Spécification formelle du composant Sum de Descartes

Synthèse

Cette note décrit la spécification formelle du composant Sum de Descartes. Celui-ci traduit un programme écrit en langage Sum, qui est un langage dit "algorithmique de bas niveau", en un programme écrit dans un langage cible. Le langage Sum est décrit dans une autre note [Dormoy 93a].

Le langage cible choisi ici est C. Toutefois, seul un petit sous-ensemble de C est utilisé. En particulier, les seules structures de données de C utilisées sont les tableaux, à quelques exceptions près. La spécification écrite devrait donc subir très peu de mofications pour être adaptée à d'autres langages cibles, comme Cobol ou AMPS.

Après une première maquette écrite en Shal, nous avons entièrement spécifié le composant Sum dans la langage de Descartes. La version actuelle de cette spécification est longue de 4621 lignes, et utilise 471 fonctions, plus 138 associations. Sa taille n'est donc pas négligeable.

L'essentiel du travail à effectuer pour passer de Sum au "C appauvri" est contenu dans trois tâches : le codage des constantes utilisées dans les types scalaires finis de Sum; la simulation de l'allocation dynamique dans un langage dont nous supposons qu'il ne contient pas cette fonctionalité; l'écriture de fonctions auxilliaires passant d'un type à un autre, et faisant le lien entre codages et objets codés. De plus, la traduction d'une simple instruction Sum peut comporter de nombreux cas, et en particulier produire plusieurs instructions C.

Pour résumer, l'esprit de Sum est d'appauvrir considérablement le niveau du langage dans lequel le programme généré est exprimé, et de prendre en charge la "bidouille" purement informatique, qui est relativement indépendante de l'écriture du programme vue d'un point de vue "conceptuel", mais qui n'en consomme pas moins une bonne partie du temps du programmeur.

1 Présentation générale de la spécification

1.1 Différences entre Sum, C et Cobol

Dans sa forme, Sum [Dormoy 93a] semble très peu différent d'un langage comme C. Si par contre nous le comparons à Cobol, les différences s'agrandissent, de la même manière que C semble éloigné de Cobol.

Pourtant, la spécification que nous avons écrite en Descartes (qui est un langage très compact) compte plus de 4500 lignes. D'où vient qu'il faille tant "d'instructions" pour franchir la distance apparemment courte entre Sum et C ?

A cela deux raisons. Premièrement, nous n'avons pas généré du C en utilisant toutes les possibilités de ce langage, loin de là. Nous avons plutôt généré du "sous-C", tellement appauvri qu'il est finalement très proche d'autres langages comme Fortran, Cobol, Basic, etc. Qu'on en juge: un programme issu de Sum ne contient comme seule structure de données que les tableaux d'entiers, de réels ou de caractères[1]. Il n'y a pas de sous-programme récursif. Aucune fonction d'allocation dynamique de mémoire n'est utilisée. Même si cela n'a pas été fait, il faudrait très peu de choses pour qu'il n'y ait plus de sous-programmes, de variables locales, etc. En fait, masqué sous la syntaxe de C, le programme généré est plutôt écrit dans un sur-assembleur, intersection des langages habituels.

La deuxième raison est plus fondamentale : la différence entre Sum est C n'est pas si petite qu'il y paraît au premier abord.
Il y a deux différences essentielles entre Sum et C. Premièrement, Sum autorise des *types scalaires finis* dont les éléments sont des constantes d'un autre type, éventuellement structuré. Ces types finis peuvent servir d'indices à des tableaux. On peut donc avoir en Sum un tableau indicé par (un nombre fini de) chaînes de caractères, par exemple.

Deuxièmement, les types de Sum sont structurés en une hiérarchie de sous-types, qui permet par généricité l'expression de clichés de programmation. Ces sous-types élargissent la compatibilité habituellement admise entre types. Ainsi, une affectation peut produire une erreur !

1.2 Esprit de la spécification

Il s'avère que la suppression de ces éléments de Sum pour passer à C n'est pas immédiate. D'ailleurs, notre expérience personnelle (écriture de Boojum en Pascal) nous a montré que les problèmes de codage et de compatibilité entre données mémorisées sous des formes différentes consomment une part importante du travail de programmation. C'est aussi une tâche assez pénible, car "bête" et répétitive, et nous ne sommes pas fâché de l'avoir effectuée "une bonne fois pour toute".

---

[1] Eventuellement de bits dans le futur.

En définitive, la spécification que nous avons écrite règle tous ces problèmes "d'un coup". Le langage obtenu est un C extrêmement appauvri, et des modifications minimes seraient nécessaires pour adapter la spécification à d'autres langages, comme Fortran, Cobol ou AMPS. Nous n'utiliserons aussi qu'une partie de ces langages. En fait, les programmes générés n'utilisent qu'un "sur-assembleur" qui se cache derrière une syntaxe C, Cobol ou autre.

1.3 Contenu et syntaxe d'un programme

Lorsque nous disons que notre spécification formelle écrite en Descartes génère un programme C, nous ne sommes pas tout à fait exact. En effet, nous générons une *représentation* d'un programme C (dans la structure de données de Descartes) qu'il s'agira encore d'écrire dans la syntaxe de ce langage. Cela ne signifie pas qu'il reste un travail "non fait", car le programme C généré est tout à fait complet. Il y a juste un problème de "réécriture", ou de "présentation", si l'on veut, du résultat. Donc, lorsque nous disons *programme C*, nous ne nous soucions pas de la *syntaxe* de C. D'ailleurs, comme dans ErgoAlg et Sum [Dormoy 93a] [Dormoy 93b], l'ordre total des instructions du programme C généré n'est pas précisé : seul un ordre partiel, avec l'utilisation de la notion de voisinage, est défini. Il est très simple à partir de cet ordre de reconstituer un ordre total.

Cela a en fait un énorme avantage. Nous verrons en effet que le passage à Cobol (cf. § 5) s'avère très facile en réutilisant intégralement la spécification de Sum pour C. Pourquoi ? Simplement parce que bien des notions sont communes aux langages C et Cobol, sous la surface de leurs syntaxes respectives. Par exemple, ces deux langages ont des affectations et des tableaux ! Et le fait de générer un programme écrit dans un langage C considérablement appauvri facilite cette "translation" vers d'autres langages, ce qui était le but recherché.

Ainsi, à une toute petite exception près, nous pensons (nous en serons sûr après avoir essayé) que le programme généré par la spécification *actuelle* de Sum est *aussi* un programme *Fortran*.

1.4 Présentation de la suite de la note

La suite de la note est divisée en quatre parties. Le § 2 présente toutes les transformations effectuées sur les constantes Sum (notamment *codage* et *numérotation*), et tous les types de fonctions auxilliaires générées par Sum pour éventuellement effectuer ces transformations dans le programme généré. Le § 3 montre un exemple de traduction d'instruction Sum, à savoir l'affectation. Le § 4 présente la mise en œuvre de l'allocation dynamique dans le programme généré. Enfin, le § 5 mentionne les ajouts qu'il faudrait faire à la spécification actuelle pour générer un programme Cobol.

On trouvera en annexe un petit morceau de la spécification réalisée, à savoir les modèles conceptuels d'un programme Sum et d'un programme C, et quelques fonctions Descartes à la base de la spécification.

2 Transformations sur les constantes

Nous entreprenons la description des principales difficultés de la spécification. Nous commençons par les affaires de codage des constantes.

Sum comporte des types scalaires finis [Dormoy 93a]. Ces types sont des ensembles de constantes Sum, qui peuvent être structurées, et donc appartenir à un autre type. On impose cependant la contrainte que toutes les constantes d'un même type structuré soient toutes d'un même type. Ce type commun s'appelle le *type générique* du type scalaire fini. Un type scalaire fini est considéré comme *sous-type* de son type générique.

2.1 Codage des éléments des types scalaires finis

Pour homogénéiser la représentation informatique, les constantes éléments d'au moins un type scalaire fini sont en général codées par des entiers positifs[2]. Le codage obéit aux contraintes suivantes :

- Il est injectif (i.e. deux constantes distinctes ont deux codes distincts).

- Il ne dépend pas des types scalaires finis, c'est-à-dire qu'une constante a un code unique, même si elle appartient à plusieurs types scalaires finis.

Prenons un exemple. Soit le type Sum suivant

JDS = {Lundi,Mardi,Mercredi,Jeudi,Vendredi,Samedi,Dimanche}

dont le type générique est "chaîne de caractères" (par exemple un pointeur vers un caractère).

Les éléments de ce type seront (par exemple) respectivement codés en 2, 4, 7, 19, 23, 45, 32 (on verra qu'il n'est pas toujours possible d'obtenir un codage en intervalle).

Cela a des implications sur la traduction des variables et instructions Sum en C. Ainsi, si dans le programme Sum on a l'instruction

X <- Samedi

Cette instruction sera traduite en C par

X' <- 45

avec X' de type entier.

---

[2] Cependant, lorsque rien ne pousse à coder de telles constantes (pas d'utilisation en indice de tableau, pas d'affectation avec d'autres types, …), elles ne sont pas codées. Le type scalaire fini est alors utilisé dans ce cas de la même manière que son type générique. Le codage n'est donc pas tout à fait systématique.

Nous verrons qu'il est préférable de coder les éléments d'un même type scalaire fini par des entiers formant un intervalle. Cependant, cela n'est pas toujours possible[3]. De plus, nous choisissons de coder les entiers par eux-mêmes. En pratique, on code donc d'abord les entiers, puis les autres éléments par des codes non encore utilisés.

A chaque fois, donc, qu'un type scalaire fini est utilisé, on le remplacera par son code.

2.2 Codage et décodage

2.2.1 Motivation

Néanmoins, il est parfois nécessaire d'utiliser en C l'élément du type Sum en tant que tel. Par exemple, si l'on doit imprimer le contenu de la variable X ci-dessus,

```
print(fichier,X)
```

cette instruction serait incorrectement traduite dans le langage cible par

```
print(fichier,X')
```

qui imprimerait 45, au lieu de samedi.

Il faut donc introduire dans le langage cible un moyen de décodage des codes des éléments de JDS.

D'autres cas peuvent se produire : le type JDS est compatible avec les chaînes de caractères, on peut avoir dans le programme Sum une affectation

```
Y <- X
```

où Y est de type chaîne de caractères.

Dans ce cas aussi, il faudra traduire cette instruction en introduisant un décodage de X.

Inversement, une instruction de lecture, ou une affectation dans l'autre sens

```
X <- Y
```

nécessite de mettre en œuvre le codage, c'est-à-dire de passer dans le programme généré de "Samedi" à 45.

Enfin, ce qui vient d'être dit pour les affectations est également valable pour les appels de fonction.

2.2.2 Principe de mise en œuvre

---

[3] Exemple : $t_1 = \{a,b\}$, $t_2 = \{b,c\}$, $t_3 = \{a,c\}$.

Conceptuellement, mettre en œuvre codage ou décodage consiste à représenter des fonctions bijectives entre deux ensembles finis. Le moyen le plus simple dans les langages de programmation ordinaires est d'utiliser un tableau ou une fonction. Cependant, le tableau n'est pas toujours possible, pour des raisons de type des indices, ou souhaitable, pour des raisons de taille de mémoire utilisée. De plus la fonction peut s'avérer particulièrement simple, et être préférée au tableau.

La mise en œuvre du codage ou du décodage est choisie par les heuristiques suivantes.

2.2.3 Mise en œuvre du décodage

Il s'agit, étant donné un code, de retrouver l'objet initial.

- Si l'objet est un entier, son code est égal à lui-même, et il n'y a rien à représenter.

Regardons les autres cas. Soient M et m respectivement les codes maximal et minimal du type. Soit d = M - m + 1. Soit c le cardinal du type.

- Si d/c = 2, alors on choisit une représentation en tableau :
  TableauDecode[code] = objet

Ce tableau devra être déclaré en variable globale et initialisé dans une procédure appelée au début du programme. Si le langage cible n'accepte que des indices de tableau commençant à 0 (comme dans le langage C), il faut de plus introduire un décalage, de manière à ce que

TableauDeCode[code-m] = objet

Il peut évidemment y avoir des "trous" dans ce tableau, et c'est la raison pour laquelle on utilise l'heuristique de ne pas utiliser une mémoire de plus du double du nombres d'éléments à représenter (car, pour que cette représentation ait un sens, au moins la moitié des codes doit être utilisée).

- Sinon (i.e. si d/c > 2), on choisit une représentation en fonction.

Cela signifie que l'on synthétise une fonction spécifique dans le langage cible, dont le corps est une suite de if-then-else fournissant l'objet en fonction du code. Les if-then-else sont cependant construits par dichotomie de l'ensemble des codes de manière à ce que le nombre de tests effectués lors de son appel soit en nombre au plus logarithmique en fonction du cardinal du type.

Exemple :

Soit le nouveau type scalaire fini Sum

Ecours= {"Français","Arabe","Mathématiques","Musique"}

Supposons que les codes de Ecours sont 23, 1, 5, 107. On a donc

m = 1 M = 107 d = 107 c = 4

Comme d/c = 26,75 > 2, on choisit une représentation en fonction. On doit donc écrire une fonction C, disons decEcours :

```
char *decEcours(code)
int code;

{
if code<=5 then
    if code=1 then return "Arabe"
    else return "Mathématiques"
else if code = 23 then return "Français"
    else return "Musique"
}
```

On voit que la fonction fait au plus $\log_2(4) = 2$ tests.

Si, par contre, les codes de Ecours sont 53, 54, 55, 56 alors

m = 53 M = 56 d = 4 c = 4

Comme d/c = 1 <= 2, on choisit une représentation en tableau. On doit donc initialiser un tableau (avec décalage m = 53), disons decEcours :

decEcours[0] = "Français"
decEcours[1] = "Arabe"
decEcours[2] = "Mathématiques"
decEcours[3] = "Musique"

2.2.4 Mise en œuvre du codage

Les types de type générique Entier ne se voient pas associer de moyen de codage, puisqu'un entier est identique à son code. Dans les autres cas, utiliser un tableau est impossible. Il faut donc utiliser une fonction, qui aura un squelette constitué de if-then-else imbriqués. Cependant, si le type générique du type scalaire fini vers lequel on code est structuré (tableau ou enregistrement), il n'est pas possible de procéder par dichotomie. On a donc une simple séquence de conditions. De plus, les comparaisons dans chaque *if* doivent comparer le contenu de la quantité à coder. Elles appellent donc des *fonctions de comparaison*, générées par ailleurs (cf. § 2.5).

2.3 Numérotation

2.3.1 Motivation

Un type scalaire fini peut servir de type indice à un tableau. En général, les langages cibles possèdent des types tableaux, mais indicés par des entiers d'un intervalle, voire d'un intervalle commençant toujours à 0 (langage C). Si un tableau Sum est indicé par un type contenant des chaînes de caractères, il est impossible de traduire directement les instructions mentionnant le tableau. De plus, les codes des éléments d'un type scalaire fini ne forment pas toujours un intervalle.

Il est donc nécessaire d'introduire un nouveau "codage" des types finis indices de tableaux, que nous appelons *numérotation*. Il est aussi nécessaire de mettre en œuvre dans le langage cible des moyens de passage du code à la numérotation.

Les exemples qui suivent montrent que la numérotation doit être différente du code, et qu'une constante doit être numérotée de façon différente selon le type scalaire fini auquel elle appartient.

Exemple 1 :

Si la variable X du type JDS est utilisée dans le tableau T de type d'indice JDS

T[X] <- 25

Une traduction brutale donnerait

T[X'] <- 25

avec X' valant 45 si X est Samedi.

Il faut en fait traduire en

T[numéro(X')] <- 25

avec numéro(45) = 5 si JDS est numéroté sur [0,6].

Exemple 2 :

Soit un autre type, l'ensemble des jours de repos des enfants JRE (sous-type de JDS)

JRE = {Mercredi,Samedi,Dimanche}

Le codage de Samedi reste 45, mais, si JRE est utilisé comme indice de tableau, sa numérotation ne pourra pas être 5. Ici, il faudra par exemple numéroter Mercredi, Samedi et Dimanche respectivement par 0 et 1 et 2.

La numérotation dépend donc du type (contrairement au codage).

2.3.2 Mise en œuvre

Seul le passage code --> numérotation est nécessaire, et seulement pour les types scalaires finis mentionnés comme indices d'au moins un type tableau (les autres ne

sont pas numérotés). De plus, il doit y avoir au moins une instruction mentionnant l'indice du tableau sous forme variable (car sinon on peut remplacer la constante par son numéro "à la compilation").

La mise en œuvre va beaucoup ressembler à celle du décodage, avec cependant une petite différence. Les numéros doivent être situés dans l'intervalle [0, cardinal du type - 1] en C.

- Si les codes du type forment un intervalle [m, M] où m et M représentent respectivement les codes minimal et maximal du type alors Num(code) = code - m et on n'utilise ni tableau ni fonction.

Dans le cas contraire, la numérotation est traduite de manière similaire au décodage, c'est-à-dire (avec d = M - m + 1 et c le cardinal du type) :

- Si d/c = 2 , le passage à la numérotation est effectué par tableau et décalage, soit

  Num(code) = TableauDeCode [code - m]

- Si d/c > 2, le passage du code à la numérotation se fait à travers une fonction constituée de if-then-else imbriqués (de façon logarithmique).

2.4 Changement de type

Dans la définition du langage Sum, certains types distincts sont compatibles. La règle générale est que l'on peut affecter un type à un sur-type, ou un type à un sous-type, mais dans ce deuxième cas avec risque d'erreur.

Toutes les affectations de ce type sont repérées dans le programme, et un ensemble de couple (t_1,t_2) est construit, où t_2 est "affecté" à t_1. Outre les instructions d'affectation *stricto sensu*, les passages de paramètres dans des appels de procédures ou fonctions et les instructions *return* conduisent également à des "affectations".

Pour chacun des couples où un type est affecté à un sous-type, une fonction est construite, qui vérifie que l'affectation est correcte, et qui produit une erreur sinon. Le cas particulier où t_1 est un type scalaire fini, et t_2 son type générique est déjà traité par la fonction de codage. Sinon, cette fonction dépend de la catégorie des types impliqués.

S'il s'agit de types scalaires finis, il s'agit simplement de vérifier que la valeur de la variable du sur-type est bien élément du sous-type. Elle a en général la forme de if-then-else imbriqués sur les codes. En fait, l'ensemble des codes de t_1 est considéré comme réunion d'intervalles disjoints, et la fonction vérifie que le paramètre appartient à un de ces intervalles. Dans le cas où l'ensemble des codes tout entier de t_1 est un intervalle, la fonction se résume donc à deux tests.

S'il s'agit de pointeurs, aucune fonction n'est générée. Une vérification aurait en effet peu de sens et serait de toute façon embêtante (cf. infra).

S'il s'agit d'enregistrements ou de tableaux, l'affectation entraîne une copie similaire aux fonctions de copie (cf. § 2.5 suivant), mais chaque copie élémentaire passe (éventuellement) par une fonction d'affectation à un sous-type. Une telle fonction est donc aussi générée pour chaque couple de types impliqué. Il est à remarquer que cela n'entraîne pas la réintroduction de la récursivité par le biais, puisque, si les structures de données peuvent être "récursives", la récursion "passe" nécessairement par un pointeur, auquel cas l'affectation est faite sans vérification.

2.5 Copie et comparaison

Les affectations entre types "complexes" (tableaux et enregistrements) sont autorisées. Une telle affectation implique évidemment une copie.

De plus, lorsqu'on a une affectation V = v, où v est de type scalaire fini et pas V, alors on *copie* v (après l'avoir décodé si nécessaire). Cela permet d'éviter que la constante valeur de v soit "abîmée" par une utilisation subséquente peu prudente de V[4].

Les appels de fonctions ou de procédures peuvent également entraîner une "affectation" du paramètre d'appel vers le paramètre formel (en entrée), et l'inverse (en sortie). Dans certains cas (lorsqu'un des deux paramètres est de type scalaire fini), on peut adjoindre une copie au décodage.

Enfin, une instruction *return* peut également être sujette à copie dans certains cas.

Par ailleurs, une comparaison entre quantités de types structurés (uniquement "==" et "!=") entraîne la comparaison de chaque composante.

Sum génère des fonctions pour effectuer des copies ou comparaisons pour les types pour lequels cela est nécessaire. Là non plus, il n'y a pas de "risque de récursivité", puisque comparaisons et affectations "s'arrêtent aux pointeurs".

3 Traduction des expressions et instructions Sum

3.1 Principe

Une simple affectation, expression de tableau ou comparaison peut donner lieu à l'utilisation dans le programme généré d'un ou plusieurs des six "opérateurs" suivants : codage, décodage, numérotation, affectation à un sous-type, copie ou comparaison de types structurés.

---

[4] Par exemple dans les instructions

```
V = "Samedi";
couper(V)
```

où V est de type tableau de caractères et *couper* une fonction qui coupe la chaîne mémorisée dans V d'une manière quelconque. Il ne faut pas *couper* la constante "Samedi", mais une copie de celle-ci.

Le seul moyen de spécifier l'utilisation de ces opérateurs serait de donner la spécification formelle des fonctions Cogito qui effectuent la traduction des instructions ou expressions, qui font 9 pages bien serrées.

3.2 Exemple : la traduction d'une affectation

Nous nous contentons de spécifier ici en termes plus formels comment une simple affectation est traduite.

Soit donc une affectation Sum x = X. x et X sont des expressions Sum. Appelons tx le type de x, et TX le type de X. Si tx (resp. TX) est un type scalaire fini, on note tgx (resp. TGX) son type générique. Soient enfin xc et XC la traduction en C des expressions x et X. L'affectation va être traduite en C de la façon suivante :

```
Si tx est un type structuré alors
    Si TX est un type structuré alors
        Copie_tx_TX(xc,XC)
    Sinon
    Si TX est un type scalaire fini codé alors
        Copie_tx_TGX(xc,Decode_TX(XC))
    Sinon
        Copie_tx_TGX(xc,XC)
Sinon
Si tx est un type scalaire fini codé alors
    Si TX n'est pas un type scalaire fini codé alors
        xc = Code_tx(XC)
    Sinon
    Si tx est un sous-type strict de TX alors
        AST_tx_TX(xc,XC)
    Sinon
        xc = XC
Sinon
Si TX est un type scalaire fini codé alors
    Si tx est un sous-type strict de TX et tx est un type
       scalaire alors
        AST_tx_TGX(xc,Decode_TX(XC))
    Sinon
        xc = Decode_TX(XC)
Sinon
Si tx est un sous-type strict de TX et tx est un type
   scalaire alors
    AST_tx_TX(xc,XC)
Sinon
    xc = XC
```

Spécification en Cogito de la traduction en C d'une affectation Sum

Notations :

- Un type structuré est un type tableau ou enregistrement. Un type scalaire est un type scalaire fini, ou Entier ou Reel.

- Copie_tx_TX est la fonction C générée par Sum qui copie une quantité de type TX dans une quantité de type tx (cette copie effectue aussi l'affectation à un sous-type si tx est sous-type de TX, avec renvoi d'erreur lorsque l'affectation n'est pas possible).

- Decode_TX est la fonction générée par Sum qui, pour un code de quantité de type scalaire fini TX renvoie la quantité elle-même.

- Code_TX est la fonction inverse (de codage).

- AST_tx_TX est la fonction qui effectue l'affectation d'une quantité de type TX vers une quantité du sous-type tx (uniquement types scalaires, et renvoi d'erreur lorsque l'affectation n'est pas possible).

Les "opérateurs" de numérotation ne sont pas mentionnés ici, car ils n'interviennent pas dans l'affectation. On voit que sur 10 cas, deux seulement correspondent à une traduction directe d'une affectation Sum par l'affectation C sur les expressions correspondantes.

Pour donner une idée de la "complexité combinatoire" de la traduction, la traduction d'un paramètre d'appel d'une fonction ou procédure est découpée en 30 cas.

4 L'allocation dynamique

4.1 Principe

Sum possède une fonction d'allocation dynamique, New. L'instruction

```
p = New(t)
```

alloue la place nécessaire à la mémorisation d'un élément de type t, et renvoie dans la variable p (qui est du type pointeur vers t) l'adresse de cet emplacement.

Sum possède également une fonction de désallocation, Dispose, qui "supprime" un "morceau de mémoire" précédemment alloué. Ainsi, l'instruction

```
Dispose(p)
```

"supprime" la zone mémoire pointée par p (précédemment allouée), et la rend donc disponible pour une future ré-allocation.

Des langages comme Pascal ou C fournissent de telles primitives d'allocation. Or, il s'avère que nous voulons que Sum aboutisse à un programme n'utilisant qu'un sous-langage de C où les seules structures de données complexes sont les tableaux, et où

l'allocation dynamique n'existe pas. Cela ne serait pas indispensable pour C, mais l'est bien pour Cobol, Fortran ou AMPS, par exemple.

Nous montrons ici comment faire. Nous avons en fait traduit dans la spécification de Sum notre expertise personnelle provenant de l'écriture de Boojum, puis des modifications apportées dans le passage à Genesia II.

4.1.1 Grands tableaux

L'idée est de simuler dans le programme l'allocation dynamique par l'utilisation de *grands tableaux*. Ces grands tableaux fournissent en fait le stock d'emplacements mémoire nécessaires à l'allocation dynamique. On peut parler de mémoire virtuelle.

Un grand tableau est alloué par unité de mémoire ou type informatique essentiels : entier, réel et caractère. Entiers et réels sont en général représentés sur un mot machine (le plus souvent 4 octets, sauf sur une machine comme le Cray, mais l'adaptation serait très facile), et les caractères sur un octet. La différence entre entiers et réels est qu'ils ne sont pas représentés ni utilisés de la même manière, et qu'il est moins coûteux de disposer de deux tableaux distincts plutôt que de mettre en œuvre des fonctions de conversion.

Il aurait enfin été envisageable d'allouer aussi un grand tableau de bits pour représenter les booléens. Cependant, outre que les compilateurs fournis par les constructeurs ne se gênent pas pour représenter les booléens sur tout un caractère (utilisant donc 8 bits là où un seul est significatif)[5], les programmes devant descendre à une utilisation fine du bit ne sont pas légion, en particulier vu le type d'applications que nous visons[6]. Nous nous contentons donc de représenter les booléens sur un octet, comme les caractères.

Nous aurions pu aussi nous limiter à allouer des zones mémoire de tailles libellées dans une unité fixe (en octets par exemple), sans nous soucier du type. Cependant, cela aurait conduit à des difficultés de partage entre langages, voire entre modèles d'ordinateurs.

4.1.2 Allocation et désallocation

Une fois que cela est fait, l'allocation dynamique (de même d'ailleurs que l'allocation des variables d'un type structuré) consiste à réserver un ou plusieurs morceaux de mémoire connexes de taille suffisante dans ces tableaux. Ensuite, des conventions fixes sont adoptées pour représenter ce qu'on a à y stocker.

Il y a deux problèmes à résoudre pour mettre cela en œuvre "proprement". Premièrement, il faut adopter ces conventions de représentation, qui doivent être

---

[5] Ce qui a l'avantage, considérable pour un constructeur qui vend bien sûr de la mémoire, d'en consommer beaucoup plus que nécessaire.

[6] Il y a bien les programmes d'échecs, où la coïncidence entre les 32 pièces, les 64 cases de l'échiquier et les mots de 32 bits de l'ordinateur est utilisée à fond, mais il s'agit là d'une espèce de programmes assez marginale, somme toute.

économiques mais générales. Deuxièmement, on peut être conduit dans le programme à *ne plus utiliser une zone mémoire précédemment allouée*. Cela se fait par la fonction Sum duale de la fonction New qui alloue la mémoire, Dispose. Il faut donc être capable de "récupérer" cette zone de mémoire en vue d'une réutilisation future.

C'est là le principe même de *l'allocation dynamique*. Sans vouloir faire un cours sur ce sujet, l'allocation dynamique est une des deux grandes méthodes de gestion de la mémoire. L'autre est constitué par les *garbages collectors*, utilisés dans Lisp, notamment. Un garbage collector ne se soucie pas de la mémoire "perdue" tant qu'il en reste suffisamment, et tente de tout "récupérer" d'un coup lorsqu'elle vient à manquer.

Un algorithme d'allocation dynamique est quelque chose de difficile à mettre en œuvre. Leur mise au point dans les années 50-60 a été cruciale pour le développement de l'informatique [Knuth 65]. La raison de cette difficulté est que les programmes peuvent demander des éléments de mémoire de *n'importe quelle taille*, et à *n'importe quel moment*. Ils doivent donc être très "rusés", et très généraux.

4.1.3 Principe de la mise en œuvre

Or, la situation avec Sum est très différente. En effet, la donnée du programme suffit pour connaître les tailles des types dont le programme est susceptible de demander l'allocation. Idem pour les types qui seront désalloués. Nous avons donc encore le "n'importe quand", mais nous n'avons plus le "n'importe quelle taille".

Nous simplifions donc le problème de l'allocation dynamique en générant des fonctions auxilliaires pour allouer et désallouer les zones de mémoire pour les types susceptibles de participer à l'allocation, et en utilisant des variables supplémentaires (une par taille susceptible d'être désallouée) qui serviront à mémoriser les emplacements de mémoire qui ont été désalloués (des "listes-poubelles). Lorsque l'allocation d'une zone mémoire est demandée, la fonction d'allocation ira d'abord voir s'il n'existe pas une zone de mêmes caractéristiques précédemment allouée et mémorisée dans la liste-poubelle ad hoc.

4.2 Mise en œuvre précise

4.2.1 Réservation des "grands tableaux"

Les grands tableaux ne sont pas systématiquement réservés. Si, à l'extrême, les programmes Sum à traduire ne mentionennt aucune allocation, aucun grand tableau n'est alloué. Si une allocation est mentionnée, un tableau pour les entiers est systématiquement alloué (on verra que cela est de toute façon indispensable). Si les allocations mentionnent des réels, des caractères ou des booléens, des tableaux ad hoc sont aussi alloués.

"L'allocation" des grands tableaux se fait par la déclaration dans le programme C de variables globales. Les tailles respectives des grands tableaux dépendent un peu arbitrairement de ceux qui sont alloués. Grosso modo, si p tableaux sont alloués,

chacun occupe une place égale à M/p, où M est la mémoire totale disponible (à régler selon l'ordinateur).

4.2.2 Fonctions d'allocation

Une fonction d'allocation est générée pour chaque type susceptible d'être alloué. Cette fonction est différente selon que le type est aussi susceptible d'être désalloué ou pas. Dans le second cas, l'étape 1 de l'algorithme ci-dessous n'existe pas.

La forme générale de la fonction d'allocation est la suivante :

1 - Regarder si la liste-poubelle correspondant aux éléments de la taille du type à allouer est vide. Si elle n'est pas vide, prendre un élément de cette liste, et sortir.

2 - Sinon allouer l'espace nécessaire dans le grand tableau ad hoc.

Pour chaque tableau, une variable globale est maintenue, qui indique l'indice maximum des zones de mémoire utilisées. On appelle cet indice *l'indice d'occupation* du grand tableau correspondant. Cet indice est incrémenté dans l'étape 2 ci-dessus. Si cet indice vient à dépasser la taille du tableau, alors l'allocation est impossible (plus de mémoire disponible), une erreur est produite, et le programme s'arrête brutalement.

En fait, dans le cas où le type est susceptible de désallocation, on ajoute au sein de l'étape 1 de l'algorithme précédent l'étape 1' :

1'- Si la zone mémoire récupérée dans la liste-poubelle est à l'extrémité droite de toutes les zones allouées dans le grand tableau, alors décrémenter l'indice d'occupation, supprimer cette zone de la liste-poubelle et retourner en 1.

Ainsi, l'indice d'occupation pourra "descendre". Cela diminue la probabilité qu'une "grande" zone mémoire ne puisse être allouée alors que de nombreuses "petites" zones sont disponibles.

Il est clair que cet algorithme pourrait encore être amélioré. Cependant, un algorithme encore plus simple a été mis en œuvre dans Boojum et Genesia II, et il n'a jamais posé de problème insurmontable.

Cet algorithme va être précisé pour chaque catégorie de type (enregistrements, …).

4.2.3 Fonctions de désallocation

Le schéma général consiste en une unique étape :

1 - Mettre dans la liste-poubelle ad hoc le morceau de mémoire à désallouer.

Cependant, une autre étape 0 est ajoutée avant l'étape 1, qui permet de faire "descendre" dans certains cas l'indice d'occupation des grands tableaux :

0 - Si la zone à désallouer est à l'extrémité droite de toutes les zones allouées dans le tableau, alors décrémenter l'indice d'occupation d'autant, et sortir.

4.2.4 Chaînage des listes-poubelles du grand tableau d'entiers

Une liste-poubelle est, comme son nom l'indique, une liste, dont les éléments sont des zones de mémoire contiguës et toutes de même taille, taille qui caractérise justement la liste-poubelle, avec le type du grand tableau auquel elle fait référence. Ces éléments de mémoire sont donc chaînés, et le problème est de savoir comment ils le sont.

Si la liste-poubelle concerne le tableau entier, les choses sont faciles. En effet, chaque élément de mémoire à mémoriser dans la liste est suffisamment grand pour contenir au moins un entier. Or, l'adresse d'un tel élément est un indice du grand tableau, qui est lui aussi un entier. On peut donc mémoriser le lien avec l'élément suivant en stockant dans la première "case" d'un élément l'indice où commence l'élément suivant. La liste-poubelle elle-même est une simple variable entière qui contient l'adresse (indice du tableau) du premier élément de la liste. Enfin, la fin de liste est indiquée par une "adresse impossible", c'est-à-dire un entier qui *ne peut pas* être un indice de tableau, par exemple -1[7].

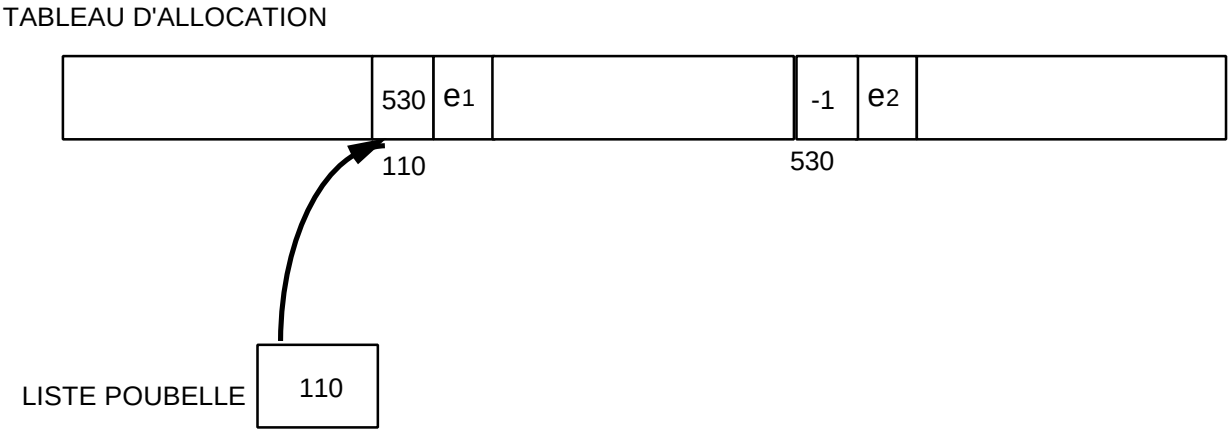

Figure 1 : liste-poubelle des paquets de 2 entiers

Cela est rendu plus clair dans la figure 1. Ici, la liste-poubelle contient des paquets de deux entiers. Le premier paquet commence à l'indice 110 , et c'est la valeur contenue par la liste-poubelle. La première case de ce premier paquet contient l'entier 530, qui est donc l'indice où commence le second paquet mémorisé dans la liste-poubelle (la valeur e1 contenue par la deuxième case de ce paquet n'a pas d'importance). La première case de ce second paquet contient -1, et la liste-poubelle se termine donc là.

On voit donc que la mémorisation de la liste-poubelle ne consomme aucune place supplémentaire (sinon un entier pour la variable elle-même), puisque les liens sont

[7] Les indices des grands tableaux commencent à 0. -1 représente donc *nil*, ou *null* en C.

justement mémorisés dans la place "perdue". Par ailleurs, ajouter et enlever un élément à la liste-poubelle est trivial.

4.2.5 Chaînage des listes-poubelles dans les autres grands tableaux

L'avantage du grand tableau entier était qu'il contenait justement des entiers. Or, les tableaux réel et caractère ne sont pas compatibles avec les entiers, et ne peuvent donc contenir des adresses (indices de tableau).

Pour chaîner les morceaux de mémoire désalloués dans ces tableaux, il faut allouer des éléments à deux entiers dans le tableau entier. La première case de chacun de ces paquets de deux entiers contiendra l'adresse de l'élément suivant (dans le tableau entier), et le deuxième l'indice du tableau réel ou caractère où débute la zone désallouée.

La figure 2 montre une disposition possible de la liste-poubelle pour les zones de 3 caractères.

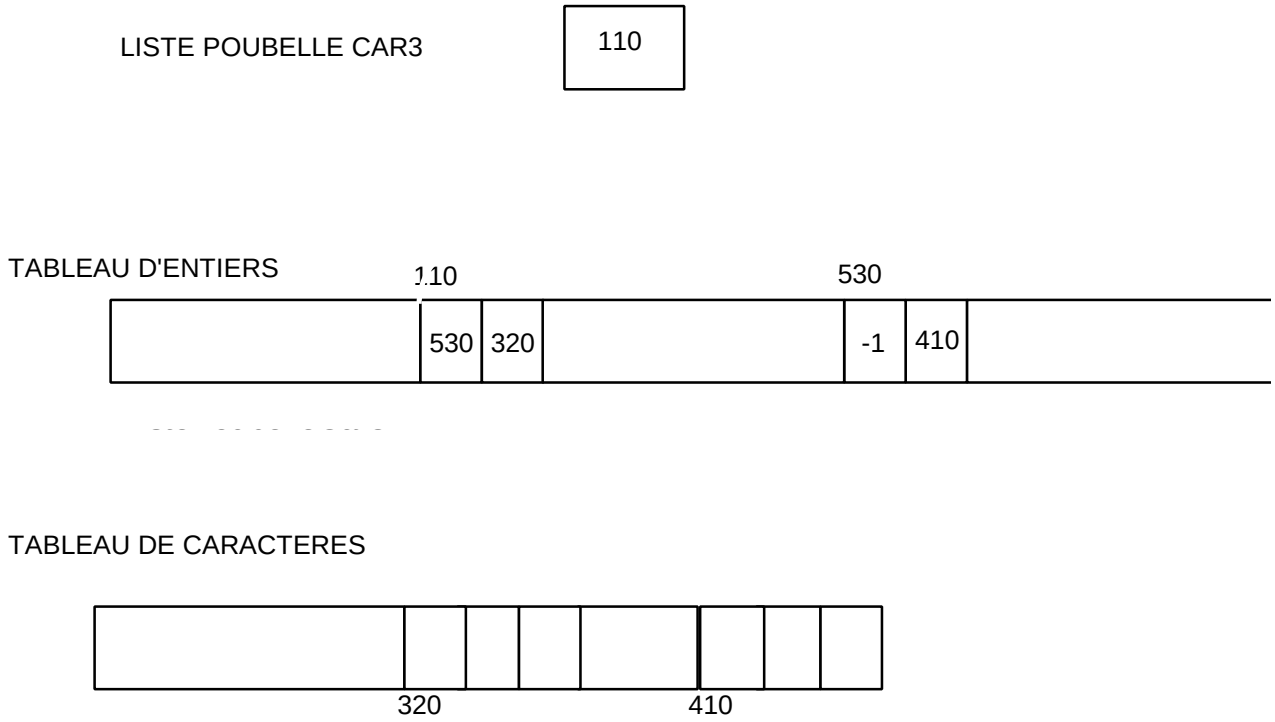

Figure 2 : liste-poubelle des paquets de 3 caractères

Contrairement aux listes-poubelles associées aux entiers, on a ici une perte de mémoire (puisque l'on consomme deux entiers). Cependant, lorsqu'un élément mémorisé dans une telle liste-poubelle est réutilisé pour une allocation de mémoire, l'élément de 2 entiers est désalloué (et peut donc être réutilisé). Il s'agit malgré tout d'un compromis, qui vise à ne pas "perdre le fil" de la mémoire désallouée.

4.2.6 Représentation et allocation des enregistrements

Les enregistrements sont constitués de champs, chaque champ ayant un type. Nous décrivons ici comment un tel enregistrement est représenté en mémoire.

Tout d'abord, un paquet d'entiers est alloué (dans le grand tableau entier), ayant un nombre d'entiers égal au nombre de champs de l'enregistrement. Ensuite, pour chaque champ :

- Si le champ est de type entier (ou est compatible avec le type entier, comme un pointeur, cf. infra), alors la valeur du champ est mémorisée dans la case ad hoc.

- Sinon, si le champ est du genre entier, mais occupe plus que la place d'un seul entier (par exemple un tableau d'entiers), ou s'il est du genre réel ou caractère, alors une zone mémoire est allouée pour le type de ce champ, et son adresse est mémorisée dans la case correspondant au champ.

Cela nécessite que les champs soient eux-mêmes codés par des entiers entre 0 et n-1, où n est le nombre de champs de l'enregistrement. Ainsi, si v est une variable du type enregistrement e contenant le champ c, et si le champ c a été codé par le nombre 5, l'accès au champ c se fait par l'expression C

TABE[vc + 5]

avec vc la traduction dans le langage-cible de la variable v, et où TABE est le grand tableau entier. La valeur de vc est l'indice de TABE où commence l'enregistrement.

Prenons un exemple (cf. figure 3). Soit un enregistrement e constitué de 4 champs C_1, C_2, C_3 et C_4 respectivement de types entier, caractère, tableau de 3 réels, enregistrement de champs D_1 et D_2, de types respectifs Entier et pointeur vers quelque chose.

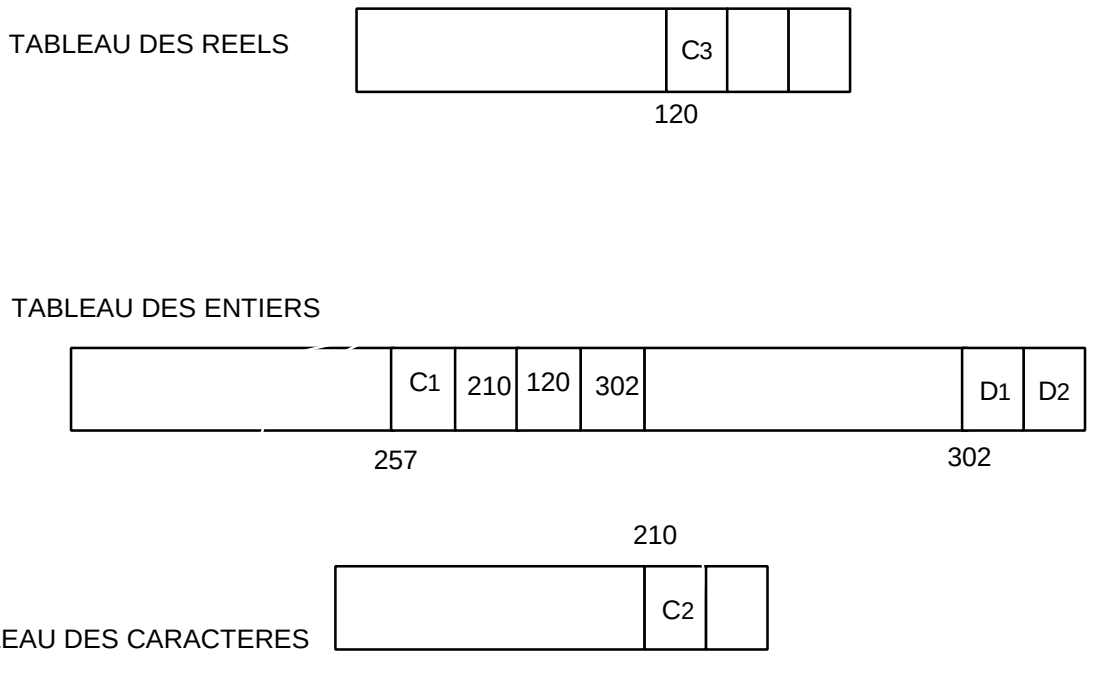

Figure 3 : représentation d'un enregistrement

L'allocation de l'espace à cet enregistrement se déroulera de la manière suivante :

1. Les champs sont codés par Sum, par exemple de sorte que le code de C_i soit i-1.
2. On alloue dans le grand tableau des entiers un paquet de 4 entiers, ce paquet commençant à l'adresse (indice du grand tableau) 257. On a :

- La case à l'adresse 257 contiendra la valeur du champ C_1.
- La case à l'adresse 257 + 1 = 258 contiendra le pointeur 210 de la zone allouée pour le contenu du champ C_2 dans le tableau des caractères.
- La case à l'adresse 257 + 2 = 259 contiendra le pointeur 120 de la zone allouée pour le champ C_3 dans le tableau des réels (paquet de 3 réels, cf. § 4.2.8).
- La case à l'adresse 257 + 3 = 260 contiendra le pointeur 302 de la zone réservée à l'enregistrement contenant les champs D_1 et D_2 dans le tableau des entiers, l'allocation de l'espace pour ces champs étant faite de la même manière.

Voyons maintenant comment se fait l'accès aux éléments de l'enregistrement. Supposons que nous ayons une variable v de type e, et dont la valeur à un instant donné soit l'enregistrement décrit ci-dessus. La variable correspondant en C à v, notée vc, est de type entier, et contient la valeur 257.

Pour accéder aux champs de v, on utilise en C les expressions :

TABE[vc]	Entier mémorisé dans le champ C_1
TABC[TABE[vc + 1]]	Caractère mémorisé dans le champ C_2
TABR[TABE[vc + 2] + 1]	Deuxième réel mémorisé dans le champ C_3
TABE[TABE[vc + 3] + 1]	Champ D_2 de l'enregistrement mémorisé dans le champ C_4

4.2.7 Codage des champs des enregistrements

Quelques remarques sur le codage des champs. Les champs sont codés par des entiers successifs commençant à 0. Cependant, ce codage n'est pas entièrement libre. On impose en effet qu'un même champ partagé par plusieurs enregistrements soit codé d'une manière unique.

Cela implique que les champs d'un sous-type d'un enregistrement soient codés par des nombres inférieurs à ceux qui ne le sont pas. Ainsi, si on a deux types d'enregistrements e et f, que f est un sur-type de e, alors les champs de f devront être codés par des nombres inférieurs à ceux des codes des enregistrements de e qui ne sont pas dans f[8].

Ces deux contraintes visent à assurer la généricité de façon "naturelle" dans la traduction des programmes Sum. En effet, toutes les opérations sur un champ d'un

[8] Rappelons qu'un type enregistrement a *plus* de champs que ses *sur*-types.

sur-type seront aussi valables sur le même champ d'un sous-type, puisque ce champ est dans les deux cas à la même place relative par rapport au "début" de l'enregistrement, et que les expressions générées pour y accéder sont identiques. De plus, il faut évidemment que les champs du sur-type soient "au début" des champs du sous-type pour ne pas perdre de place mémoire.

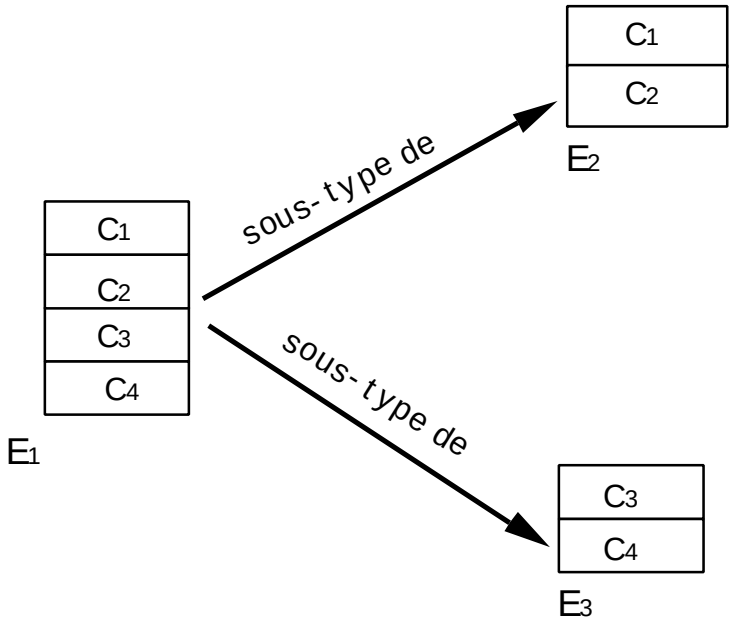

Figure 4 : codage des champs impossible

D'où la contrainte qu'un type enregistrement ne saurait être sous-type de deux types non comparables. Un exemple est fourni par la figure 4 : tous les champs C_1, C_2, C_3 et C_4 de E_1 devraient avoir un code dans {0,1}, du fait de la présence simultanée de E_2 et E_3.

4.2.8 Représentation et allocation des tableaux

Les tableaux peuvent être représentés dans les "grands tableaux". C'est le cas du tableau de 3 réels contenu dans le champ C_3 de l'enregistrement pris en exemple dans le § 4.2.6.

L'avantage des tableaux est que ses éléments sont de même type. Un tableau est donc tout simplement mémorisé dans un bloc de mémoire de taille le cardinal de ses indices, et de type fonction du type des éléments du tableau.

Si les éléments du tableau sont de type compatible entier, réel ou caractère, il n'y a pas de problème. Sinon, il faut passer, comme pour les enregistrements par une "indirection". Le bloc correspondant au tableau est alors alloué dans le grand tableau *entier*, des zones sont allouées dans le ou les tableaux ad hoc pour mémoriser les éléments du tableau (autant que d'éléments du tableau), et les adresses de ces zones sont mémorisées aux indices correspondants dans le bloc.

4.2.9 Allocation des pointeurs et des scalaires

L'allocation des éléments d'un type structuré (tableau ou enregistrement) "s'arrête aux pointeurs". Cela signifie que, par exemple, si un champ d'un enregistrement est d'un type pointeur, l'allocation de l'enregistrement implique l'allocation de la place pour mémoriser le pointeur (équivalente à un entier), mais pas de la place nécessaire à la mémorisation de ce que pointe le pointeur. D'ailleurs, s'il en était autrement, le système pourrait rapidement être conduit à allouer une place mémoire infinie (en cas de "circuit" dans la définition du type alloué).

Il n'y a donc allocation de mémoire concernant un pointeur que si l'on a un ordre

```
pp = New(t_ptr)
```

où t_ptr est un type pointeur. Cette instruction va allouer de la place pour *1* entier (qui est bien la place nécessaire au stockage d'une valeur de type pointeur), et renvoyer l'adresse de l'emplacement ainsi alloué. La variable pp est d'un type pointeur vers un pointeur. En pratique, les pointeurs vers les pointeurs sont rares.

De même, on peut avoir allocation d'un scalaire, comme dans

```
pe = New(Reel)
```

cette instruction alloue un emplacement dans le grand tableau réel pour *1* réel, et affecte l'indice du tableau correspondant à la variable C correspondant à pe.

4.2.10 Représentation des pointeurs

La question pour les pointeurs n'est pas tant de savoir comment ils sont alloués que de savoir ce qu'est "l'adresse" qu'ils mémorisent.

Nous avons vu qu'une quantité allouée l'est dans un ou plusieurs grands tableaux, et que "l'adresse" de la zone allouée est l'indice du grand tableau où commence le premier emplacement alloué. Nous définissons le pointeur vers cette zone comme étant cet indice.

Si l'on y regarde de près, cela aboutit à une situation qui peut paraître étrange. Donnons-nous un type t, par exemple le type enregistrement du § 4.2.6, v une variable de type t, et p de type pointeur vers t. Quelles valeurs contiennent les variables C pc et vc déduites de p et v par la traduction après que les instructions Sum (traduites en C)

```
p = New(t)
v = *p
```

ont été exécutées ?

D'après notre définition, pc contient l'indice où commence la zone allouée, par exemple 257 (cf. figure 3). Or, cette adresse est le seul moyen de caractériser

complètement l'enregistrement alloué. La variable vc contiendra donc aussi la valeur 257.

Autrement dit, l'instruction

```
v = *p
```

est toujours traduite en C par

```
vc = pc
```

i.e. des variables dans la situation de p et v auront "en C" la *même* valeur[9]. La différence tient en fait dans les types qu'ils représentent, et donc dans les opérations que l'on peut leur appliquer. Même si elles ont la même valeur, il faut bien *deux* variables.

Pour autant, le programmeur en Sum (humain ou synthétiseur de programme) n'a pas à se soucier de cette particularité de la traduction.

4.2.11 Allocation des variables de types structurés

Les fonctions et procédures de Sum *ne sont pas* récursives. Cela a pour conséquence qu'il est possible d'allouer a priori ("à la compilation") et une fois pour toutes ces variables, qu'elles soient globales ou locales.

Pour les variables d'un type scalaire (codé s'il est fini) ou d'un type pointeur, cela ne pose pas de problème, car leurs équivalentes en C seront d'un des types entier, réel ou caractère.

Pour les autres (variables de types structurés), il va falloir utiliser notre représentation. Dans notre spécification, des zones mémoire sont allouées pour toutes ces variables par une procédure d'initialisation générée, et exécutée au début du programme généré. Cela était de toute façon indispensable pour les variables de type enregistrement, mais aurait pu être modulé pour les variables de type tableau. Ainsi, une variable dont le type est "tableau de 255 caractères" aurait pu être traduite en C par une variable également de type "tableau de 255 caractères". Nous n'avons cependant pas fait cela, car une même expression Sum aurait alors été traduite différemment en C selon le mode d'allocation/déclaration en C de la variable qu'elle implique. Nous avons donc voulu éviter ce genre de jonglerie (et même sans doute d'autres problèmes).

---

[9] Plus précisément, c'est l'expression Sum *p qui est traduite en C par pc lorsque p est un pointeur vers un type structuré. Par contre, si p est un pointeur vers un scalaire ou un pointeur, l'expression *p sera traduite par TABE[p], TABR[p] ou TABC[p] selon le cas. C'est même encore un peu plus compliqué, dans le cas où p est un pointeur vers un type scalaire fini dont le type générique est structuré : on retombe dans le cas des types structurés après décodage.

En résumé, toutes les variables de types structurés ont pour équivalentes en C des variables de type *int*, ,la place mémoire qui leur est nécessaire est allouée au début de l'exécution du programme C dans les grands tableaux, et les adresses de ces zones leur sont allouées. On peut constater que la valeur de ces variables ne changera pas au cours de l'exécution du programme.

4.2.12 Entorses en C à la représentation des adresses

Nous avons en fait utilisé pour la traduction en C un artifice permis par ce langage, et qui change les expressions C générées mentionnant les grands tableaux.

Reprenons l'exemple du § 4.2.6. D'après ce que nous avons dit, l'expression Sum v.C_1 est traduite en C par l'expression TABE[vc] (pourvu que C_1 soit codé par 0). Cela est le schéma général, qui reste valable pour tous les autres langages, mais nous avons utilisé un raccourci pour C *qui peut améliorer l'efficacité du programme généré d'un facteur 2.*

En C, l'adresse d'un tableau est disponible au programmeur. Plus précisément, la valeur de la variable TABE est en fait l'adresse du début de la zone mémoire (cette fois au sens des "vraies" adresses gérées par le système d'exploitation[10]) où la place de TABE est réservée. Le type de TABE (tableau de xxx entiers) est compatible avec le type *int, c'est-à-dire pointeur vers un entier. Pour dénoter le contenu de la i+1ème case du tableau TABE, on peut utiliser indifféremment une des deux expressions

```
TABE[i]
*(TABE + i)
```

En fait, la première expression est traduite en la seconde par le compilateur C. L'addition dans l'expression TABE + i est l'addition des adresses. TABE étant un pointeur vers un entier, on augmente l'adresse contenue dans la variable TABE de i "crans", la taille d'un cran étant fonction du type pointé (ici, entier).

Les expressions traduites utilisant notre variable de type enregistrement contiennent toujours des expressions de la forme

```
TABE[v + i]
```

avec i le code d'un des champs. Vue sous forme "pointeur", cette expression est

```
*(TABE + v + i)
```

Il s'avère donc que, quelque soit i, la même expression TABE + v est calculée. Or, cette expression n'est rien d'autre que l'adresse (au sens du système d'exploitation) du début de la zone mémoire réservée à l'enregistrement. Il serait donc judicieux, pour

[10] Qui sont pourtant encore des adresses virtuelles…

éviter à chaque accès à l'enregistrement de recalculer TABE + v, *de mémoriser dans v cette adresse plutôt que l'indice du tableau*.

Une fois cette modification opérée, l'accès au champ de code i se fait par l'expression

```
*(v + i)
```

qui ne comporte plus qu'une addition au lieu de deux.

Cette "ruse" ne tient cependant que pour un langage comme C. Elle serait impossible en Pascal. En tout état de cause, la représentation des adresses par les indices des grands tableaux fonctionne quel que soit le langage.

5 Traduction en Cobol

Une stagiaire de fin de DESS, M^{elle} Fatima Belgoul, a étudié pendant l'été 92 la traduction en Cobol. Pour l'essentiel, ce qui a été dit pour C doit être repris. Il y a cependant quelques modifications à apporter, notamment quant à l'utilisation de l'opérateur de référencement & en C, qui sert à "simuler" les passages de paramètres par adresse dans ce langage (qui ne connaît que le passage par valeur). Une autre contrainte est levée : celle d'indicer les tableaux en commençant par 0. Il y a donc certains décalages dans nos numérotations qui peuvent disparaître, bien que le programme reste correct si nous ne le faisons pas.

A ces deux exceptions près, le passage à Cobol nécessite des *ajouts* à la spécification écrite pour C. Le point essentiel est qu'il n'y a pas de notion de fonction ou de procédure en Cobol, et par conséquent pas de variable locale non plus. Par ailleurs, il y a bien sûr des différences de syntaxe, mais nous ne nous en soucions pas ici.

Il s'avère que le travail à effectuer est mineur. Il est en effet facile de simuler l'appel d'une fonction ou d'une procédure par l'utilisation d'instructions *goto*. Il faut ajouter à cela le renommage des paramètres et variables locales aux fonctions pour les rendre globales, et faire précéder et succéder "l'appel" de fonction par des affectations ad hoc entre paramètres formels et d'appel. Cela est possible *parce que les programmes générés ne comportent pas de fonction récursive*. Ce serait sinon plus compliqué, car il faudrait alors simuler (au moins une partie de) la pile d'appel.

En pratique, plutôt que de modifier la spécification de Sum pour le langage-cible Cobol, il est plus simple de partir du programme "C" généré, et d'y opérer les modifications pour passer à Cobol. On peut donc dire en définitive que Sum-->C génère un nouveau langage-pivot, qui s'avère être un sous-langage de C, et que la spécification des modifications pour Cobol constitue un nouveau composant de Descartes, transformant un "sous-C" en "sur-assembleur". D'ailleurs, le programme "Cobol" final obtenu pourrait aussi bien être transcrit dans une syntaxe C (un sous-

sous-langage C encore plus appauvri, où il n'y aurait même pas de fonction, et où tout le programme consisterait en une unique fonction *main* !).

Il ne semble donc plus exister de problème grave au changement de langage-cible pour les applications futures.

Références

[Dormoy 93a] Jean-Luc Dormoy. Définition du langage Sum. Note EDF HI21/8342.

[Dormoy 93b] Jean-Luc Dormoy. Définition du langage ErgoAlg. Note EDF HI21/8340.

[Knuth 65] Knuth. *The Art of Computer Programming*. Volume 1 - *Fundamental Algorithms*. Addison-Wesley, 1965.

Annexe :
Modèle conceptuel Descartes de programmes Sum et C
et quelques fonctions Descartes de base

La spécification de Sum en Descartes est trop longue pour être fournie ici. Elle nécessiterait aussi des commentaires sur le langage de Descartes lui-même, que nous avons dû "interpréter" pour écrire la spécification, ce qui est hors du champ de la présente note. Nous nous contentons de donner le modèle conceptuel de programmes Sum et C (i.e. des données d'entrée et de sortie de la spécification), et de quelques fonctions de base, ceci pour donner une première idée de ce comment une partie de Descartes peut être spécifiée en lui-même.

En général, on présente une représentation graphique du modèle conceptuel, ce qui est plus agréable à l'œil. Cependant, un traitement informatisé d'une telle représentation n'existe pas encore, et nous avons donc écrit une représentation textuelle en vue d'une utilisation prochaine par Descartes.

Nous n'avons pas distingué dans le modèle conceptuel attributs et associations. De plus, les associations sont représentées comme des fonctions, car il s'est avéré que nous n'avons eu besoin dans la spécification de ces associations que prises dans un seul "sens".

On trouvera ensuite quelques fonctions "de base", qui spécifient ce que sont une sous-expression, un sous-type, etc.

Enfin, dans tout ce qui suit, une syntaxe "personnelle", mais constamment suivie, de Descartes a été adoptée. Elle n'est pas exactement compatible avec le premier analyseur syntaxique de Descartes qui a été réalisé, et un travail d'adaptation de l'une (la spécification) à l'autre (la syntaxe) devra être effectué.