DEFINITION DU LANGAGE ErgoAlg

JL.DORMOY

SYNTHESE :

Cette note décrit le langage ErgoAlg, un des langages intermédiaires du synthétiseur de programmes Descartes.

En général, un langage informatique est décrit par sa syntaxe (comment écrire un programme dans ce langage) et sa sémantique (ce qu'un programme écrit dans ce langage effectue).

Cependant, seul ce qui est dit ici sur la sémantique de ErgoAlg sera valable. En effet, ce langage est en quelque sorte "virtuel", puisque, intermédiaire dans la synthèse de programmes par Descartes, il n'est pas destiné à être utilisé manuellement comme langage de programmation. De plus, les programmes en ErgoAlg seront toujours *représentés* comme données d'un des langages de mise en œuvre du synthétiseur Descartes. Seule sa sémantique compte donc. Cependant, on utilisera une syntaxe "naturelle" pour énoncer les exemples.

ErgoAlg se distingue des autres langages informatiques par ses structures de données. Celles-ci comprennent la notion d'ensemble (et aussi celle de liste, mais cette dernière a moins d'importance). Ce faisant, la plupart des constructeurs et opérateurs sur les ensembles présents dans Cogito, le langage actuel d'entrée de Descartes, sont ici aussi utilisés. Cependant, des structures de données qui seraient équivalentes dans Cogito (au sens où Cogito peut passer de l'une à l'autre) ne sont plus ici équivalentes. Cela est en particulier vrai pour les *index*, qui modélisent les chemins construits et maintenus pour retrouver les données facilement. En résumé, ErgoAlg est plus proche de la machine que Cogito, et les choix faits par Cogito ne sont pas remis en cause par ErgoAlg.

Une autre particularité est qu'il y a une certaine identification entre types de données et données elles-mêmes. Par exemple, une variable ErgoAlg F pourra avoir pour type d'être sous-ensemble d'un ensemble E, E étant elle-même une variable ErgoAlg d'un type ensemble. Des restrictions seront

toutefois définies sur la stabilité d'un ensemble de données utilisé comme type.

L a description porte ensuite sur les instructions du langage. On y retrouve les instructions classiques d'un langage de programmation, en particulier l'affectation et les boucles, modulo le fait que les ensembles sont types de données. La notion de pointeur, déjà bien connue dans les langages comme C, est introduite, pour distinguer l'affectation de la référence à un objet complexe de l'affectation de l'objet lui-même (par copie). Cependant, dans un langage comme C, si la notion de pointeur peut être définie axiomatiquement (i.e. en dehors de toute référence à l'implantation physique des données), elle est en fait généralement comprise à partir de son implantation physique (quasiment canonique). Dans ErgoAlg, aucune référence à une implantation physique ne sera possible, car c'est justement le travail des composants ErgoAlg et Sum de la choisir parmi plusieurs possibilités.

Nous axons enfin notre présentation autour d'un exemple présenté et commenté dès le début. Le lecteur familier des langages de programmation avancés pourra ainsi rapidement saisir "l'esprit" de ErgoAlg, et sa description, qui suivra, ne sera plus qu'une explicitation "d'évidences".

1 Un programme ErgoAlg

Nous présentons ici un programme ErgoAlg. Nous avons choisi un programme non trivial, et qui nécessite une explication conséquente, parce qu'il représente justement assez bien le niveau de complexité visé.

1.1 La consistance d'arcs dans un CSP

Le programme suivant effectue ce que l'on appelle la consistance d'arcs dans un problème de satisfaction de contraintes (en Anglais, CSP, ou *Constraints Satisfaction Problem*). L'algorithme fourni est adapté de celui décrit par Mohr et Henderson en 1986 dans AI Journal [Mohr & Henderson, 1986]. Il était déjà présent dans Alice [Laurière, 1978].

Un CSP (binaire) est caractérisé par un ensemble de variables V, un ensemble K d'arcs (v,w) entre variables de V, avec v ? w, et pour chaque (v,w) de K la contrainte C(v,w), qui est un ensemble de couples d'objets éléments d'un certain univers U. On suppose de plus en général que les contraintes définies par K et C sont symétriques. Par exemple, on peut considérer le CSP

$V = \{v_1,v_2,v_3\}$
$K = \{(v_1,v_2),(v_2,v_3),(v_3,v_1)\}$
$C(v_1,v_2) = \{(a,a),(a,c),(b,b),(b,d),(b,e)\}$
$C(v_2,v_3) = \{(a,e),(b,e),(c,d),(d,a),(e,a)\}$
$C(v_3,v_1) = \{(d,b),(e,a),(a,e)\}$

(plus les arcs symétriques pour K et C).

Une représentation graphique de ce CSP est donnée par la figure 1.

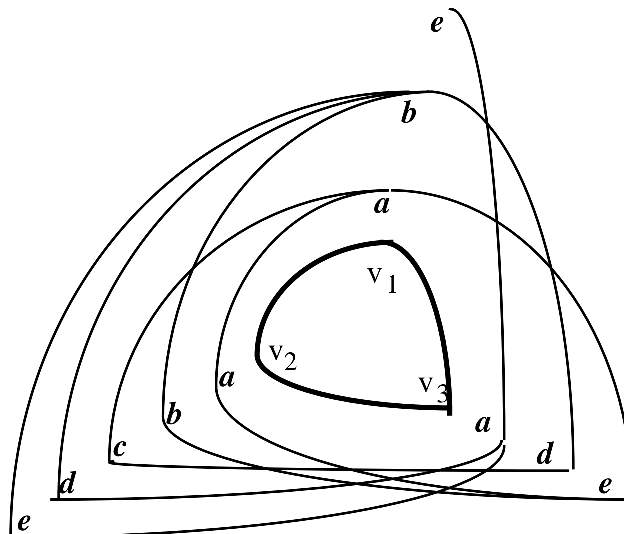

Figure 1 : un CSP

Résoudre un CSP, c'est trouver par quels objets remplacer les variables (chaque variable étant remplacée par un objet unique) pour satisfaire toutes les contraintes. A priori, un CSP peut avoir zéro, une, ou plusieurs solutions. Par exemple, le CSP ci-dessus a une seule solution :

$v_1 = a$, $v_2 = a$, $v_3 = e$

Beaucoup de travaux ont été consacrés à la résolution des CSP, qui est un problème NP-Complet. Certains ont imaginé des méthodes dites de *propagation* qui transforment n'importe quel CSP en un CSP plus simple, et cela par des algorithmes polynomiaux. Une de ces méthodes, la plus connue, est la *consistance d'arcs*. La consistance d'arcs consiste à tout d'abord considérer les domaines des variables du CSP, c'est-à-dire l'ensemble des valeurs de la variable pouvant participer à une solution, puis à éliminer des éléments de ces domaines selon la "règle" suivante : *si une variable v a l'objet x dans son domaine, et si pour un arc (v,w) de K, il n'y a pas de couple de la forme (x,y) dans C(v,w), alors supprimer x du domaine de v.* En effet, un tel objet x ne pourra participer à aucune solution.

Si l'on considère notre exemple, on voit que l'objet *e* doit être supprimé du domaine de v_1 (qui a été initialisé à $\{a,b,e\}$). Une fois cette suppression accomplie, l'objet *a* peut maintenant être supprimé du domaine de v_3, puis *d* et *e* du domaine de v_2. La consistance d'arcs s'arrête là.

On voit donc que la consistance d'arcs peut se "propager" récursivement dans le graphe des contraintes. On voit aussi qu'elle ne fait qu'approcher la solution, puisqu'elle laisse par exemple l'objet *b* dans le domaine de v_2, qui ne participe à aucune solution globale.

Quel algorithme mettre en œuvre pour effectuer cette consistance d'arcs ? Il est étonnant de constater qu'il a fallu attendre 16 années pour qu'un algorithme optimal (en complexité) soit publié, en 1986, et après trois autres algorithmes sous-optimaux. Le premier algorithme (dit de Waltz) avait été publié en 1972, dans un contexte de traitement d'images. Toutefois, Jean-Louis Laurière avait mis en œuvre l'algorithme optimal dans Alice en 1976, mais n'avait pas jugé bon de mettre en valeur une telle trivialité (par rapport à la complexité d'Alice).

1.2 L'algorithme de Mohr et Henderson

L'algorithme de Mohr et Henderson, écrit en ErgoAlg, est donné en page suivante. Cet algorithme porte dans la littérature le nom d'AC-4.

```
ArcCons(V↓,K↓,C↓,U↓,D↑)

V : Set(Variable)            /* L'ensemble des variables du CSP */
K : Set(V x V)          /* Le graphe de contraintes */
C : K --> Set(U x U)         /* Les contraintes */
G : V --> Set(V)             /* Les successeurs du graphe de
                                contraintes */
D : V --> Set(U)             /* Les domaines des variables */
U : Set(Objet)          /* L'univers des objets considérés */
M : V x U --> Set(V x U)     /* Les contraintes indexées par
                                (var,obj) */
P : List(V x U)              /* Liste des couples (var,obj) à
                                propager */
E : V x U --> {0,1}          /* Couples (var,obj) déjà ôtés */
N : V x V x U --> Entier     /* Compteur (var,var,obj) des
                                contraintes restantes */
v, w : V
a, b : U


P <-- ø
/* Initialisation et construction de G */
Pour tout v de V faire
     G(v) <-- ø
Pour tout (v,w) de K faire
     G(v) <-- G(v) + {w}

Pour tout v de V faire
     D(v) <-- ø
/* Construction des domaines D initiaux */
     Pour tout w de G(v) faire
          Pour tout (a,b) de C(v,w) faire
               D(v) <-- D(v) ∪ {a}
               M(w,b) <-- ø
               N(v,w,a) <-- 0
/* Construction de M, et des N initiaux */
     Pour tout w de G(v) faire
          Pour tout (a,b) de C(v,w) faire
               M(w,b) <-- M(w,b) + {(v,a)}
               N(v,w,a) <-- N(v,w,a) + 1
/* Initialisation des couples (var,obj) à propager */
     Pour tout a de D(v) faire
          E(v,a) <-- 0
          Pour tout w de G(v) faire
               Si N(v,w,a) = 0 alors
                    P <-- (v,a).P
                    E(v,a) <- 1

/* Consistance d'arcs en tant que telle */
Tant que P ? () faire
     (w,b) <-- Tete(P)
     P <-- Fin(P)
     E(w,b) <-- 1
     D(w) <-- D(w) - {b}
     Pour tout (v,a) de M(w,b) faire
          N(v,w,a) <-- N(v,w,a) - 1
          Si E(v,a) = 0 et N(v,w,a) = 0 alors
               P <-- (v,a).P
               E(v,a) <-- 1
```

On peut noter que seules les 10 dernières lignes mettent en œuvre la consistance d'arcs en tant que telle. Toute la partie précédente de l'algorithme vise à "préparer le terrain" en structurant les données de façon ad hoc. Cet algorithme est de complexité optimale (ku^2, avec k = #K et u = #U[1]). Sa complexité minimale vient justement d'un bon choix des structures de données. Les deux structures cruciales sont M et N.

Le lecteur pourra se reporter à l'article pré-cité s'il a du mal à se convaincre que l'algorithme est correct.

1.3 Spécification en Cogito de l'exemple

Le lecteur pourra constater la différence entre le programme ErgoAlg qui précède et la spécification Cogito du même programme. Pour autant, le générateur de programmes Cogito n'est pas encore capable de regénérer ArcCons, nous nous y employons.

Modèle conceptuel :

Arc :: Variable x Variable
Contrainte :: Variable x Variable x Objet x Objet
Domaine :: Variable x Objet

(Arc, Contrainte et Domaine sont des associations).

Spécification des traitements :

ArcCons : Set(Variable) x Set(Arc) x Set(Contrainte) --> Set(Domaine)
(V,K,C) -->
Propage(InitDom(V,K,C),K,C)

Propage : Set(Domaine) x Set(Arc) x Set(Contrainte) --> Set(Domaine)
(D,K,C) -->

soit dD = {Domaine(v,a) ∈ D | ∃ w Arc(v,w) ∈ K et
∀ b Domaine(w,b) ∈ D ==> Contrainte(v,w,a,b) ∉ C}

Si dD = Ø alors D
Sinon Propage(D - dD,K,C)

InitDom : Set(Variable) x Set(Arc) x Set(Contrainte) --> Set(Domaine)
(V,K,C) -->
{Domaine(v,a) | v ∈ Variable et ∃ w Arc(v,w) ∈ K et
∃ b Contrainte(v,w,a,b) ∈ C}

1.4 Présentation de ce qui suit

[1] #E désigne le cardinal d'un ensemble E.

Les parties 2, 3, et 4 présentent ErgoAlg avec assez de détails, mais sans prétention d'exhaustivité. Nous visons en fait à faire saisir par le lecteur les aspects techniques essentiels, dont certains ne sont pas "faciles". Autant que faire se peut, nous nous appuierons sur l'exemple qui précède. La partie 5 précisera certains points.

2 Déclaration d'un programme ErgoAlg

La déclaration du programme ErgoAlg ArcCons est

ArcCons(V↓,K↓,C↓,U↓,D↑)

V, K, C, U et D sont les variables passées en paramètre. La flèche vers le haut ou le bas après chaque variable indique si la variable est passée en *entrée* ou en *sortie* du programme. On pourrait avoir les deux flèches, auquel cas la variable serait passée en entrée *et* en sortie. On a donc trois cas possibles.

Une variable est déclarée en *entrée seulement* si et seulement si elle n'est pas modifiée (par affectation ou effet de bord) dans le programme. Ainsi, on pourra vérifier que les variables V, K, C et U ne sont pas modifiées dans ArcCons. Une variable est déclarée en *sortie seulement* ssi elle est modifiée (et renvoyée à l'extérieur comme résultat) par le programme, mais que la procédure appelante n'a pas initialisé cette variable. C'est le cas de la variable D, qui est construite puis modifiée dans le programme. Une variable en *entrée et sortie* (il n'y en a pas dans l'exemple) devrait être correctement initialisée en entrée de programme, et verrait sa valeur modifiée en sortie. Par ailleurs, les modifications d'un paramètre par le programme sont reflétées sur les variables de l'appel.

Ces notions d'entrée et de sortie sont différentes de ce que l'on trouve dans des langages comme Pascal. En Pascal, on peut passer une variable par *valeur* ou par *adresse*. Le premier cas correspond à une copie de la valeur de la variable, et les modifications subies par le paramètre n'affecteront pas la variable mentionnée dans l'appel. Ainsi, si l'on considère l'appel f(v), que v vaut 3 à l'appel, et qu'il est passé par valeur, même si le programme

```
procedure f(x : integer) ;
```

modifie dans son corps la valeur de x, v vaudra toujours 3 après l'appel de f. Dans ErgoAlg les choses sont fort différentes : définir un paramètre de programme en *entrée seulement* signifie que l'on s'engage à ce que le programme ne modifie pas la valeur du paramètre. Cela pourrait peut-être être vérifié ou prouvé à la seule vue du programme, mais ce n'est pas évident (et même indécidable en général).

A l'opposé, l'appel par adresse en Pascal signifie que les modifications subies par le paramètre dans le corps du programme appelé sont reflétées sur la variable appelante. Cela correspond plus au cas *sortie* ou *entrée et sortie* en ErgoAlg, quoique, dans la traduction en Sum qui sera effectuée, il n'est pas sûr que l'on n'effectuera pas de copies des valeurs (et Sum lui-même traitera différemment la génération en C ou en

Fortran : C ne connaît que le passage par valeur, et Fortran que le passage par adresse).

Pour résumer, les déclarations *entrée* et *sortie* en ErgoAlg ne sont pas analogues aux passages par adresse et valeur. Ils correspondent à un "engagement" du programmeur (ou plutôt du synthétiseur de programmes ErgoAlg Cogito) sur la modification ou non des valeurs des variables par le programme. C'est une *méta-information*[2].

Enfin, le programme ArcCons ici défini est une procédure, au sens où il n'est pas une fonction. Une fonction renverrait une valeur non mentionnée comme paramètre dans la déclaration. Procédures et fonctions sont admises en ErgoAlg.

3 Variables et types

3.1 Déclaration des variables

Un certain nombre de variables du programme ArcCons sont déclarées avant que son corps ne soit défini. Ces déclarations mentionnent les *variables utilisées* dans le programme, et définissent leurs *types* respectifs.

```
V : Set(Variable)          /* L'ensemble des variables du CSP */
K : Set(V x V)             /* Le graphe de contraintes */
C : K --> Set(U x U)       /* Les contraintes */
G : V --> Set(V)           /* Les successeurs du graphe de contraintes */
U : Set(Objet)             /* L'univers des objets considérés */
D : V --> Set(U)           /* Les domaines des variables */
M : V x U --> Set(V x U)   /* Les contraintes indexées par (var,obj) */
P : List(V x U)            /* Liste des couples (var,obj) à propager */
E : V x U --> {0,1}        /* Couples (var,obj) déjà ôtés */
N : V x V x U --> Entier   /* Compteur (var,var,obj) des contraintes
                              restantes */
v, w : V
a, b : U
```

3.2 Variables ErgoAlg

Certaines déclarations ne font que définir le *type des paramètres* du programme, d'autres définissent d'autres variables, qui sont *locales* à la procédure. Le mélange des deux déclarations ne peut prêter à confusion.

Les variables locales sont bien locales au programme. Elles ne peuvent être partagées avec d'autres programmes ErgoAlg. Par contre, on considère en ErgoAlg des variables globales, qui sont connues de tous les programmes ErgoAlg que l'on peut être conduit à prendre en compte. Il y a donc trois types de variables : les variables paramètres, les variables *vraiment* locales, et les variables *vraiment* globales. Les

[2] Celle-ci pourra d'ailleurs servir à d'autres tâches que la synthèse de programme, comme de vérifier qu'un paramètre en *entrée* est bien initialisé avant son appel.

paramètres sont en fait des variables locales particulières. ArcCons ne comporte pas de variable globale.

3.3 Types ErgoAlg

Le lecteur doit avoir une première idée en tête pour bien comprendre notre conception des types ErgoAlg : *les types ErgoAlg sont des ensembles* d'objets qui ne se modifient pas le temps de leur utilisation durant l'exécution du programme. Certains de ces ensembles sont finis, d'autres infinis. Réciproquement, on verra qu'un ensemble présent dans les données d'un programme ErgoAlg (nécessairement fini !) peut y être utilisé comme type sous certaines conditions de stabilité. Etant donc des ensembles presque “normaux”, les types peuvent être utilisés à peu près partout où un ensemble peut l'être (sauf dans certains cas pour les types infinis). On pourra par exemple tester l'appartenance d'une quantité à un type.

Vus d'un point de vue plus classique, les types sont soit des types "élémentaires", soit construits à partir de constructeurs connus. Nous présentons d'abord les constructeurs de types, puis les types élémentaires. Nous inversons ainsi l'ordre généralement suivi dans la présentation d'un langage de programmation.

3.4 Constructeurs de types ErgoAlg

Les constructeurs de type de ErgoAlg sont la définition d'ensemble en extension {…}, le produit cartésien x, la réunion de types U, la flèche de "fonction-données" -->, le constructeur de pointeurs *, et les constructeurs "ensemble de" et "liste de" Set et List. Il vaut mieux ne pas chercher d'analogie directe avec les types des langages de programmation structurés, comme les enregistrements ou les tableaux. C'est justement le travail d'ErgoAlg de faire ces liens.

Un ensemble en extension est constitué de constantes de n'importe quel type. Un produit cartésien est aisément compréhensible. La construction Set(t) est le type des ensembles constitués d'éléments x de type t. De même, List(t) est le type des listes dont les éléments sont de type t.

*t est l'ensemble des *x où x est élément de t. Le constructeur de pointeur * construit ce qui ressemble fort à des pointeurs dans les langages de programmation habituels. Nous préciserons sa définition et surtout son utilisation au §4.3.2, qui est très liée à la sémantique de l'affectation. Le lecteur pourra suivre les explications qui suivent en se référant à sa seule intuition de ce qu'est un type pointeur.

Les deux autres constructeurs (la flèche --> et la réunion U) nécessitent des explications plus longues, et nous les présentons dans les paragraphes qui suivent.

3.5 Index

3.5.1 Définition

La flèche symbolise une *fonction*, ou un *index*[3]. Une flèche t_1 --> t_2 peut être vue comme un *tableau* qui, à tout élément de type t_1, associe *au plus un* élément de type t_2. La différence majeure avec un "tableau" d'un langage de programmation est que l'ensemble des indices du "tableau" n'est pas déterminé à l'avance. L'ensemble des éléments de type t_1 auxquels on a effectivement associé un élément de type t_2 peut d'ailleurs varier au cours de l'exécution du programme.

Une autre manière de comprendre ce que représente un index dans ErgoAlg est de considérer le type conceptuel de Cogito dont il peut provenir. On peut par exemple avoir dans Cogito un ensemble de couples E de type t_1 x t_2, que l'on décide, pour des raisons d'efficacité, de représenter physiquement par la "fonction" $x \in t_1$ --> $\{y \in t_2 \mid (x,y) \in E\}$. Cette "fonction" est justement dans ErgoAlg un index de type t_1 --> Set(t_2)[4]. Autrement dit, un index peut provenir de ce qui, conceptuellement, est un ensemble de couples, mais dont la représentation physique est "orientée" dans un certain sens. De la même manière, une association A du modèle conceptuel sur t_1 x t_2 pourra correspondre dans ErgoAlg à un index t_1 --> Set(t_2). Enfin, on peut associer un index dans un sens ou dans l'autre, voire les deux si la redondance est souhaitable.

Cependant, pour ErgoAlg, un index est différent d'un ensemble de couples, et les opérateurs qui pourront être appliqués à un élément de ce type index dans le corps du programme seront différents de ceux applicables à un produit cartésien.

L'utilisation essentielle d'un index est similaire à celle d'un tableau. Si v_1 est de type t_1, et C de type t_1 --> t_2, alors C(v_1) est (l'unique) v_2 tel que C(v_1) = v_2. D'où son nom d'*index*, qui, à tout élément de t_1 associe un élément de t_2.

Attention ! Lorsque nous disons que C est analogue à un *tableau*, cela ne signifie pas que l'index sera représenté physiquement par un tableau dans le programme généré à partir du programme ErgoAlg. Cela peut être vrai, mais ErgoAlg peut aussi décider de représenter physiquement un index d'une autre manière (notamment lorsque le type t_1 n'est pas borné supérieurement par quelque chose de connu à l'avance et de "pas trop gros").

Il existe enfin un autre constructeur de type, particulièrement utile pour construire des index qui "bouclent" (cf. 2.2.4). Ce constructeur, noté ', et nommé "élément de l'index" s'applique à un type index, par exemple

(t_1 --> t_2)'

Il a la signification suivante. Une variable de type

C : t_1 --> t_2

---

[3] La flèche vue comme fonction se rapproche des fonctions de Cogito, la flèche vue comme index se rapproche d'une mise en œuvre physique en Sum.

[4] On peut se ramener dans le bon cas de monovaluation à la fonction qui à x associe l'unique y tq $(x,y) \in E$, qui correspondra dans ErgoAlg à un index de type t_1 --> t_2.

"représente" en quelque sorte un *ensemble* de couples (v_1,v_2) de t_1 x t_2, mais où "l'accès" de v_2 à partir de v_1 est "favorisé". Le type qui représente *un seul* de ces couples sera justement (t_1 --> t_2)'. Ainsi, t_1 --> t_2 est l'analogue de Set(t_1 x t_2), et (t_1 --> t_2)' celui de t_1 x t_2.

3.5.2 Dénomination des types et index "bouclants"

Un type peut être nommé par un identificateur, par exemple

Ind = E --> Set(F)

Cela, allié aux types "élément de l'index", est particulièrement utile lorsque l'on veut avoir des index qui "bouclent", comme par exemple :

Ind1 = E --> Ind2
Ind2 : F --> *(Ind1')

P : Ind1

On peut se faire l'image suivante (cf. figure 2) de cette structure de données. P "contient" une sorte d'ensemble des éléments de E, mais où chaque élément e de E "pointe" aussi sur un ensemble d'éléments de F. De plus, on a l'intention que chaque élément f de F ainsi pointé par e pointe lui-même sur l'élément de l'index P(e) qui lui a "donné naissance"[5].

[5] En toute rigueur, la définition du type oblige f à pointer sur un élément d'index de type Ind1, et donc pas nécessairement sur "celui qui lui a donné naissance". L'exemple que nous décrivons ici constitue un cas particulier d'utilisation des index croisés.

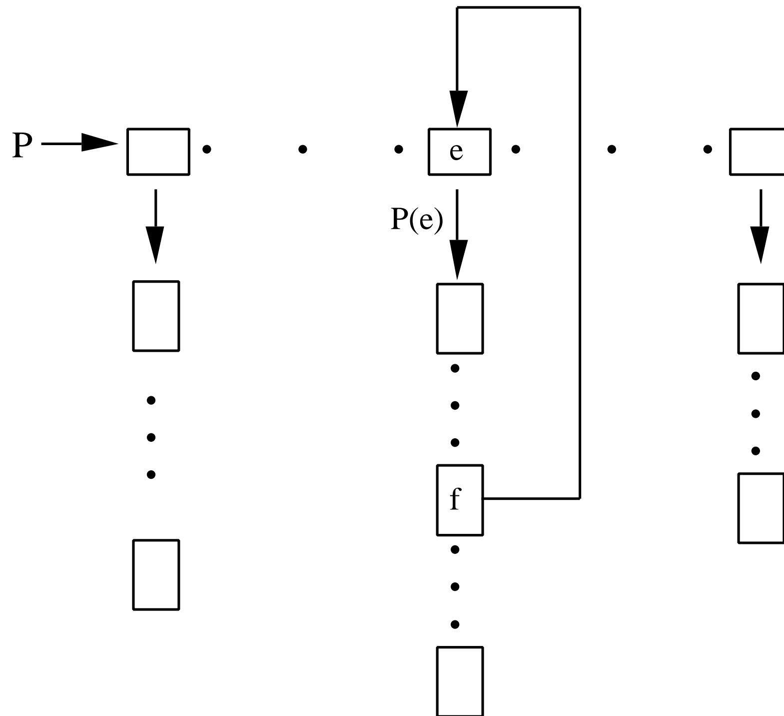

Figure 2 : Structure de données par index bouclants

Ce genre "d'index croisé" est particulièrement efficace lorsque la représentation d'un élément de e consomme beaucoup de mémoire, car ce qui est mémorisé au niveau de f est une sorte de pointeur sur e. De plus, il permet à f de "retrouver directement l'ensemble auquel il appartient".

3.6 Réunion de types

Un type peut être réunion de deux ou plus de types, même "hétérogènes" (dans un sens que nous définissons ensuite). Cela est particulièrement utile pour définir des types "récursifs", comme par exemple :

Doublet = (*Doublet x *Doublet) ∪ Atome

Nota : L'expression précédente utilise encore la dénomination d’un type, ce qui est indispensable à la construction de types récursifs.

Le lecteur averti aura reconnu une définition de type ErgoAlg redéfinissant la notion de doublet Lisp. Un doublet est soit un couple de pointeurs vers un doublet, soit un atome.

On peut aussi donner une définition d'une liste Lisp utilisant l'opérateur ErgoAlg List :

ListeLisp = List(ExprLisp)
ExprLisp = ListeLisp ∪ Atome

Ainsi, une liste Lisp est une liste dont les éléments sont des listes Lisp ou des atomes.

Il est à noter que cette utilisation de la "réunion récursive" est indispensable dans ce cas. En effet, dans ErgoAlg, un type List(t) contient des listes dont les éléments sont tous de type t. Si l'on veut des listes "de profondeur quelconque", l'usage de la récursivité dans la définition de t est indispensable, et donc aussi l'arrêt de cette récursivité sur un type "feuille", d'où la réunion de types.

3.7 Types élémentaires

Nous n'avons pas encore défini les types "de base" à partir desquels, en utilisant les constructeurs, on peut définir n'importe quel type.

Les types de base contiennent tout d'abord les types prédéfinis, à savoir Entier, Reel, Caractere et Booleen.

Une deuxième sorte de type de base est constituée des types nommés et construits à partir des types prédéfinis ou de type nommés de la même sorte. Dans l'exemple, le type Variable est de cette sorte. Cela n'est pas précisé dans le sous-programme ArcCons, mais ce type doit être défini pour l'ensemble de programmes contenant ArcCons, par exemple comme Variable = List(Caractère). De la même manière, on a pu définir Objet = List(Caractère). Cependant, les deux types Variable et Objet sont considérés comme distincts, et même d'intersection vide : une variable n'est pas un objet, de même réciproquement.

Les types prédéfinis et de la seconde sorte précédente sont collectivement nommés *types génériques.* La propriété essentielle d'un type générique est qu'il constitue un ensemble (fini ou infini) parfaitement déterminé quelles que soient les données sur lesquelles sera exécuté le programme ErgoAlg.

Les types sont donc considérés comme des ensembles (infinis, comme Entier, ou finis, comme Caractere). Cela nous permet d'étendre la notion de type à une troisième sorte de type élémentaire : *un ensemble passé en paramètre d'une fonction ErgoAlg peut sous certaines conditions y être utilisé comme type.* Nous détaillons dans ce qui suit ce point très important.

On a l'habitude dans les langages classiques de voir les types complètement séparés des variables et des données[6]. Or, ici, une variable peut *aussi* se voir attribuer un rôle de type, comme dans

K : Set(V x V)

Que peut bien signifier cela ? Ici, la variable V a été déclarée en *entrée seulement*, et donc ne changera pas de valeur dans le programme. C'est aussi un ensemble. Or, K a pour vocation d'être un ensemble de couples de variables de V. La déclaration

K : Set(Variable x Variable)

---

[6] Quoique les *classes* d'un langage objet (qui sont comme des types) soient souvent aussi considérées comme des objets.

serait donc moins précise, car Variable est un ensemble qui peut être beaucoup plus grand que V. Avec la déclaration

K : Set(VxV)

on sait que K ne contient que des couples d'éléments de V, et le fait que V ne change pas pendant l'exécution de ArcCons rend cette définition aisément cohérente[7].

Les autres déclarations utilisent cette possibilité, éventuellement de façon récurrente. Par exemple, on a

C : K --> Set(U x U)

Rien n'interdit d'avoir des circuits dans l'utilisation de variables de type ensemble comme types[8].

Les variables qui "vont bouger" dans le programme peuvent aussi être définies de cette façon, comme

D : V --> Set(U)
M : V x U --> Set(V x U)

Ici, D est un paramètre de ArcCons en *sortie seulement*, et M une variable locale, dont les valeurs changent lorsque le corps de la procédure est exécuté. Le fait d'utiliser des variables dans leur définition de type correspond à un engagement du programmeur sur les limites que peuvent prendre leurs valeurs. Cela est particulièrement utile au générateur de programme ErgoAlg pour choisir les structures de données physiques par lesquelles les représenter (il peut par exemple dans certains cas utiliser des structures de données limitées a priori en taille, comme des tableaux).

3.8 Types génériques, sous-types et compatibilité de types

Les trois paragraphes qui suivent définissent des relations, “type générique” et “sous-type”, entre types d’un programme ErgoAlg. Ces relations sont construites à partir de la seule donnée du programme ErgoAlg (l’utilisateur n’a pas à les exprimer). Elles permettent de définir la compatibilité entre types. Nous supposons en effet que dans certains cas des quantités de types différents peuvent être affectées dans un programme ErgoAlg.

3.8.1 Type générique

---

[7] La suppression d'un élément de V serait problématique d'un point de vue conceptuel (faudrait-il maintenir K ?), l'ajout serait embêtant pour la synthèse de programme (si K est représenté dans une structure *limitée en taille* par la taille de V).

[8] C'est-à-dire qu'une variable v_1 serve dans la définition du type de v_2, …, que v_{n-1} serve dans celle de v_n, et que v_n serve à son tour dans la définition du type de v_1.

Les définitions précédentes font apparaître deux sortes de types : les types construits à partir de types prédéfinis (Entier, Reel, Caractere, Booleen) et les types utilisant dans leur définition un ensemble passé en paramètre d'une procédure. Les premiers sont appelés *types génériques*.

Nous définissons une relation, nommée aussi type générique, et notée TG, qui associe à tout type, un type générique. Cette relation est étendue aux variables, le type générique d'une variable étant le type générique de son type. Considérons d'abord l'exemple. Les déclarations de types :

```
V : Set(Variable)            /* L'ensemble des variables du CSP */
K : Set(V x V)          /* Le graphe de contraintes */
C : K --> Set(U x U)         /* Les contraintes */
G : V --> Set(V)             /* Les successeurs du graphe de
                                contraintes */
D : V --> Set(U)             /* Les domaines des variables */
U : Set(Objet)          /* L'univers des objets considérés */
M : V x U --> Set(V x U)     /* Les contraintes indexées par
                                (var,obj) */
P : List(V x U)              /* Liste des couples (var,obj) à
                                propager */
E : V x U --> {0,1}          /* Couples (var,obj) déjà ôtés */
N : V x V x U --> Entier     /* Compteur (var,var,obj) des
                                contraintes restantes */
v, w : V
a, b : U
```

permettent de définir les types génériques de V, K, C, G, etc, de la façon suivante :

TG(V) = Set(Variable)
TG(K) = Set(Variable x Variable)
TG(C) = Set(Variable x Variable) --> Set(Objet x Objet)
TG(G) = Variable --> Set(Variable)
TG(D) = Variable --> Set(Objet)
TG(U) = Set(Objet)
TG(M) = Variable x Objet --> Set(Variable x Objet)
TG(P) = List(Variable x Objet)
TG(E) = Variable x Objet --> {0,1}
TG(N) = Variable x Variable x Objet --> Entier
TG(v) = TG(w) = Variable
TG(a) = TG(b) = Objet

Par ailleurs, si on a défini Variable et Objet par

Variable = List(Caractere)
Objet = List(Caractere)

alors on a

TG(Variable) = TG(Objet) = List(Caractere)

Moralement, le type générique d'un type ou d'une variable est donc sa "structure informatique essentielle". Le type générique correspond beaucoup plus au type habituellement rencontré dans les langages informatiques.

Une tentative de fonction permettant de définir TG(t) pour un type t est la suivante :

TG(t)

1/ Initialiser TG(t) à t

2/ Si TG(t) mentionne un type non générique u, et si le type de u a (nécessairement) la forme Set(tu), alors remplacer u par tu dans l'expression de t. Sinon arrêter.

3/ Aller en 2/.

Le problème de cette procédure est qu'elle n'est correcte que si "on ne boucle pas", i.e. ssi on ne rentre pas dans un circuit de mentions de types non génériques : u_1 mentionne u_2, …, u_{n-1} mentionne u_n, u_n mentionne u_1. Cela se passe par exemple si l'on a une procédure ErgoAlg ayant deux paramètres U et V en entrée seulement, et dont les types sont définis par

U : Set(*V)
V : Set(U)

Dans ce cas, l'application de la règle conduit à l'expression Set(*U). En étant arrivé là, on décide de créer un nouveau type générique AbstrU avec

AbstrU = Set(*AbstrU)

et on définit

TG(U) = AbstrU
TG(V) = AbstrU

Quelle différence entre U et AbstrU ? U, étant un type non générique, est un paramètre de procédure ErgoAlg, et donc un ensemble fini. AbstrU, par contre, est un ensemble infini. Les définitions des types de U et V est donc plus précise que la donnée de leurs types génériques.

Pour aboutir à ce résultat, la procédure doit donc être transformée en lui adjoignant un ensemble TS des types déjà substitués, et en modifiant son corps de la façon suivante :

TG(t)

1/ Initialiser TG(t) à t et TS à {t}

2/ Si TG(t) mentionne un type non générique u n'appartenant pas à TS, et si le type de u a (nécessairement) la forme Set(tu), alors remplacer u par tu dans l'expression de t, et ajouter u à TS. Sinon aller en 4/.

3/ Aller en 2/.

4/ Pour tout u de TS-{t}, remplacer u par TG(u)

5/ Si $t \in TS$, construire le type générique de t par substitution de t dans TG(t) par un nouvel identificateur de type.

Remarque 1 : Deux types non génériques distincts peuvent avoir des types génériques créées par l'étape 5/ de la fonction TG isomorphes. Dans ce cas, on identifie les deux types génériques.

Remarque 2 : On définit le type générique d'un ensemble en extension comme la réunion des types génériques de ses éléments.

3.8.2 Sous-type

Fondamentalement, nous considérons les types comme des ensembles. Or, les ensembles sont ordonnés par la relation d'inclusion. Nous allons définir une relation nommée sous-type et notée $\propto$, qui est incluse dans la relation d'inclusion.

On considère que, parmi les types prédéfinis, x est sous-type de x, et que sinon seul Entier est sous-type de Reel :

$$\text{Entier} \propto \text{Reel}$$

Pour les types génériques, on définit $\propto$ par les règles :

s et t ensembles en extension et $t \supseteq s$ $\Rightarrow$ $s \propto t$
s ensemble en extension $\Rightarrow$
$s \propto$ l'union des types des éléments de s
$s \propto t \Rightarrow *s \propto *t$
$s \propto t \Rightarrow \text{Set}(s) \propto \text{Set}(t)$
$s \propto t \Rightarrow \text{List}(s) \propto \text{List}(t)$
$s_1 \propto t_1$ et $s_2 \propto t_2 \Rightarrow s_1 \text{-->} s_2 \propto t_1 \text{-->} t_2$
$s_1 \propto t_1$ et … et $s_n \propto t_n \Rightarrow s_1 \cup \ldots \cup s_n \propto t_1 \cup \ldots \cup t_n$
$s_1 \propto t_1$ et … et $s_n \propto t_n \Rightarrow s_1 \text{ x } \ldots \text{ x } s_n \propto t_1 \text{ x } \ldots \text{ x } t_n$
n=2 et $s_1 \text{ x } \ldots \text{ x } s_n \propto \text{List}(s_1 \cup \ldots \cup s_n)$

Le dernière ligne signifie que l'on identifie les tuples avec des listes de type ad hoc.

De plus, on étend $\propto$ par “maximisation du point fixe” pour les types construits à partir d’eux-mêmes, comme Doublet = (*Doublet x *Doublet) $\cup$ Atome. Cela signifie, par exemple, que ce type est sur-type du type DouEnt = (*DoubEnt x *DoubEnt) $\cup$ Entier dès que Atome est sur-type de Entier. Cette définition est similaire à celle donnée dans le langage Sum [Dormoy, 1993a].

Enfin, pour les types non génériques, on a :

$$TG(s) \propto t \Rightarrow s \propto t$$

On constate donc que la relation sous-type ne mentionne jamais à droite un type non générique. Pour étendre cette relation, on pourrait considérer des inclusions entre paramètres d'une même fonction. Mais il faudrait alors connaître ces inclusions a priori, car les retrouver par analyse du programme s'avère difficile ou impossible.

3.8.3 Compatibilité de types

Bien que l'affectation soit détaillée dans le §4.3, nous présentons tout de suite la motivation essentielle des définitions de type générique et de sous-type. Nous considérons en effet que l'on peut étendre l'affectation (et ses analogues, comme le passage de paramètres à une procédure ErgoAlg) à des opérandes de types différents sous certaines conditions.

Une affectation

```
x <-- y
```

est admise pourvu que l'un des types génériques tgx et tgy de x et y soit sous-type de l'autre.

Si tgy est sous-type de tgx, alors l'affectation est toujours possible. Si tgy est sur-type de tgx, alors l'affectation est aussi possible, mais il est clair qu'elle pourra perdre tout sens si la valeur de y n'est pas de type tgx. Dans ce cas, le programme généré par ErgoAlg pourra produire une erreur. C'est au programmeur de s'assurer qu'un tel cas ne peut se produire.

4 Instructions

Le programme exemple mentionne essentiellement trois sortes d'instructions : des affectations, des tests (Si … alors … Sinon …), et des boucles. Nous décrivons d'abord les deux dernières sortes d'instructions, car les affectations sont les plus complexes.

4.1 Tests

Les tests sont tout à fait classiquement de la forme Si … alors … Sinon …. On peut envisager aussi un case of, comme en Pascal. Ils ne posent pas de problème.

4.2 Boucles

On a dans l'exemple deux sortes de boucles : parcours d'un ensemble, et boucle "Tant que …".

Un parcours d'ensemble a la forme

```
Pour tout w de G(v) faire
        …
```

L'ensemble parcouru (ici G(v)) doit rester fixe pendant le parcours de la boucle (sinon on ne sait pas très bien que faire d'un élément de l'ensemble traité puis ôté, ou d'un élément de l'ensemble ajouté). D'ailleurs, faire autrement introduirait une sorte de récursivité, qui doit justement être éliminée par Cogito. Le programmeur s'engage donc à ce que cette condition soit respectée. L'ordre de parcours de l'ensemble n'est évidemment pas défini. Le programmeur s'engage donc aussi à ce que le résultat du programme ne dépende pas de cet ordre[9].

Les boucles *Tant que*, quant à elles, ont la sémantique habituelle.

4.3 Affectation

4.3.1 Affectation par copie

On a tout d'abord l'affectation "normale", notée <--, par exemple

```
G(v) <-- ø
N(v,w,a) <-- 0
```

Cette affectation agit *par copie*, c'est-à-dire que la valeur de la partie droite est copiée, puis affectée comme valeur de la partie gauche.

En particulier, si l'on considère l'instruction

```
E <-- F
```

où E et F sont deux ensembles, alors F est *recopié* dans E[10].

4.3.2 Affectation par adresse et pointeurs

Même si aucune instruction ne semble dans l'exemple mentionner le cas qui va suivre, on pourrait vouloir ne pas copier la valeur de la partie droite dans la partie gauche, mais affecter son "adresse", en particulier lorsque les types des opérandes sont complexes. Pour faire comprendre cela, considérons à nouveau une affectation

```
E <-- F
```

---

[9] De manière plus subtile, le résultat pourrait dépendre de l'ordre, comme dans le programme

```
C : E --> Entier
Compteur <-- 0
Pour tout a de E faire
        C(a) <-- Compteur
```

qui numérote les éléments de E via un index C. Ici, la numérotation dépend de l'ordre du parcours, mais *le programmeur doit considérer n'importe quelle numérotation comme correcte*.

[10] Cela signifie que l'on constitue un "nouvel" ensemble (c'est-à-dire mémorisé à un "autre endroit") dont les éléments sont exactement ceux de F.

où E et F sont des ensembles du même type. Vue notre définition de l'affectation, cette instruction signifie que l'on recopie F avant de l'affecter à E. La conséquence de cela est que, si E est ensuite modifié, la modification ne sera pas reflétée par effet de bord sur F. Mais il est clair que, dans toute implantation physique de cela, le programme va consommer une quantité importante de mémoire.

Or, il se pourrait que l'on se moque (voire que l'on souhaite, ou que l'on sache qu'ils n'interviennent pas) des effets de bord que pourrait impliquer une affectation par adresse.

Pour pouvoir définir cela, on introduit un nouveau constructeur de type, appelé *pointeur*, et noté *, et deux opérateurs utilisables dans les instructions, le *déréférencement* * et le *référencement* &[11].

Ainsi, si l'on remplace l'affectation précédente par

    &E <-- &F

alors l'affectation va copier *l'adresse* de F (sa *référence*) dans *l'adresse* de E. E et F deviennent ainsi *physiquement identiques*. Si ensuite l'un est modifié, l'autre l'est aussi.

Quel est le type de l'expression &E ? Si E est du type Set(A), &E est une expression du type *pointeur vers Set(A)*. Ce type se note

    *Set(A)

Mais d'autres notations sont aussi possibles. Au lieu d'utiliser E et F, on aurait pu utiliser à leur place des variables pE et pF (pointeurs vers E et F), déclarées par

    pE, pF : *Set(A)

et remplacer l'affectation

    &E <-- &F

par

    pE <-- pF

Si maintenant, nous voulons effectuer une affectation par copie (et non par adresse), alors que nous ne disposons plus que de pE et pF, nous écrirons

    *pE <-- *pF

En effet, si on a (dans notre tête, ou dans le programme) pE = &E, alors on a de manière équivalente *pE = E. ** et & sont inverses l'un de l'autre.*

---

[11] Ces notations sont inspirées de C.

Nous pouvons maintenant pleinement comprendre le déroulement exact de l'instruction

E <-- F

On peut voir cette instruction comme s'exécutant en deux étapes :
- constituer un ensemble résultat de l'opérande droite mais sans décider de sa référence;
- donner comme référence à cet ensemble la référence de E.

Ainsi, la référence de E reste inchangée. Cette "vision" de l'affectation va être particulièrement importante pour comprendre l'effet des opérateurs ensemblistes.

Pour résumer, toute variable ErgoAlg a deux "aspects" : sa valeur, et sa référence. On peut voir sa référence comme l'adresse de la zone de mémoire physique où elle est stockée. Toutefois, cela ne doit être pris qu'en un sens métaphorique, car ErgoAlg peut mémoriser les données selon de multiples possibilités. Cette notion de pointeur sera parfaitement claire d'emblée à tout lecteur familier de C.

Remarques :

- Il est a priori tout à fait possible de prendre le pointeur d'un pointeur (quoiqu'à notre avis on puisse s'en passer).
- &E peut être vue comme une nouvelle variable, qui a une référence et un contenu. Mais c'est une variable un peu particulière, car son contenu est initialisé à la référence de E.
- En fait, il est de mauvaise économie que d'affecter une valeur à &E. En effet, l'instruction &E <-- &F "perd" la référence de E, mais aussi la "place mémoire" qui a été allouée implicitement par la déclaration de E (cf. §4.5). Il est donc préférable d'utiliser des variables pointeurs, et l'instruction pE <-- pF. Par contre, l'utilisation de &E lorsqu'elle n'est pas affectée ne pose pas de problème.

4.3.3 Opérateurs et affectation

Aux différents types sont associés des opérateurs qui peuvent être utilisés dans les instructions. Par exemple, pour les ensembles, on a la réunion, l'intersection, la différence, etc. Dans ErgoAlg, ces opérateurs sont présents, mais vont être en fait "exécutés" de diverses manières. Si notre définition de l'affectation est *logiquement* correcte, elle va pouvoir être mise en œuvre de différentes manières selon les cas.

Prenons par exemple l'instruction

E <-- F $\cup$ G

que l'on lit "affecter à E l'union de F et de G". Cette instruction a la signification suivante :

"Constituer un ensemble, dont les éléments sont exactement ceux de F ou ceux de G, et donner à cet ensemble la référence de E".

Donc, cette instruction comporte une *copie* des éléments de F et G. F et G ne sont pas modifiés par l'exécution de cette instruction. Il est important de noter que la copie *vient de l'utilisation de l'opérateur* U.

Or, souvent, on a l'intention de modifier un ensemble E en lui ajoutant les éléments d'un ensemble F. On écrit alors l'instruction

E <-- E ∪ F

A priori, si l'on suit la spécification de l'affectation, les éléments de E et de F sont copiés dans un ensemble, et le résultat se voit donner la référence de E. Si ErgoAlg exécutait bien physiquement l'instruction de cette manière, on aurait donc "perdu" la place mémoire occupée par E avant l'affectation. De plus, cette opération serait d'autant plus longue que le cardinal de E serait grand.

Or, après avoir rempli par "copie" l'ensemble E ∪ F, ErgoAlg doit lui donner la référence de E. Il serait donc plus simple de modifier le contenu de l'ensemble E, en lui ajoutant les éléments de F. C'est ce qu'ErgoAlg va en fait effectuer. Ainsi, la logique de l'instruction est bien respectée, mais il n'y a pas de perte de mémoire.

En définitive, l'instruction

E <-- E ∪ F

peut être vue de façon "naturelle" : on ajoute à E les éléments de F.

De telles instructions sont présentes dans notre exemple, comme

D(v) <-- D(v) ∪ {a}

Il est à noter que la règle de l'affectation définie ici s'applique aussi aux types simples (comme les entiers) :

N(v,w,a) <-- N(v,w,a) + 1

L'affectation a alors le sens habituel. Elle s'applique aussi aux listes :

P <-- (v,a).P

"." est l'opérateur de concaténation. Cette instruction revient à ajouter (v,a) en tête de la liste P[12]. On a aussi

---

[12] Ami lispien, attention ! Cette instruction ne change pas la référence de P. Cette référence de P ne se confond donc pas avec l'adresse du *doublet* racine de l'arbre qui représenterait physiquement la liste. En fait, il n'y a pas de doublet *du tout* dans ErgoAlg. Une liste peut être représentée de bien

P <-- Fin(P)

qui remplace P par sa fin (i.e. on lui enlève son premier élément).

4.4 Opérateurs ensemblistes contraints

On rencontre aussi dans l'exemple des instructions du genre

G(v) <-- G(v) + {w}

Ici, "+" est la réunion, mais avec la connaissance supplémentaire que les éléments du terme à droite de "+" n'apparaissent pas dans l'ensemble à gauche de "+".

Donc, on a

E <-- F + G

si, et seulement si, F et G sont *disjoints*. Il s'agit en fait d'une méta-information, qui sera utilisée par ErgoAlg pour savoir qu'il est inutile de chercher les éléments communs à F et G afin de ne pas créer de "doublons" dans la représentation physique de $F \cup G$. Il est clair que l'utilisation du "+" ensembliste correspond à un engagement du programmeur (il n'est pas vérifié), et qu'il est de l'intérêt de l'efficacité du programme généré d'utiliser un "+" à la place d'une réunion ordinaire à chaque fois que cela est possible.

4.5 Références et allocation de mémoire

Il peut paraître contradictoire de parler d'allocation de mémoire pour ErgoAlg, puisque, lorsque l'on écrit un programme dans ce langage, on ne sait pas comment les données vont être physiquement représentées.

Chaque déclaration de variable conduit à la *création conjointe* d'une référence et de la place mémoire nécessaire au “rangement du contenu” de la variable. Si l'on considère les variables

a : E x F
H : Set(E)

où E et F sont des ensembles, la "place mémoire" occupée par le contenu de a sera toujours de deux unités, celle occupée par H sera indéfinie (éventuellement bornée supérieurement si l'on connaît une borne supérieure du cardinal de E).

---

des manières, et ErgoAlg a (ou aura) une certaine expertise en la matière.

Cependant, cette création n'est effectuée que pour chaque variable déclarée, ce qui peut conduire à devoir allouer explicitement de la place mémoire dans lun programme ErgoAlg. Ainsi, si l'on considère

pH : *Set(E)

la place mémoire nécessaire au contenu de *pH *n'est pas* "créée" par ErgoAlg. Cela pose problème si l'on entend utiliser l'instruction

*pH <-- H

Moralement, donc, une déclaration de variable crée sa référence et la "place mémoire" nécessaire pour stocker sa valeur. Mais si la variable est de type pointeur, la valeur de la variable est une référence, mais la "place mémoire" nécessaire au stockage des valeurs à cette référence n'est pas créée. Pour résumer de façon courte, "l'allocation de mémoire par déclaration de variable s'arrête aux pointeurs".

Si nous reprenons l'exemple de notre index "bouclant" :

Ind1 = E --> Ind2
Ind2 : F --> *(Ind1')

P : Ind1

la déclaration de P va créer l'allocation de la place pour les éléments e de E, pour chacun de ces e la place pour les f, mais pour chacun de ces f *seulement la place pour une référence à un élément de type* Ind1'. D'ailleurs, si l'opérateur * avait été omis, P nécessiterait une place mémoire infinie. Plus généralement, on a la contrainte : *tout circuit dans la définition d'un type doit passer par un constructeur* *.

Pour créer "de la place", plusieurs possibilités. Soit on utilise une variable déclarée, comme dans

pH : *Set(E)
H : Set(E)

pH <-- &H

Sinon, on peut allouer dynamiquement de la "place mémoire" par une fonction spéciale, notée simplement New :

pH : *Set(E)

pH <-- New(Set(E))

New est une fonction qui prend un type en argument, et renvoie une *référence* sur ce type. On peut ainsi créer autant de références que l'on veut, qui seront mémorisées dans les collections (ensembles, listes, index) valeurs des variables déclarées. New possède dans ErgoAlg sa fonction duale habituelle : Dispose. L'instruction

```
Dispose(X)
```

supprime la référence de la variable X, et bien sûr son contenu.

Considérons un autre exemple, mentionnant un index bouclant légèrement modifié par rapport au précédent :

```
Ind1 = E --> *Ind2
Ind2 : F --> *(Ind1')

P : Ind1
```

La déclaration de P n'implique plus maintenant que l'allocation de la place nécessaire aux éléments e de E, plus pour chaque e la *place pour une référence* vers un index de type Ind2. Si l'on veut y stocker une référence qui "a un sens", on pourra utiliser New, comme par exemple dans

```
Pour tout e de X faire
    P(e) <-- New(Ind2)
```

En définitive, on a donc, via le constructeur *, toutes les ressources habituellement fournies par les pointeurs en C, Pascal ou Lisp[13]. Cependant, les explications métaphoriques qui précèdent ne doivent pas être prises au pied de la lettre, car ErgoAlg reste maître des problèmes de la "gestion de la mémoire".

4.6 Pointeur *nil* et structures vides

Comme dans les langages de programmation habituels, une valeur particulière de pointeur, notée nil, existe. Cette valeur est compatible avec tous les types pointeurs, *mais seulement avec ces types*.

Les types de "collections" (ensembles, listes, index) n'admettent pas nil comme valeur possible : il ne faut pas confondre les collections avec des pointeurs ! Chacune des trois grandes sortes de collection a sa propre valeur de "collection vide". Pour les ensembles, c'est l'ensemble vide ø, pour les listes la liste vide [], pour les index l'index vide ° (qui est la "fonction" dont le domaine est vide).

4.7 Ajout et suppression d'un élément d'index

Un index C : t_1 --> t_2 est comme une fonction de t_1 dans t_2. Nous utilisons ici le mot *fonction* dans son sens mathématique, qui est différent de celui *d'application* : C associe à tout élément de t_1 *au plus un* élément de t_2 (et non *exactement un* élément de t_2). Cela signifie donc que l'on n'a pas nécessairement une valeur C(x) pour tout élément x de t_1.

---

[13] Quoique la mise en œuvre d'un *garbage collector* dans le programme généré par ErgoAlg ne soit pas envisagé. On préfère miser sur une utilisation judicieuse de l'allocation dynamique par Cogito.

Comme toute "fonction", un index C a un domaine, c'est-à-dire l'ensemble des éléments de l'ensemble de départ ayant au moins une image par C. Cette fonction est disponible dans ErgoAlg, et notée Dom. En notation ensembliste, on a

$$Dom(C) = \{e \in E \mid \exists f \in F \quad C(e) = f\}$$

Le domaine de l'index vide est l'ensemble vide :

Dom(°) = ø

Pour que cette définition ait un sens, il faut définir comment élément d'index peut être ajouté ou ôté à un index.

L'ajout d'un élément à un index se fait par l'affectation. L'instruction

C(x) <-- y

met y comme valeur de C(x). C(x) pouvait déjà exister (i.e. x pouvait être dans le domaine de C), ou ne pas exister avant cette instruction.

Le retrait utilise une fonction ErgoAlg prédéfinie :

Kill(C(x))

Cette instruction va ôter la valeur y qu'avait C(x). x doit appartenir au domaine de C pour que cette instruction s'exécute correctement. Après cette instruction, x n'appartient plus au domaine de C.

Prenons un exemple, celui de l'index bouclant :

Ind1 = E --> Ind2
Ind2 : F --> *(Ind1')

P : Ind1

Si e est dans le domaine de P, l'instruction

Kill(P(e))

va supprimer la valeur i2 de P(e). Mais attention ! Cette valeur, qui est ici du type index Ind2, n'est pas "détruite". Elle peut en effet être valeur d'autres variables ou éléments d'index.

Il est à noter que nous aurions pu simuler la suppression de la valeur de e pour l'index P par l'instruction

P(e) <-- °

Mais cela ne fait que changer la valeur de e à ° (l'index vide), et e appartient encore au domaine de P après cette instruction.

5 Détails de ErgoAlg

La présentation qui précède est suffisante pour comprendre un programme ErgoAlg. Cependant, nous formalisons ici un peu plus ce que l'on a le droit d'utiliser et ce qui est interdit de faire dans ErgoAlg.

5.1 Procédures et fonctions

Nous avons déjà décrit comment déclarer un programme ErgoAlg. Ils sont de deux types : les procédures ne renvoient pas de valeur, les fonctions renvoient une valeur, dont le type est défini dans la déclaration:

    f(…) : t

Lorsque l'on a affaire à une fonction, la valeur est renvoyée par une instruction "return" :

    return v

où v est une variable de type t.

Un programme ErgoAlg peut être appelé par un autre programme, par exemple

    p(…)
    w <-- f(…)

où p est une procédure et f une fonction.

Attention ! Les appels *récursifs* sont *interdits*. Cogito assure en effet la dérécursivation des programmes.

5.2 Instructions

Les instructions ErgoAlg acceptées sont :

- l'affectation
- le test (Si … alors … Sinon …)
- les boucles (parcours d'ensemble et Tant que)
- l'appel de procédure
- return
- break, qui sort de la boucle la plus interne qui le contient
- l'instruction vide (comme *continue* en Fortran), qui n'a aucune action, mais qui est utile pour le synthétiseur de programmes

- l'instruction multiple (un bloc d'instructions)

5.3 Ordre entre instructions

A priori, les instructions d'un programme ErgoAlg sont complètement ordonnées. Toutefois, pour des raisons pratiques de synthèse de programme, l'ordre pourra être défini de façon non standard, à l'instar des instructions d'un programme Sum [Dormoy, 1993a].

5.4 Variables et types

Comme on l'a vu, il y a trois sortes de variables : les variables globales (connues de tous les programmes ErgoAlg considérés), les paramètres de programmes et les variables locales aux programmes (qui ne sont pas des paramètres).

Une variable comporte deux aspects : sa *référence* et son *contenu* (cf. §3).

Les variables ont un type. Les types ont été définis au §3.

Les types sont construits par les constructeurs :

- ensemble en extension {…} (dont les intervalles d'entiers, comme [1,4])
- Set(t) = Ensemble des parties de t
- List(t) = Ensemble des listes de t
- le produit cartésien x, et l'exponentiation xx
- le définisseur d'index -->
- l'instance d'index '
- le constructeur de pointeur *
- la réunion ∪

Certains types sont compatibles. Par exemple, une liste de List(t) ayant 4 éléments est aussi considérée comme étant du type t x t x t x t[14]. Autrement dit, t x t x t x t est exactement l'ensemble des listes à composantes dans t ayant 4 éléments. *Listes et tuples sont compatibles.*

Remarque : On n'a pas associativité des opérateurs x et -->. Ainsi, E x (F x G) et (E x F) x G sont deux types distincts. Par contre, ∪ est associative sur les types.

---

[14] Puisque nous disposons d'un opérateur d'exponentiation, ce type peut aussi être noté t xx 4. Plus généralement, on pourrait considérer un type t_1 xx t_2, où t_2 est un type ensemble. Celui-ci serait alors égal au type index t_2 --> t_1 (car E^F dénote en théorie des ensembles l'ensemble des fonctions de E dans F). Pour retomber sur ses pieds, il faudrait alors considérer t xx 4 comme t xx [1,4]. Ceci serait tout à fait possible, car un quadruplet d'éléments de t peut être vu comme une fonction de [1,4] vers t. De plus, les notations seraient les mêmes pour désigner par exemple la troisième composante d'un élément v de t : v(3). Tout fonctionne donc parfaitement.

Enfin, tout circuit dans la définition d'un type doit passer par un opérateur *.

5.5 Constantes

On peut définir des constantes de n'importe quel type, par exemple,
cc_chaine = ("c","h","a","i","n","e") : List(Caractere)

Les constantes "fonctionnent" comme des variables, mais on n'a pas le droit de changer leur valeur (ni leur référence via &). Une même constante déclarée plusieurs fois correspond à une référence unique. On peut "récupérer" la référence d'une constante par l'opérateur &. Il est de la responsabilité du programmeur de ne pas changer le contenu d'une référence de constante.

5.6 Expressions

Les expressions sont construites à partir des constantes, des variables ou d'autres expressions de plusieurs manières :

- utilisation de la structure des types
- appel de fonction
- utilisation d'opérateurs (qui peuvent être vus comme des fonctions).

Les expressions utilisant la structure des types sont :

- les ensembles ou listes en extension, comme
  {v,P(a),1} ou (v,P(a),1)
- l'image d'index, comme
  P(e)
- le $n^{ième}$ élément d'un vecteur ou d'une liste (similaire à l'image d'index, cf. note du § 5.4), comme
  v(n)

L'appel de fonction est évident. Les opérateurs envisagés sont

- Opérateurs ensemblistes ∪, ∩, -, +, x, xx, Card, ∈
- Opérateurs sur les listes .[15], |[16], Reverse, Tete, Fin, Pied, Debut, Long, Member
- Opérateur Dom sur les index
- Opérateurs arithmétiques habituels +, -, *, /, **, div, mod, ent, …
- Opérateurs logiques et, ou, non
- Comparateurs =, !=, <, >, <=, >=
- Opérateurs liés aux pointeurs * et &.

Evidemment, il y a des contraintes "sémantiques" sur la formation des expressions : des contraintes statiques (comme les contraintes de types) ou des contraintes

[15] Le "cons" de Lisp.
[16] Append de listes.

dynamiques (il est par exemple interdit de prendre la 5ème composante d'une liste n'en ayant que 3).

5.7 Signification des opérateurs de comparaison

Les comparateurs = et != doivent être correctement interprétés selon les types de leurs opérandes. != est toujours la négation de =.

Sur des types prédéfinis scalaires (entiers, réels), = a le sens habituel. Sur les pointeurs, = signifie l'identité de ses opérandes (et non l'identité de ce qu'elles pointent).

Lorsque E et F sont des ensembles, E=F signifie que les éléments de E sont exactement les éléments de F. Si l'on veut comparer les références de E et F, on doit utiliser &E=&F. Mais attention ! Si les éléments de E et F sont des pointeurs (par exemple des références d'ensembles), ces pointeurs doivent être identiques deux à deux. Par contre, si les éléments de E et F sont des ensembles, alors ces ensembles doivent être égaux du point de vue de leur contenu. L'égalité entre E et F ne fonctionnera donc pas de la même manière selon que E et F sont de types Set(*Set(H)) ou Set(Set(H)).

Les listes sont comparées de la même manière.

Une comparaison C(x)=expr ou expr=C(x), où C est un index, signifie que *x appartient au domaine de C et* que C(x) vaut expr.

Enfin, une comparaison C=D, où C et D sont des index, signifie que les domaines de C et de D sont égaux, et que, pour tout x dans le domaine commun, C(x)=D(x).

∈ et Member sont aussi des comparateurs. Leur sens est lié à celui de =. Ainsi, x∈E signifie "il existe y dans E tel que x=y".

Enfin, rappelons que les expressions *quantifiées* sont *interdites* en ErgoAlg, comme dans

    Tant que ∃ x ∈ E …

C'est à Cogito de faire le travail qui consiste à en fournir un équivalent algorithmique.

5.8 Appartenance à un type

Un type étant un ensemble, on peut tester l'appartenance d'une quantité ErgoAlg à un type. Cela est particulièrement utile pour les réunions de types, comme Doublet = (*Doublet x *Doublet) ∪ Atome. On peut ainsi avoir l'instruction

    Si x ∉ Atome alors …

où x est de type Doublet. Son sens est : x n'est pas un atome.

5.9 Syntaxe

Comme il a été dit en introduction, nous ne définissons pas ici de syntaxe pour ErgoAlg. En effet, un programme ErgoAlg est normalement représenté dans la structure de données de Descartes, que celui-ci soit écrit en Genesia II, Shal, ou Descartes lui-même.

Pourtant, nous avons ici donné des exemples en utilisant une syntaxe "naturelle". Il peut aussi être intéressant de pouvoir sortir les résultats de Cogito (qui est un programme ErgoAlg) pour sa mise au point, voire d'écrire manuellement des programmes ErgoAlg dans certains cas. Il faudra donc définir dans ce cas une syntaxe.

Le problème avec notre syntaxe "naturelle" est qu'elle comporte de nombreuses ambiguïtés pour des algorithmes d'analyse courants, comme ceux fonctionnant avec des grammaires SLR(1) (comme l'algorithme de yacc, par exemple). Ce problème sera résolu dans un premier temps en adoptant une syntaxe moins naturelle, mais acceptable, et dans un second temps en étendant les capacités du générateur d'entrées-sorties [Dormoy, 1993b].

6 Conclusion

Cette note a permis de spécifier les éléments du langage ErgoAlg et leur sémantique, comme base à la construction des composants Cogito et ErgoAlg de Descartes. Des modifications locales seront sans doute nécessaires au cours de ce travail. Cependant, il semble que le fond des choses ne sera pas très différent de ce qui est exposé ici. La première vérification sera la réalisation du traducteur ErgoAlg lui-même (qui traduit un programme ErgoAlg en un programme Sum).

Références

[Dormoy, 1993a] Jean-Luc Dormoy. *Définition du langage Sum.* Note EDF HI21/8342.

[Dormoy, 1993b] Jean-Luc Dormoy. *Définition du Io de spécification des entrées-sorties de Descartes.* Note EDF HI21/8341.

[Laurière, 1978] Jean-Louis Laurière. *A Language and a Program for Stating and Solving Combinatorial Problems.* Artificial Intelligence Vol. 10, pp 29-127, 1978.

[Mohr & Henderson, 1986] Roger Mohr and Thomas C. Henderson. *Arc and Path Consistency Revisited.* Artificial Intelligence Vol. 28, n° 2, pp 225-233, March 1986.