manquait, reprenant éventuellement des éléments du premier prototype Ada/ED, soit au contraire dans un style très concret, utilisant un SETL de bas niveau. Du début à la fin de son écriture, le système a été exécutable et donc vérifiable. Les erreurs de conceptions apparaissent donc très vite et peuvent être rectifiées sans remises en cause dramatiques. Enfin le système final sert de spécification formelle pour la réalisation du système de production ultime. A ce stade, il n'y a plus d'incertitudes sur les algorithmes, sur les structures de données, sur la complétion du modèle, et le travail de réalisation avance très vite.

A partir du modèle en SETL que nous avons réalisé avec Jean-Pierre Rosen, la Machine Ada a été implantée avec succès sur processeur Motorola 68000 et validée au printemps 1986 avec la suite de l'A.C.V.C, ainsi que sur Intel 8086, pour lequel la validation est imminente. Un facteur de l'ordre de 400 a été gagné en temps d'exécution par rapport au prototype de haut-niveau, et sur un IBM PC, le système a à peu près les performances d'un bon interprète BASIC.

Notre machine Ada a également été portée avec succès sur un supercalculateur ELXSI System 6400 sous Enix.

Conformément aux objectifs initiaux, le système reste très ouvert et laisse le champ libre à diverses expérimentations. Par exemple dans le domaine de l'optimisation, et surtout de l'optimisation globale, en insérant une phase entre l'expanseur et le générateur de code, travail actuellement en cours à N.Y.U. Dans le domaine de la gestion des tâches, il est possible de remplacer le noyau exécutif par d'autres propositions, comme celle de R. Ogor de l'E.N.S.T de Bretagne par exemple [Ogor 85].

D'autres travaux sont actuellement en cours à N.Y.U. pour utiliser la machine Ada sur un "Ultracomputer", ordinateur à très haut degré de parallélisme de type MIMD à mémoire partagée ; les premiers résultats sont prometteurs [Schonberg 85]. Ce travail devrait faire l'objet du Ph. D. de Susan Flynn.

Au-delà de ces extensions et portages, nous espérons voir un jour tout ou une partie des mécanismes que nous avons proposés dans notre machine virtuelle intégrés à une autre machine Ada, plus concrète.

ANNEXES

Annexe A.
Le langage de programmation SETL

A.1 L'idée de départ

Parmi tous les formalismes mathématiques développés depuis le XIXème siècle, la *théorie des ensembles* reste toujours l'un des plus largement utilisé. La plupart des opérations mathématiques peuvent s'exprimer simplement et naturellement en utilisant la théorie des ensembles, et celle-ci repose sur une toute petite poignée de primitives, complétées par quelques mécanismes de définitions. Les notations et les concepts de la théorie des ensembles s'avèrent amplement suffisants pour décrire la plupart des algorithmes et structures de données dont on peut avoir besoin en programmation. Ils sont par ailleurs d'une généralité et d'une universalité qui les rendent utilisables dans un domaine très vaste.

Cette idée a donné le jour au langage de programmation SETL qui offre des variantes de ces primitives comme primitives de base du langage, en les restreignant aux *ensembles finis*.

Ce langage a été développé en plusieurs étapes au Courant Institute of Mathematical Sciences de New York University à partir de 1970 par une équipe dirigée par J. T. Schwartz et qui comprenait entre autres R. B. K. Dewar et E. Schonberg. L'implémentation utilisée actuellement est celle réalisée en 1978, date à laquelle on peut considérer que le langage s'est stabilisé [Dewar 79, Schonberg 81, Dewar 82].

A.2 Principales caractéristiques techniques du langage SETL

Le langage offre toutes les primitives de la théorie des ensembles finis sans autre restriction sur leur domaine. On trouve donc les opérateurs de la logique (et, ou, non, implication, équivalence), les quantificateurs existentiel et universel, avec leurs règles usuelles. Sur les ensembles on dispose de l'égalité et de l'appartenance. Il est possible de définir des ensembles à l'aide de constructeurs d'ensembles :

$$\{ e : x_1, x_2, \ldots, x_n \mid C \}$$

où e est une expression, les x_i une liste de variables avec l'indication de leur domaine et C un prédicat; ce constructeur désigne l'ensemble de toutes les valeurs que peut prendre e lorsque les x_i varient en satisfaisant le prédicat C. On peut définir des fonctions et prédicats et les utiliser en notation préfixée ou infixée, comme des opérateurs.

Ceci ne suffit pas encore à faire un langage de programmation ; on introduit donc des types de données de base : les entiers, les réels, les chaînes de caractères et les atomes, puis les notions de procédures et de fonctions avec les passages de paramètres usuels. Des primitives intéressantes ont été empruntées à d'autres langages: *string-matching, backtracking,* compilation séparée... Le langage est séquentiel, impératif, avec affectation, faiblement typé *a priori,* et avec allocation dynamique de mémoire, ce qui en fait un cousin de Lisp, APL et SNOBOL. Mais la lisibilité reste très supérieure à ces derniers : l'aspect général d'un programme SETL se rapproche plus d'Algol ou de Pascal.

Les types de données les plus importants en SETL sont les ensembles (*sets*), les séquences (ou n-uplets, ou *tuples*) et les relations (*maps*).

1) les ***ensembles*** de SETL ont toutes les propriétés usuelles définies en mathématique; ce sont des collections d'objets de type quelconque, pas nécessairement du même type, non-ordonnées et ne contenant pas deux fois le même objet. On peut y appliquer les opérations d'union, intersection, complémentaire, cardinal, ensemble des sous-ensembles etc... Nous avons vu plus haut les constructeurs d'ensembles. Il existe également des *itérateurs* sur ensembles qui sont des structures de contrôle du langage qui opèrent sur des ensembles:

```
{ x in S | C(x) }
```

représente le sous-ensemble de S dont les éléments satisfont le prédicat C(x); ou encore une boucle de recherche en utilisant le quantificateur existentiel:

```
exists X in C | C(X)
```

est une expression booléenne dont la valeur est vraie s'il existe un élément de S qui satisfait C(X); de plus X prend comme valeur un tel élément.

2) les ***séquences*** sont des suites ordonnées de longueur arbitraire, indexées par des entiers positifs, dont les composants, comme ceux des ensembles, sont de type quelconque et pas nécessairement homogène; il est possible de construire des ensembles de séquences, des séquences d'ensembles, des séquences de séquences d'ensembles etc... L'insertion et la suppression d'éléments à l'une ou l'autre extrémité d'une séquence permettent de les utiliser comme des piles ou des files d'attente. La concaténation de séquences les apparente aux listes. Des séquences hétérogènes de longueur fixe sont souvent utilisées en SETL là où dans d'autres langages on utilise des enregistrements ou articles. Les constructeurs de séquences et les itérateurs sur séquence sont similaires à ceux sur les ensembles. Par exemple:

```
[ x : x in [2..100] | ( not exists y in [2..x-1] | x mod y = 0 ) ]
```

construit la séquence des nombres premiers inférieurs à 100.

3) Les ***relations*** en SETL reproduisent fidèlement la notion de relation en théorie des ensembles: une relation est un ensemble de paires ordonnées, c'est à dire de séquences de longueur 2. Les premiers composants de ces séquences constituent le *domaine* de la relation et les seconds la *portée*. Domaine et portée d'une relation peuvent être quelconques: on peut construire des relations dont le domaine est un ensemble de relations, des relations de séquences à ensemble de chaînes de caractères, etc. Les relations peuvent être utilisées comme fonctions tabulées et l'application R(X), où R est une relation, peut soit s'évaluer comme une expression, soit être le membre de gauche d'une affectation. Enfin l'image d'un élément x dans une relation peut être unique (fonctions) ou être un ensemble

(correspondance); dans ce dernier cas c'est l'ensemble-image R{X} qui est évalué ou affecté; on parle alors de relation «multivaluée». Notons enfin que les séquences peuvent être considérées comme des relations dont le domaine serait les entiers naturels.

Lorsque la structure des données est connue, l'affectation peut être une opération très puissante de «déballage» de séquences:

```
[a, b, [c, -, -, g], h, [i, j], -] := T ;
```

Ces notions familières constituent le coeur de SETL. Mais ce qui distingue SETL d'une pure expression mathématique, c'est la contrainte d'exécutabilité, et pour garantir qu'un programme SETL soit toujours exécutable, seuls des objets finis peuvent être représentés et ils doivent être construits explicitement avant de pouvoir être utilisés ailleurs (pas d'évaluation paresseuse). Pour exprimer des notions d'objets infinis, on dispose des mécanismes habituels d'itération et de récursivité.

La simplicité conceptuelle de SETL s'appuie sur un environnement d'exécution complexe, capable d'exécuter toutes les primitives sur les ensembles, séquences et relations quels que soient les types de leurs composants. On peut dire que ces types de données sont des types de données abstraits pour lesquels le processeur SETL dispose d'implémentations complètes, libérant l'usager de la charge de les spécifier lui-même.

A.3 Un vaste spectre sémantique

Mais l'un des aspects les plus intéressants de SETL en tant qu'outil de prototypage est son vaste spectre sémantique; il y a en fait plusieurs langages dans le langage. Il est possible d'adopter un style très abstrait, n'utilisant que des contructeurs et itérateurs sur ensembles et séquences, des relations et des opérations logiques; le résultat est en général dense, d'aspect très formel, et souvent peu efficace. Il est possible d'utiliser des constructions plus proches de celles des langages de programmation classiques : le programme SETL de recherche des nombres premiers donné au §A.2 peut s'écrire avec des boucles `loop for ...; end loop;` contenant un `if ... then ... end if;` :

```
primes := {2..100};
loop forall x in [2..100] do
    loop forall p in primes | p < x do
        if x mod p = 0 then
            primes less:= x;
            quit;
        end if;
    end loop;
end loop;
```

Enfin il est possible de typer *a posteriori* un programme, en utilisant le sous-langage de spécifications des données [Dewar 79], et permettre ainsi à l'environnement d'exécution de choisir des représentations physiques en mémoire plus adaptées et des primitives de manipulations plus efficaces [Schonberg 81]. Enfin il est possible, après avoir spécifié plus finement le type des données de générer du code natif, augmentant ainsi considérablement la vitesse d'exécution du programme. Ce vaste spectre sémantique permet de raffiner un prototype logiciel en rendant sélectivement certains de ses composants plus efficaces, mais tout en restant dans le même cadre linguistique. Notre expérience avec les versions successives d'Ada/ED, décrite dans [Kruchten 85], a suivi cette approche.

A.5 Disponibilité de SETL.

SETL est disponible sur DEC VAX (sous UNIX ou VMS), ainsi que sur IBM (sous MVS et CMS), sur DEC 20 (sous TOPS-20), sur Amdahl (sous UTX) et sur des machines à base de MC68000 telles que le Sun Workstation. Le langage est partiellement compilé, partiellement interprété, avec la possibilité sur certaines machines de générer du code natif. Le système est écrit en LITTLE, un langage d'implémentation de système indépendant de la machine, conçu également à NYU.

A.6 Un exemple de programme SETL

La procédure SETL suivante est un prédicat (fonction à résultat booléen) qui indique si un graphe, représenté sous forme de relation de noeud à noeud, contient un cycle:

```
procedure has_cycle(graph);
    return exists subgraph in pow graph | subgraph /= {} and
        (forall [h1,t1] in subgraph | exists [h2,t2] in subgraph | t1=h2);
end procedure;
```

Nota: **pow** = power-set = ensemble des sous-ensembles d'un ensemble.

Annexe B.

Glossaire de termes propres à Ada et à la Machine Ada

Nous donnons ici des définitions des principaux termes utilisés dans ce document, avec le cas échéant leur traduction en anglais. Les termes précédés d'un astérisque sont des termes propres à la Machine Ada.

Accès (type accès). *(Anglais : Access type)* Une valeur d'un type accès (ou valeur accès) est soit la valeur "null", soit une valeur qui désigne un objet créé par un allocateur (primitive "new"). C'est ce qu'on appelle un pointeur ou une base dans d'autres langages.

Agrégat. L'évaluation d'un agrégat rend une valeur d'un type composé. Cette valeur est spécifiée en donnant la valeur de chacun de ses composants.

Allocateur. L'évaluation d'un allocateur crée un objet et retourne une nouvelle valeur accès qui désigne cet objet.

***Anonymes.** Les tâches anonymes sont la tâche de fond *(Idle task)* et la tâche principale *(Main task)*, créées par le relieur pour tout programme Ada.

***Bloc de Contrôle de Tâche.** *(Anglais : Task Control Block, T.C.B)* Ensemble d'informations liées à une tâche, situé au fond de sa pile. On y trouve des informations sur son état, et sur ses diverses entrées.

Cadre. *(Anglais : frame)* Un cadre est une construction de programme qui est soit une instruction bloc, soit un corps de sous-programme ou de paquetage.

Collection. Une collection est l'ensemble de tous les objets créés par l'évaluation d'allocateurs pour un type accès.

Contrainte. Une contrainte détermine un sous-ensemble des valeurs d'un type.

Corps souche. *(Anglais : stub)* Un corps souche est une forme de corps qui indique que le corps-vrai est défini par une sous-unité, compilée séparément.

Discriminant. Un discriminant est un composant particulier d'un objet ou d'une valeur de type article. Les sous-types d'autres composants ou même leur présence ou absence peuvent dépendre de la valeur du discriminant.

Elaboration. L'élaboration d'une déclaration est le processus par lequel la déclaration produit ses effets (comme par exemple la création d'un objet); ce processus se déroule lors de l'exécution du programme.

Entrée. *(Anglais: entry)* Une entrée est utilisée pour la communication entre tâches. Vue de l'extérieur, une entrée est appelée tout comme on appelle un sous-programme; son comportement interne est spécifié par une ou plusieurs instructions "accept" qui spécifient les actions à exécuter lorsque l'entrée est appelée.

***Environnement d'appel.** *(Anglais : Stack frame)* Ensemble d'informations dans la pile liées à l'activation d'un sous-programme, d'une instruction "accept" ou d'un corps de tâche. On y trouve notamment les informations de retour à l'appelant, les paramètres, la table de relais.

***Environnement de bloc.** *(Anglais : Block frame)* Ensemble d'informations dans la pile liées à un «cadre», dont l'instruction bloc est un exemple, d'où son nom. Il sert à accéder aux variables locales et on y trouve des liens vers les objets qui en dépendent, notamment les sous-tâches, des objets de collection et un éventuel traite-exception.

Exception. Une exception est un cas d'erreur qui peut survenir lors de l'exécution du programme. Lever une exception, c'est abandonner l'exécution normale du programme de façon à signaler qu'une

erreur s'est produite. Une réponse à l'exception peut être spécifiée par un traite-exception.

***Expanseur.** *(Anglais : Expander)* Partie du compilateur située entre l'analyse sémantique et la génération de code et qui effectue des transformations sur la représentation interne d'un programme afin de la simplifier.

***Morceau.** *(Anglais : chunk)* Bloc d'unités de mémoire contiguës allouées en une fois dans un tas. Il est constitué d'une en-tête donnant sa taille et un lien de dépendance, et d'une zone de stockage utile contenant un objet Ada.

Mutation. Opération de changement de sous type d'un article avec discriminant avec valeurs par défaut, par affectation globale. (Défini dans [Ichbiah 79b, Dod 84]).

***Patron.** *(Anglais : template)* Un patron de type est une structure de donnée servant à caractériser un type lors de l'exécution d'un programme. On y trouve les informations nécessaires à la création d'objets de ce type, à la sélection de composant, et la qualification des valeurs du type.

***Piège.** *(Anglais : catch-all)* Traite-exception particulier permettant de traiter les exceptions pour n'importe quel cadre en l'absence de traite-exception.

Pragma. Un pragma fournit des informations au compilateur.

***Prélude, *postlude.** Activités liées au passage des paramètres respectivement avant et après l'appel d'un sous-programme ou l'appel d'une entrée.

***Relais.** L'ensemble de relais *(Anglais : Relay-set)* associé à un sous-programme, à un corps de tâche ou à une instruction "accept" est l'ensemble des références aux objets non locaux et non globaux, auquel il fait référence directement ou indirectement.

***Relieur.** *(Anglais : binder)* Partie du compilateur Ada qui extrait d'une bibliothèque de programmes un ensemble cohérent d'unités de compilation et en forme un programme exécutable.

Rendez-vous. Un rendez-vous est l'interaction qui se produit entre deux tâches parallèles quand une tâche a appelé une entrée de l'autre tâche et qu'une instruction "accept" correspondante est exécutée par l'autre tâche pour le compte de la tâche appelante.

***Topographie de mémoire.** *(Anglais : Memory map)* Tables permettant la conversion entre des adresses virtuelles formées d'un numéro de segment et d'un déplacement dans ce segment et une adresse absolue en mémoire.

Traite-exception. *(Anglais : exception handler)* Un traite-exception est une partie du texte du programme spécifiant une réponse à l'exception. Exécuter un tel texte s'appelle traiter l'exception.

Type. Un type caractérise à la fois un ensemble de valeurs et un ensemble d'opérations applicables à ces valeurs.

Vecteur. Le vecteur d'exception dans un environnement de bloc désigne le traite-exception courant, par défaut le piège.

Annexe C.

«Appendice F» du système Ada/ED

La définition du langage impose que «le manuel de référence de chaque implémentation comprenne un appendice (appelé Appendice F) qui décrit toutes les caractéristiques liées à l'implémentation.» [LRM F]. Voici l'appendice F du système Ada/ED, dans sa version pour DEC VAX sous VMS.

F. Caractéristiques liées à l'implémentation

(0) Limites de l'implémentation

Longueur maximale d'un identificateur : 120 caractères
Longueur maximale d'une ligne de texte source : 120 caractères
Longueur maximale d'un fichier de texte source : 32767 lignes

(1) La forme, les endroits autorisés, et l'effet de chaque pragma lié à l'implémentation

Ada/ED ne reconnaît pas de pragmas liés à l'implémentation. Les pragmas définis par le langage sont correctement reconnus et leur légalité est vérifiée, mais à part LIST et PRIORITY, ils n'ont aucun effet sur l'exécution du programme. Un message d'avertissement est généré pour indiquer que le pragma est ignoré par Ada/ED.

(2) Le nom et le type de chaque attribut lié à l'implémentation

Il n'y a aucun attribut lié à l'implémentation dans Ada/ED.

(3) La spécification du paquetage SYSTEM

```
package SYSTEM is

    type SEGMENT_TYPE is range 0..255;
    type OFFSET_TYPE  is range 0..32767;
    type ADDRESS is record
            SEGMENT : SEGMENT_TYPE := SEGMENT_TYPE'LAST;
            OFFSET  : OFFSET_TYPE  := OFFSET_TYPE'LAST;
        end record;

    type NAME is (ADA_ED);
    SYSTEM_NAME   : constant NAME := ADA_ED;

    STORAGE_UNIT : constant := 8;
    MEMORY_SIZE  : constant := 2**16 - 1;

-- Nombres nommés dépendant du système

    MIN_INT       : constant := -(2**30) - 1;
    MAX_INT       : constant := 2**30 - 1;
    MAX_DIGITS    : constant := 6;
    MAX_MANTISSA  : constant := 63;
    FINE_DELTA    : constant := 2.0**(-30);
    TICK          : constant := 0.01;

-- Autres déclarations dépendant du système

    subtype PRIORITY is INTEGER range 1..10;

    SYSTEM_ERROR : exception;

end SYSTEM;
```

(4) La liste de toutes les restrictions sur les clauses de représentation

Ada/ED ne supporte de clause de représentation que pour le *small* des réels à point-fixe, et un programme contenant une autre clause de représentation est considéré comme illégal.

Les *small* valides pour cette implémentation sont de la forme :

$$S = 2^p.5^q, \quad \text{avec p et q entiers et les conditions}$$
$$-9 \leq q \leq 9 \quad \text{et} \quad 2^{-30} \leq S \leq 2^{30}$$

Les attributs de représentation sont reconnus comme le demande la définition du langage.

(5) Les conventions utilisées pour tout nom introduit par l'implémentation dénotant des composants liés à l'implémentation

Ada/ED n'introduit aucun nom lié à l'implémentation. (Le composant booléen CONSTRAINED des articles avec discriminants mutables est lisible au moyen de l'attribut du langage CONSTRAINED.)

(6) L'interprétation des expressions qui apparaissent dans les clauses d'adresse, y compris celles pour les interruptions

Les expressions d'adresse sont supportées par Ada/ED. Le type ADDRESS défini dans le paquetage SYSTEM est un article avec deux composants. Le premier est le numéro de segment, le second le déplacement dans ce segment; tous deux sont des entiers positifs. Les clauses d'adresse portant sur des entrées de tâches (interruptions) ne sont pas supportées.

(7) Toute restriction sur les conversions sans vérification

Ada/ED reconnait les conversions sans vérification et vérifie leur validité. Toutefois tout programme qui exécute une conversion sans vérification est considéré comme erroné et lève l'exception PROGRAM_ERROR.

(8) Toute caractéristique, liée à l'implémentation, des paquetages d'entrées-sorties

(a) Les fichiers temporaires sont supportés. La convention de nommage de tels fichiers est la suivante : xhhmmss.TMP, avec

x : S = SEQUENTIAL_IO
D = DIRECT_IO
T = TEXT_IO

hh : heure de création

mm : minute de création

ss : seconde de création

(b) La suppression de fichier est supportée.

(c) Un seul fichier interne peut être associé à un fichier externe donné à un instant donné (pas d'accès multiples aux fichiers).

(d) les noms de fichiers utilisés par les procédures CREATE et OPEN sont des noms de fichier standard de VMS. La fonction FORM retourne la chaîne donnée en paramètre FORM lors de la création du fichier. Aucune caractéristique liée au système n'est associée à ce paramètre.

(e) 17 fichiers au maximum peuvent être ouverts simultanément lors de l'exécution d'un programme.

(f) Le fichier d'entrée standard par défaut peut être spécifié par le paramètre DATA de la commande ADA. Si un fichier est spécifié, il doit être possible de l'ouvrir au début de l'exécution du programme, sinon

l'exception PROGRAM_ERROR est levée. Si aucun fichier n'est spécifié, on utilise SYS$INPUT. Le fichier de sortie standard est SYS$OUTPUT.

(g) SEQUENTIAL_IO et DIRECT_IO supporte comme paramètre générique des types tableaux *contraints*, des types article sans discriminants ou bien dont les discriminants ont une valeur par défaut.

(h) Les entrées-sorties portant sur des types accès sont possibles mais l'usage de valeurs de type accès engendrées lors d'une autre exécution est erroné.

(i) La terminaison du programme principal provoque la fermeture de tous les fichiers ouverts et la destruction des fichiers temporaires.

(j) Le paquetage LOW_LEVEL_IO n'est pas implémenté.

(k) Une marque de fin de page est constituée d'un enregistrement de fichier contenant le seul caractère page suivante (ASCII.FF); l'effet de l'utilisation de ce caractère dans un fichier de données est indéfini.

Annexe D. Documents divers

a) Premier certificat de validation d'un compilateur Ada

Ada® Joint Program Office

UNITED STATES
DEPARTMENT OF DEFENSE

New York University

has successfully validated

Ada/Ed

according to ANSI/MIL-STD 1815A

Certificate No. 001

Date April 11, 1983

Robert F. Mathis

Robert F. Mathis PhD
Technical Director
Ada Joint Program Office

b) Annonce de la disponibilité du produit final (*In: Ada Letters* 6(5))

THE NYU ADA/Ed-C TRANSLATOR

A new version of the Ada/Ed system is available. Ada/Ed-C is a machine-independent interpreter, written in C, that supports full ANSI-Ada. It is considerably faster than its predecessor (x10 compilation rate, x400 execution rate on the average) and has the same friendly user interface and precise error reporting.

Ada/Ed-C is currently available on VAX running Unix BSD.4.2, on SUN workstations, and on the ELXSI 6400 running Enix. These three versions nave passed all ACVC tests (version 1.7) and have successfully undergone validation on April 21, 1986.

For further information regarding distribution and pricing of Ada/Ed for the above, write:

NYUADA project
New York University
251 Mercer Street
NY NY 10012
(212) 460-7482

A version of Ada/Ed for VAX/VMS also passes all ACVC tests (version 1.7) and has successfully undergone validation on June 10, 1986. This version is distributed through National Technical Information Service (NTIS).
Springfiled, Va 22161

A version of Ada/Ed for the IBM/PC is targeted for release in July 1986

Bibliographie

[Afnor 86] Afnor : *Manuel de référence du langage de programmation Ada*, Norme Française NF Z-65-700, 1986.

[Aho 77] A. V. AHO et J. D. ULLMAN : *Principles of Compiler Design*, Addison-Wesley (Reading, Mass.), 1977.

[Appelbe 82] B. APPELBE et G. DISMUKES : "An Operational Definition of Intermediate Code for Implementing a Portable Ada Compiler", *Proceedings of the AdaTEC Conference on Ada,* Arlington (Virginia) 6-8 Octobre 1982, ACM, 266-274.

[Atkinson 82] L. V. ATKINSON : "Optimizing Two-state Case Statements in Pascal", *Software-Practice and Experience* 12, 1982, 879-882.

[Barbacci 85] M. R. BARBACCI, W. H. MADDOX, T. D. NEWTON et R. G. STOCKTON : "The Ada+ Front-end and Code Generator", *Proceedings of the Ada International Conference,* Paris, 14-16 mai 1985, 343-354. (aussi dans *: Ada Letters* 5 (2), septembre/octobre 1985).

[Barnes 84] J. G. P. BARNES : *Programming in Ada*, International Computer Science Series, Addison-Wesley (Londres), 2ème édition, 1984.

[Bernstein 80] R. BERNSTEIN : *Producing good code for the Case statement,* Note technique, IBM Research (Yorktown Heights), 1980.

[Bishop 80] J. M. BISHOP : "Effective Machine Descriptors for Ada", *SigPLAN Notices* 15 (11), Novembre 1980, ACM, 235-242.

[Burke 82a] M. G. BURKE et G. A. FISHER, Jr. : "A Practical Method for Syntactic Error Diagnosis and Recovery", *Proceedings of the SIGPLAN'82 Conference on Compiler Construction,* Boston (Mass.), ACM, Juin 1982.

[Burke 82b] M. G. BURKE : *A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery,* Thèse de Doctorat, Courant Institute of Mathematical Sciences, New York University, Décembre 82.

[Cunin 80] P. Y. CUNIN, M. GRIFFITHS et J. VOIRON : *Comprendre la compilation,* Springer-Verlag (Berlin) 1980.

[Dewar 79] R. B. K. DEWAR, A. GRAND, S. C. LIU, J. T. SCHWARTZ et E. SCHONBERG : "Programming by Refinement, as Exemplified by the SETL Representation Sublanguage", *ACM Transaction on Programming Languages and Systems,* 1 (1), juillet 79, 27-49.

[Dewar 80] R. B. K. DEWAR, G. A. FISHER,Jr., E. SCHONBERG, R. FROELICH, S. BRYANT, C. F. GOSS et M. G. BURKE : "The NYU Ada Translator and Interpreter", *Proceedings of the Compsac'80 Conference,* Chicago (Illinois), IEEE, octobre 1980.

[Dewar 82] R. B. K. DEWAR, E. SCHONBERG et J. T. SCHWARTZ : *Higher Level Programming – An Introduction to the Programming Language SETL,* Courant Institute of Mathematical Sciences, New York University, 1982.

[Dijkstra 68] E. W. DIJKSTRA : "Cooperating Sequential Processes", p. 43-112 *in: Programming Languages,* F. Genuys (éd.), Academic Press, New York, 1968.

[DoD 78] *Department of Defense Requirements for High Order Computer Programming Languages – "STEELMAN"*, Defense Advanced Research Projects Agency, Arlington, Virginia, Juin 1978.

[DoD 80] United States Department of Defense : *Reference Manual for the Ada Programming Language,* Proposed Standard Document, juillet 1980.

[DoD 82] United States Department of Defense : *Reference Manual for the Ada Programming Language,* Revised Proposed ANSI Standard Document, juillet 1982.

[DoD 83] United States Department of Defense : *Reference Manual for the Ada Programming Language,* ANSI/MIL-STD-1815A, janvier 1983 (cf. [Afnor 86] pour une traduction en français).

[DoD 84] United States Department of Defense : *Rationale for the Design of the Ada Programming Language,* Draft for editorial review, Honeywell et Alsys, janvier 1984.

[Donahue 85] J. DONAHUE et A. DEMERS : "Data Types are Values", *ACM Transactions on Programming Languages and Systems* 7 (3), Juillet 1985, 426-445.

[Er 85] M. C. ER : "Practical Considerations of Global and Local Variables", *Software-practice and Experience* 15 (5), Mai 1985, 499-502.

[Falis 82] E. FALIS : "Design and Implementation in Ada of a Run-Time Task Supervisor", *Proceedings of the AdaTEC Conference on Ada,* Arlington (Virginia) 6-8 Octobre 1982, ACM, 1-9.

[Fisher 83] G. A. FISHER, Jr. : *Spécifications internes du second compilateur Ada,* Telesoft, Inc. San Diego (Ca.), 1983 (communication privée confidentielle).

[FisherDL 83] D. L. FISHER : "Global variables versus Local variables", *Software-Practice and Experience* 13 (5), Mai 1983, 467-469.

[Goodenough 81] J. B. GOODENOUGH : "The Ada Compiler Validation Capability", *IEEE Computer*, juin 1981, 57-64.

[Goos 83] G. GOOS et alii : *An Optimizing Ada Compiler*, Institut für Informatik, Université de Karlsruhe, 1983.

[Gries 71] D. GRIES : *Compiler Construction for Digital Computers,* John Wiley & Sons, New York, 1971.

[Griffiths 76] M. GRIFFITHS : "Run-Time Storage Management", *in: Compiler Construction – An Advanced Course,* J. Eickel (éd.), Springer-Verlag, Berlin, 1976.

[Gupta 85] R. GUPTA et M. L. SOFFA : "The Efficiency of Storage Management Schemes for Ada Programs", *SIGPLAN Notices* 20 (11), novembre 1985, ACM, 30-38.

[Henessy 82] J. L. HENESSY et N. MENDELSOHN : "Compilation of the Pascal Case Statement", *Software–Practice and Experience,* 12, 1982, 879-882.

[Hilfinger 83] P. N. HILFINGER : *Abstraction Mechanism and Language Design,* The M.I.T. Press, Cambridge (Mass.), 1983.

[Hill 76] U. HILL : "Special Run-Time Organization Techniques for Algol 68", *in: Compiler Construction –An Advanced Course,* J. Eickel (éd.), Springer-Verlag, Berlin, 1976.

[Hill 83] D. D. HILL : "An Analysis of C-Machine Support for Other Block-Structured Languages" *ACM Computer Architecture News* 11 (4), septembre 1983, 6-16.

[Hisgen 80] A. HISGEN, D. A. LAMB, J. ROSENBERG et M. SHERMAN : "A Run-Time Representation for Ada Variables and Types", *SigPLAN Notices* 15 (11), Novembre 1980, ACM, 82-90.

[Ibsen 83] L. IBSEN, L. O. K. NIELSEN et N. M. JØRGENSEN : *A-Machine Specification,* Rapport interne n°ADA/RFM/0001, Christian Rovsing A/S, Ballerup (Danemark), mars 1983.

[Ibsen 84] L. IBSEN "A Portable Virtual Machine for Ada", *Software–Practice and Experience* 14 (1), janvier 1984, 17-29.

[Ichbiah 79a] J. D. ICHBIAH : "Preliminary Ada Reference Manual", *SigPLAN Notices* 14 (6), juin 1979, ACM, partie A.

[Ichbiah 79b] J. D. ICHBIAH , J. G. P. BARNES, J. C. HELIARD, B. KRIEG-BRÜCKNER, O. ROUBINE et B. A. WICHMAN : "Rationale for the Design of the Ada Programming Language", *SigPLAN Notices* 14 (6), juin 1979, ACM, partie B.

[Kamrad 83] J. M. KAMRAD : "Run-Time Organization for the Ada Language System Programs", *Proceedings of the Ada-Europe/AdaTEC Joint Conference on Ada,* Bruxelles, 16-17 mars 1983, 10(1-23).

[Katwijk 84] J. van KATWIJK et J. van SOMEREN : "The Doublet Model : Run-Time Model and Implementation of Ada Types", *SigPLAN Notices* 19 (1), janvier 84, ACM, 78-92.

[Kearns 83] J. P. KEARNS et M. L. SOFFA : "The implementation of recursion in a coroutine environment", *Acta Informatica* 19 (), 1983, 221-233.

[Kruchten 78] Ph. KRUCHTEN : "Quelques propositions pour l'architecture du processeur rapide de la machine ARCADE", Rapport interne projet Arcade, E. N. S. T., Juillet 1978.

[Kruchten 83] Ph. KRUCHTEN et E. SCHONBERG : "The Ada/ED system: a large scale experiment in Software Prototyping using SETL", *in: Approaches to Prototyping,* R. Budde (éd.), Springer-Verlag, 1983.

[Kruchten 84a] Ph. KRUCHTEN et E. SCHONBERG : "Le système Ada/ED: une expérience de prototype utilisant le langage SETL", *Technique et Science Informatiques,* 3 (3), 1984, 193-200

[Kruchten 84b] Ph. KRUCHTEN, E. SCHONBERG et J. T. SCHWARTZ : "Software prototyping using the SETL Programming Language", *IEEE Software,* 1 (4), 1984, 66-75.

[Kruchten 85] Ph. KRUCHTEN : "Le langage de programmation SETL et son utilisation pour la réalisation de prototypes de logiciels", *BIGRE + Globule,* **43/44**, juillet 85.

[Knudsen 84] J. L. KNUDSEN : "Exception Handling – A Static Approach", *Software–Practice and Experience* **14** (5), mai 1984, 429-449.

[Lahtinen 82] P. LAHTINEN : "A Machine Architecture for Ada", *Ada Letters* 2 (2), septembre-octobre 1982, 28-33.

[Lecussan 86] B. LECUSSAN : "Réflexions à propos des machines-langage", *Bigre+Globule*, **50**, septembre 86, 2-13.

[Myers 78] G. J. MYERS : *Advances in Computer Architecture*, Wiley & Sons (New-York) 1978.

[NYU 83a] NYUADA (nom collectif) : *Semantic Actions for Ada,* Rapport technique n°84, Courant Institute of Mathematical Sciences, New York University, 1983.

[NYU 83b] NYUADA (nom collectif) : *An Executable Semantic Model of Ada,* Rapport technique n°85, Courant Institute of Mathematical Sciences, New York University, 1983.

[Nowitz 82] D. A. NOWITZ : *The Ada Breadboard Compiler : Code Generation,* Mémorandum interne n°TM82-45412-13, AT&T Bell Laboratories, Murray Hill (N.J.), 1982.

[Ogor 85] R. OGOR : *Spécification opérationnelle en Ada d'un noyau pour le langage Ada,* Thèse de Docteur-Ingénieur, Université de Rennes 1, décembre 1985.

[Rosen 83] J. P. ROSEN : "A Kernel for Tasks and Rendezvous Management in Ada", *Proceedings of the Ada-Europe/AdaTEC Joint Conference on Ada,* Bruxelles, 16-17 mars 1983, 8(1-20).

[Rosen 84a] J. P. ROSEN : "On the Use of TEXT_IO on Interactive Terminals", *Proceedings of the IEEE Conference on Ada Applications and Environment,* St Paul (Minnesota), 16-18 Octobre 1984.

[Rosen 84b] J. P. ROSEN : "Arithmétique réelle en Ada", *BIGRE+Globule* 42, décembre 1984, 67-75.

[Rosen 86] J. P. ROSEN : *Une Machine Ada virtuelle : le système d'exploitation*, Thèse de Doctorat, Ecole Nationale Supérieure des Télécommunications, Paris, Octobre 1986.

[Rosenberg 80] J. ROSENBERG, D. A. LAMB, A. HISGEN et M. SHERMAN : "The Charette Ada Compiler", *SigPLAN Notices* 15 (11), Novembre 1980, ACM, 72-80.

[Rubine 82] D. H. RUBINE : *A Hybrid Ada Interpreter,* Mémorandum interne n°TM82-45412-13, AT&T Bell Laboratories, Murray Hill (N.J.), 1982.

[Schonberg 81] E. SCHONBERG, J. T. SCHWARTZ et M. SHARIR : "An Automatic Technique for Selection of Data Representation in SETL Programs", *ACM Transactions on Programming Languages and Systems* 3 (2), avril 1981, 126-143.

[Schonberg 82] E. SCHONBERG et G. A. FISHER,Jr. : "An Efficient Method for Handling Operator Overloading in Ada", *Proceedings of the AdaTEC Conference on Ada,* Arlington (Virginia) 6-8 Octobre 1982, ACM, 107-111.

[Schonberg 85] E. SCHONBERG et E. SCHONBERG : "Highly Parallel Ada - Ada on an Ultracomputer", *Proceedings of the Ada International Conference*, Paris, 14-16 mai 1985, 58-71.

[Schonberg 86] E. SCHONBERG et D. SHIELDS : "From prototype to efficient implementation : a case study using SETL and C", *Proceedings of the 19th Hawai International Conference on System Science*, B. Shriver (ed.), IEEE, 1986.

[Schulthess 82] P. SCHULTHESS et F. VONAESCH : "O.P.A - a New Architecture for Pascal-like Languages", *ACM Computer Architecture News* 10 (6), décembre 1982, 9-20.

[Sherman 80] M. SHERMAN *et al.* : "An Ada Code Generator for VAX 11/780 with Unix", *SigPLAN Notices* 15 (11), Novembre 1980, ACM, 91-100.

[Tannenbaum 83a] A. S. TANNENBAUM, H. VAN STAVEREN, E. G. KEIZER et J. W. STEVENSON : *Description of a Machine Architecture for use with Block Structured Languages*, Rapport n°IR-81, Vrije Universiteit, Amsterdam, août 1983.

[Tannenbaum 83b] A. S. TANNENBAUM, H. VAN STAVEREN, E. G. KEIZER et J. W. STEVENSON : "A Practical Tool Kit for Making Portable Compilers", *Communications of the ACM* 26 (9), septembre 83, 654-662.

[Vliet 85] J. C. Van VLIET et H. M. GLADNEY : "An Evaluation of Tagging", *Software-Practice and Experience* 15(9), Septembre 1985, 823-837.

[Waite 76] W. M. WAITE : "Relationship of Languages to Machines", *in: Compiler Construction - An Advanced Course,* J. Eickel (éd.), Springer-Verlag, Berlin, 1976.

[Wetherell 82] C. S. WETHERELL : *The Ada Breadboard Compiler : An Overview,* Mémorandum interne n°TM82-45412-13, AT&T Bell Laboratories, Murray Hill (N.J.), 1982.