RAIRO. INFORMATIQUE THÉORIQUE

JEAN-LUC RÉMY

Construction, évaluation et amélioration systématiques de structures de données

RAIRO. Informatique théorique, tome 14, n° 1 (1980), p. 83-118

<http://www.numdam.org/item?id=ITA_1980__14_1_83_0>

R.A.I.R.O. Informatique théorique/Theoretical Informatics
(vol. 14, n° 1, 1980, p. 83 à 118)

CONSTRUCTION, ÉVALUATION ET AMÉLIORATION SYSTÉMATIQUES DE STRUCTURES DE DONNÉES (*)

par Jean-Luc RÉMY (1)

Communiqué par M. SINTZOFF

Résumé. — *Un formalisme très simple, les types abstraits, utilisant des équations et des préconditions, est proposé pour décrire les propriétés des objets et des opérations d'une structure de données. Deux structures, les ensembles et les dictionnaires (ou arbres binaires de recherche), sont présentées en exemple. Un petit nombre de règles ou d'heuristiques est alors développé pour construire systématiquement les algorithmes classiques de recherche et d'insertion dans les dictionnaires. La contribution la plus originale consiste à prolonger cette construction par des phases d'évaluation de la complexité et d'amélioration des algorithmes. On est ainsi conduit à transformer l'algorithme simpliste d'insertion de manière à ce que la hauteur des arbres soit maintenue constante aussi souvent que possible. Des transformations permettant de rééquilibrer un arbre (rotation simple et double) et une propriété d'arbres quasi équilibrés (AVL) sont retrouvées de manière presque automatique. Il reste bien entendu qu'un certain nombre d' « eureka » sont nécessaires et chacun d'eux est mis en évidence au cours de la construction.*

L'algorithme obtenu finalement n'est pas nouveau mais est assez compliqué pour justifier l'intérêt de ce type de recherche qui se trouve au carrefour de deux préoccupations : les spécifications abstraites de structures de données et les évaluations d'algorithmes.

Abstract. — *A very simple formalism, viz. abstract data types, using equations and preconditions, is proposed for describing the properties of the objects and the operations of a data structure. Two cases studies are detailed: sets, and dictionaries (viz. binary search trees). A small number of rules and heuristics are then developed in order to systematically construct the classical algorithms of search and insertion in a dictionary. The more original contribution of the paper is to extend this construction by additional phases: the complexity of the algorithms is evalued and then their efficiency is improved. We are thus led to transform the trivial insertion algorithm so that the height of the trees is kept constant as much as possible. Some operations for rebalancing trees (viz. simple and double rotations) and a property of quasi-balanced (AVL) trees are thereby reinvented almost automatically. Of course, a number of « eurekas » remain necessary; each of them is carefully pinpointed during construction.*

The algorithm obtained finally is not new; but it is sufficiently intricated so as to justify the validity of this type of research, which is at the cross-roads of two concerns: the abstract specifications of data structures, and the complexity analysis of algorithms.

(*) Reçu en juillet 1978, révisé en janvier 1979.

(1) Attaché de Recherche au Centre de Recherche en Informatique de Nancy, Nancy.

R.A.I.R.O. Informatique théorique/Theoretical Informatics, 0399-0540/1980/ 83 '$ 5.00

INTRODUCTION

Un problème informatique peut être généralement formulé de la manière suivante :

P : « Étant donné des objets $x_1, \ldots, x_n$ vérifiant une **précondition** notée $\Phi(x)$, trouver des objets $y_1, \ldots, y_m$ tels que $x_1, \ldots, x_n, y_1, \ldots, y_m$ vérifient une **postcondition** que l'on note $\Psi(x, y)$.

Les formules Φ et Ψ, dont nous n'avons pas dit pour l'instant dans quel langage les écrire, représentent les **spécifications** de P. Une seconde formulation peut être :

« *Calculer une fonction f telle que :*

1° *$f(x)$ est définie si (et seulement si) $\Phi(x)$ est vérifiée;*

2° *lorsque $f(x)$ est définie, la formule $\Psi(x, f(x))$ est vérifiée.*»

Dans ce second énoncé, le mot calculer n'a pas été précisé. Si les éléments du domaine d'arrivée de f peuvent être énumérés, il suffit pour calculer $f(x)$ de tester successivement chaque élément y jusqu'à ce que $\Psi(x, y)$ soit vérifié. Plus sérieusement, un problème est considéré comme **résolu** lorsque l'on possède une **définition explicite** de la fonction f, de la forme

$$f(x) = u(x),$$

où $u(x)$ est une certaine expression. Par exemple une solution, de l'équation du second degré $ax^2 + bx + c = 0$ est donné par la formule

$$u(a, b, c) = \frac{-b + \sqrt{b^2 - ac}}{2a},$$

à condition que a soit non nul et $b^2 - 4ac$ positif ou nul.

Dans ces problèmes plus généraux, l'expression u est construite en composant un certain nombre de **fonctions** (ou **opérations**) **de base.**

Ainsi résoudre un problème consiste à passer d'un **énoncé implicite** E (donné par les formules Φ et Ψ) à une définition explicite D.

Dès que le problème est assez important, il n'est plus possible d'exprimer sa **solution** en une simple définition et il est raisonnable de chercher à le décomposer en **sous-problèmes**. Si l'on préfère, on remplace l'énoncé initial E par plusieurs énoncés $E_1, \ldots, E_m$ portant sur des fonctions intermédiaires $f_1, \ldots, f_m$ et par une définition explicite D de f contenant les symboles $f_1, \ldots, f_m$. On obtient ainsi un énoncé partiellement explicite E' formé de $D, E_1, \ldots, E_m$. Bien entendu, il convient de s'assurer que E et E' ont les mêmes solutions ou du moins que E' vérifient les deux conditions :

toute solution de E' est solution de E
si E admet une solution, E' en admet également une.

Donnons un exemple. Soit à évaluer une expression arithmétique bien parenthésée. Il est possible de décomposer le problème en deux sous-problèmes indépendants :

(1) mettre l'expression e sous forme postfixée **POST** (e);

(2) calculer la valeur **VAL**(e') d'une expression e' en forme postfixée.

Si l'on note **EVAL** (e) la valeur d'une expression e bien parenthésée, on obtient une première définition

$$\textbf{EVAL}\ (e) = \textbf{VAL}\ (\textbf{POST}\ (e)).$$

Ce procédé de construction d'une solution par **raffinements successifs** est bien connu et nous n'en reparlons ici que pour montrer qu'il est possible d'adopter une démarche similaire sur les données.

Il est en effet essentiel pour résoudre un problème de rester le plus longtemps possible éloigné des préoccupations d'implémentation et de raisonner pour cela sur des données et des opérations abstraites.

Un deuxième problème consiste alors à **représenter** ces opérations abstraites par des opérations concrètes telles que l'échange de deux valeurs dans un tableau ou l'adjonction d'un élément en tête d'une liste chaînée. Là aussi, il est nécessaire de procéder progressivement et d'imaginer des **opérations intermédiaires** et des définitions explicites des opérations abstraites en fonction de celles-ci. Pour que le parallèle soit complet, il faut disposer de spécifications à la fois pour les opérations abstraites et pour les intermédiaires. Un **type abstrait** (ou **structure**) **de données** est caractérisé par un ensemble d'opérations et par un ensemble de spécifications de ces opérations, le problème posé est alors de représenter une structure de donnée **source** par une structure **cible** en utilisant généralement des structures intermédiaires et de telle manière que les spécifications de la source soient respectées par la structure cible.

Nous nous proposons ici de montrer, essentiellement sur un exemple que la construction de représentations d'une structure par une autre peut être dans une certaine mesure systématisée. Le formalisme utilisé ici est, à peu de chose près, le cadre algébrique (ou équationnel) développé par Guttag [17] et dans une certaine mesure par l'équipe ADJ [16]. Outre les définitions simples décrites ci-dessus, on peut écrire des définitions conditionnelles et surtout des définitions récursives. Il s'avère que ces dernières sont un moyen commode à la fois pour spécifier les opérations d'une structure et pour écrire des définitions explicites. En effet les objets d'un type abstrait sont généralement structurés, ce qui permet des définitions « par récurrence ».

Les définitions explicites construites dans ce contexte peuvent être considérées comme des programmes « sans affectation ». Au niveau de la résolution d'un

problème, il devient clair en effet qu'il faut toujours s'exprimer sur un plan fonctionnel. Dans les problèmes de tri, par exemple, il est plus clair d'écrire :

« *Trouver un tableau t' trié permutation d'un tableau t donné* »

que

« *Permuter les éléments de t de manière à obtenir un tableau trié.*»

Par contre, dans l'état actuel de l'art, ce point de vue fonctionnel a pour conséquence de nombreuses recopies au moment de l'exécution et un certain nombre de transformations sont nécessaires pour passer d'une définition explicite fonctionnelle à un programme efficace. Avec Burstall et Darlington [7] nous pensons que ces transformations peuvent être menées très systématiquement et porter le plus longtemps possible sur des programmes récursifs.

Le travail que nous proposons ici se situe actuellement en amont de celui de Burstall et Darlington puisque ceux-ci proposent des règles pour transformer des définitions explicites déjà construites. Il est par contre assez proche des travaux de Gaudel et Terrine [15], de Guttag [17] et de Darlington [10]. Dans l'exemple que nous traitons, nous faisons assez souvent appel à l'intuition, en le signalant aussi explicitement que possible. Nous voulons aussi citer le travail de Sintzoff [29] qui nous a inspirés bien qu'il soit très différent.

On peut enfin citer le travail de Manna et Waldinger [23] qui s'intéresse à la synthèse de définitions récursives et traite particulièrement la synthèse de cas.

1. GÉNÉRALITÉS SUR LES TYPES ABSTRAITS. LE TYPE ENSEMBLE

1.1. Esquisse du formalisme utilisé

Un objectif qu'il faut garder présent à l'esprit est d'obtenir un système de démonstration assez simple pour être facilement utilisable et assez puissant pour permettre toute spécification désirable. A ce titre les systèmes d'équations proposés par ADJ [16] et Guttag [17] sont extrêmement simples et permettent des démonstrations automatiques assez efficaces. Ils ne rendent cependant que partiellement compte des problèmes de préconditions.

A l'inverse, le calcul des prédicats avec quantificateurs proposés entre autre par Finance [12] et Rémy [27] est très puissant mais trop riche pour être utilisé simplement. Nous désirons éviter l'usage des quantificateurs existentiels et de la négation, étant entendu qu'une formule est toujours universellement quantifiée sur les variables qui apparaissent en son sein.

On peut encore citer les formalismes développés par de Roever [28] et Liskov et Zilles [22].

1.1.1. *Syntaxe et axiomes*

Les concepts développés ici sont classiques (*voir* par exemple [17, 21]).

Une **signature** est un ensemble de **noms de sorte** ou de **types** $\mathscr{T}_0, \mathscr{T}_1, \ldots, \mathscr{T}_n$ et un ensemble de **symboles d'opération** à chacun desquels est attaché un **profil** $i_1, \ldots, i_k \to j$ désignant les sortes des arguments et du résultat. Un symbole sans argument (donc tel que $k=0$) est encore appelé **constante**. L'une des sortes, notée **Bool**, comporte deux constantes *vrai*, *faux* et les opérations logiques *non*, *ou*, *et*. Les opérations à résultat dans **Bool** sont appelées souvent **prédicats** et l'on adoptera fréquemment des notations infixées (telles que $x < y$) ou postfixées (telles que x est défini). De plus on abrège l'équation $p(x)=vrai$ en $p(x)$.

Pour chaque type $\mathscr{T}$ et chaque entier n on introduit une opération $cas_{\mathscr{T},n}$ de profil (**Bool**, $\mathscr{T}$)$^n \to \mathscr{T}$ telle que $cas_{\mathscr{T},n}(p_1, u_1, \ldots, p_n, u_n)=u_i$ si $p_i=vrai$ (cette expression n'étant définie que si p_1 *ou* ... $p_n=vrai$ et si p_i *et* p_j est égal à *faux* dès que $i \neq j$). On écrit l'expression ci-dessus sous la forme plus classique

cas p_1 *alors* u_1,
p_2 *alors* u_2,
.....
p_n *alors* u_n *fcas*.

On écrit plus classiquement l'expression

cas p_1 *alors* u_1, *non* p_1 *alors* u_2 *fcas*

sous la forme

si p_1 *alors* u_1 *sinon* u_2 *fsi*.

On distingue, parmi les types, le **type d'intérêt** encore noté $\mathscr{T}_0$. Les autres sont encore appelées **paramètres**. Les opérations à résultat dans $\mathscr{T}_0$ sont appelées **générateurs** du type. On désire en effet représenter chaque objet par une expression obtenue en utilisant ceux-ci. Dans les cas plus simples comme les entiers, chaque expression représente un objet distinct. En général, cependant, les différents générateurs ne sont pas indépendants. Dans le type **ENSEMBLE**, par exemple, les opérations **AJOUTE** et **RETIRE** sont reliées par l'axiome

RETIRE (x, **AJOUTE** (y, E))$=$*si* $x=y$
alors E *sinon* **AJOUTE** (y, **RETIRE** (x, E)) *fsi*

dans lequel E est une variable de type **ENSEMBLE** et x, y des variables de type **ID**.

En utilisant cet axiome et l'axiome **RETIRE** (x, **VIDE**) = **VIDE**, on peut réécrire toute expression construite avec **VIDE**, **AJOUTE** et **RETIRE** en une expression équivalente, formée seulement de **VIDE** et **AJOUTE**, encore appelée **forme normale**. On peut dire que **VIDE** et **AJOUTE** forment un **système complet de générateurs** (ou encore que **VIDE** et **AJOUTE** sont les **générateurs principaux** du type).

Par ailleurs, les opérations du type abstrait dont le résultat est un type paramètre doivent également être définies par un système d'équations. Toujours dans le type **ENSEMBLE**, **EXISTE** peut être défini par les axiomes

EXISTE (x, **VIDE**) = *faux*,
EXISTE (x, **AJOUTE** (y, E)) = ($x = y$) *ou* **EXISTE** (x, E).

1.1.2. *Préconditions*

Pour compléter la spécification d'un type abstrait, il reste à formaliser le fait qu'une opération peut ne pas être partout définie ou plus exactement qu'elle transforme un objet intéressant en un autre qui l'est moins. Ainsi on peut décider de ne pas s'intéresser à **RETIRE** (x, **VIDE**) ni à **AJOUTE** (y, **AJOUTE** (y, E)). On associe pour cela à chaque opération f un prédicat ou **précondition** notée pre($f(x)$). Ainsi on peut convenir que

$$\text{pre } (\textbf{RETIRE } (x, E)) = \textbf{EXISTE } (x, E)$$

tandis que

$$\text{pre } (\textbf{AJOUTE } (x, E)) = \textit{non } \textbf{EXISTE } (x, E).$$

Il est possible d'étendre cette définition à toute expression en posant

$$\text{pre } (x) = \textit{vrai};\ \text{pre } (g(u)) = \text{pre } (g(x))\ [u/x]\ \textit{et}\ \text{pre } (u).$$

On ne s'intéresse alors qu'aux formules $u = v$ telles que pre $(u) \Rightarrow$ pre (v) soit égal à *vrai*. La signification d'une telle formule est

« *Si pre* (u) *est vérifiée, alors pre* (v) *est également vérifiée et u égale v.*»

Ainsi l'axiome

RETIRE (x, **AJOUTE** (y, E)) = *si* $x = y$
alors E *sinon* **AJOUTE** (y, **RETIRE** (x, E)) *fsi*

n'est vérifié que si

non **EXISTE** (y, E) *et* **EXISTE** (x, **AJOUTE** (y, E))

est égal à *vrai*. De plus on peut vérifier que

pre (**RETIRE** (x, **AJOUTE** (y, E))

$\Rightarrow$ pre (*si* $x = y$ *alors* E *sinon* **AJOUTE** (y, **RETIRE** (x, E)) *fsi*).

Les préconditions on été introduites également très récemment par Guttag.

1.2. Démonstrations et théorèmes

Disposant d'un ensemble d'axiomes ou propriétés et de préconditions pour que ces axiomes soient vérifiés, on peut maintenant se demander quelles autres propriétés sont vérifiées par les objets et opérations d'un type. Ces propriétés sont encore appelées **théorèmes** et peuvent être déduites par une **démonstration** dont les **règles** sont extrêmement simples.

1° **Règle d'instantiation : Si** $u(x) = v(x)$ **est un théorème contenant une variable** x **et si** a **est une expression de même type que** x, $u(a) = v(a)$ **est encore un théorème.**

Exemple : La formule suivante est déduite de l'axiome de **RETIRE** en remplaçant E par **VIDE** et y par x :

RETIRE (x, **AJOUTE** (x, **VIDE**)) = *si* $x = x$ *alors* **VIDE**

sinon **AJOUTE** (x, **RETIRE** (x, **VIDE**) *fsi*.

2° **Règle de réécriture : Si** $u = v$ **est un théorème, si** u **contient une sous-expression** u_1 **telle qu'il existe un théorème** $u_1 = u_2$, **si** v **contient une sous-expression** v_1 **telle qu'il existe un théorème** $v_1 = v_2$ **alors** $u' = v'$ **est un théorème où** u' **et** v' **sont déduites de** u **et** v **en remplaçant respectivement** u_2 **par** u_1 **et** v_2 **par** v_1.

REMARQUE : Il est facile de montrer que, si u_1 est une sous-expression de u, pre (u) implique pre (u_1) (en réalité, si u_1 est à l'intérieur d'une conditionnelle, il faut tenir compte des hypothèses introduites par celle-ci). De plus, puisque pre (u_1) implique pre (u_2), il s'ensuit que pre (u) implique pre (u'). De ce fait, il n'y a aucune vérification à effectuer sur les préconditions.

Une formule $u = v$ est un **théorème** si l'on peut en construire une **démonstration** c'est-à-dire une suite $u_0 = v_0, u_1 = v_1, \ldots, u_n = v_n$ telle que $u_0 = u, v_0 = v, u_n$ et v_n coïncident et que, pour tout i, $u_{i+1} = v_{i+1}$ se déduise de $u_i = v_i$ par réécriture.

Dans les cas les plus simples, il suffit d'effectuer des réécritures sur le terme de gauche. Il en va ainsi lorsque l'on veut simplifier au maximum une expression. Ainsi

u_0 = **EXISTE** (3, **RETIRE** (2, **AJOUTE** (2, **VIDE**))

se réécrit successivement en

$u_1 =$ **EXISTE** (3, *si* $2=2$ *alors* **VIDE** *sinon*

AJOUTE (2, **RETIRE** (2, **VIDE**)) *fsi*)

par l'axiome de **RETIRE**, puis en

$$u_2 = \textbf{EXISTE}\ (3,\ \textbf{VIDE}),$$

et enfin en

$$u_3 = \textit{faux}.$$

De la même manière

$$v_0 = \textit{non}\ (3=2)\ \textit{et}\ \textbf{EXISTE}\ (3,\ \textbf{AJOUTE}\ (2,\ \textbf{VIDE})),$$

se réécrit en

$$v_1 = \textit{non faux et}\ ((3=2)\ \textit{ou}\ \textbf{EXISTE}\ (3,\ \textbf{VIDE})),$$

par l'axiome de **AJOUTE**, puis en

$$v_2 = \textit{vrai et}\ (\textit{faux ou faux}),$$

soit encore en

$$v_3 = \textit{faux}.$$

(On a abrégé quelque peu certaines étapes triviales.)

Finalement la séquence $u_0 = v_0,\ u_1 = v_1,\ u_2,\ v_2,\ \textit{faux} = \textit{faux}$ représente une démonstration du théorème

$$\left.\begin{array}{l}\textbf{EXISTE}\ (3,\ \textbf{RETIRE}\ (2,\ \textbf{AJOUTE}\ (2,\ \textbf{VIDE})))\\ \qquad = \textit{non}\ (3=2)\ \textit{et}\ \textbf{EXISTE}\ (3,\ \textbf{AJOUTE}\ (2,\ \textbf{VIDE})).\end{array}\right\} \quad (1)$$

3° **Règle de récurrence :** La formule (1) est une instantiation de la formule plus générale

$$\textbf{EXISTE}\ (z,\ \textbf{RETIRE}\ (x,\ E)) = \textit{non}\ (z=x)\ \textit{et}\ \textbf{EXISTE}\ (z,\ E), \qquad (2)$$

qui n'est pas démontrable en utilisant uniquement les règles précédentes. On introduit donc une **règle de récurrence sur les générateurs principaux** (ici **VIDE** et **AJOUTE**) qui peut être énoncée de la manière suivante :

Règle de récurrence : Pour toute formule Φ, si Φ (VIDE) est un théorème et si Φ (AJOUTE (x, E)**) peut être démontrée à partir de Φ** (E) **alors Φ est un théorème.**

Utilisons cette règle pour démontrer le théorème (2) que nous baptisons Φ (*E*). Il n'y a rien à démontrer pour $E =$ **VIDE** puisque **RETIRE** (x, **VIDE**) n'est pas défini. Supposons maintenant Φ (*E*) vérifiée et soit y un identificateur tel que *non* **EXISTE** (y, *E*) soit vrai. Alors, le premier membre de Φ (**AJOUTE** (y, *E*)) se réécrit en

EXISTE (z, *si* $x = y$ *alors E sinon* **AJOUTE** (y, **RETIRE** (x, *E*))) *fsi*), (3)

par l'axiome de **RETIRE** puis en

si $x = y$ *alors* **EXISTE** (z, *E*)

sinon $z = y$ *ou* **EXISTE** (z, **RETIRE** (x, *E*)) *fsi*, (4)

par l'axiome de **EXISTE** puis en

si $x = y$ *alors* **EXISTE** (z, *E*)

sinon $z = y$ *ou* (*non* ($z = x$) *et* **EXISTE** (z, *E*)) *fsi*, (5)

par hypothèse de récurrence.

Par les préconditions, *non* **EXISTE** (y, *E*) est égal à *vrai*. Dans le cas $x = y$ on a donc *non* ($z = x$). De plus les conditions $x = y$, *non* ($z = x$) et $z = y$ sont incompatibles. **EXISTE** (z, *E*) est donc équivalent dans ce cas à *non* ($z = x$) et ($z = y$ ou **EXISTE** (z, *E*)).

De la même façon, lorsque *non* ($x = y$), la condition $z = y$ implique *non* ($z = x$) et l'expression suivant *sinon* dans (5) se réécrit en *non* ($z = x$) *et* ($z = y$ *ou* **EXISTE** (z, *E*)). Finalement (5) se réécrit dans cette dernière expression qui est équivalente au second membre de Φ (**AJOUTE** (y, *E*)).

Ainsi l'on a su prouver Φ (**AJOUTE** (y, *E*)) en supposant Φ (*E*) ce qui achève la démonstration.

1.3. Le type ENSEMBLE

Il n'est pas nécessaire de donner ici une description complète de ce type, puisqu'elle a été développée dans les paragraphes précédents. Notons simplement que ce type dépend des types **ID** et **BOOL**. Les opérations de **ID** sont des constantes représentant chaque élément et un prédicat = tel que $x = y$ soit vrai si et seulement si x et y sont le même élément. Les opérations du type **ENSEMBLE** (encore noté **ENS**) sont **EXISTE**, **AJOUTE** et **RETIRE** ainsi que la constante **VIDE**. Trois axiomes suffisent puisqu'il n'est plus nécessaire de définir **RETIRE** (x, **VIDE**).

Nous serons amenés dans la suite à supposer que **ID** est un ensemble totalement ordonné donc muni d'un prédicat $<$ vérifiant les propriétés habituelles. Il est alors utile d'étendre $<$ au type **ENS** et nous introduirons pour cela un prédicat **PPQ** sur **ID** $\times$ **ENS**.

Pour définir **PPQ**, il est intéressant d'introduire les quantificateurs, à condition d'en limiter très strictement l'usage. Posons donc

$$\mathbf{PPQ}(x, E) = \forall z\,(\mathbf{EXISTE}(z, E) \Rightarrow x < z)$$

(où $\Rightarrow$ est l'opération booléenne d'implication). On écrira encore $x < E$ au lieu de $\mathbf{PPQ}(x, E)$.

Nous utiliserons cette définition uniquement pour donner une définition récursive de **PPQ**. Ainsi on peut écrire :

- $x < \mathbf{VIDE} = \forall z\,(\mathbf{EXISTE}(z, \mathbf{VIDE}) \Rightarrow x < z)$

$$= \forall z\,(\mathit{faux} \Rightarrow x < z) = \forall z\ \mathit{vrai} = \mathit{vrai}$$

et

- $x < \mathbf{AJOUTE}(y, E) = \forall z\,(\mathbf{EXISTE}(z, \mathbf{AJOUTE}(y, E)) \Rightarrow x < z)$

$$= \forall z\,(((z = y)\ \mathit{ou}\ \mathbf{EXISTE}(z, E)) \Rightarrow x < z)$$

$$= \forall z\,(x < y\ \mathit{et}\ \mathbf{EXISTE}(z, E) \Rightarrow x < z)$$

$$= \forall z\,(x < y)\ \mathit{et}\ \forall z\,(\mathbf{EXISTE}(z, E) \Rightarrow x < z) = x < y\ \mathit{et}\ x < E.$$

Nous travaillons actuellement sur le type de calculs où l'on peut admettre des quantificateurs. Il semble, après quelques expériences que les deux règles suivantes soient très intéressantes

$$\forall z\,(p\ \mathit{et}\ q) = \forall z\, p\ \mathit{et}\ \forall z\, q,$$
$$\forall z\, p \qquad = p \quad \text{si } z \text{ n'apparaît pas dans } p.$$

De toute manière il n'est pas question d'admettre le calcul des quantificateurs au niveau des preuves principales. Par contre on utilisera celui-ci chaque fois qu'une propriété s'avère nécessaire dans une démonstration.

1.4. Enrichissement d'un type abstrait par introduction d'opérations nouvelles

Pour spécifier un type abstrait, il est utile (et vite nécessaire) d'adopter une démarche progressive (ou structurée) (*voir* par exemple Burstall et Goguen [8]). En particulier on peut commencer par introduire un nombre minimal

d'opérations (dont les générateurs principaux) puis enrichir le type en spécifiant de nouvelles opérations en fonction des premières. Un problème informatique consiste précisément en l'introduction de nouvelles opérations (ses inconnues) spécifiées par un énoncé en général implicite, c'est-à-dire par un ensemble d'équations; on précise également le domaine sur lequel l'inconnue doit être définie, c'est-à-dire sa précondition.

Pour simplifier, donnons les définitions uniquement dans le cas d'une inconnue unique f; on note dans ce cas $E(f)$ l'ensemble des équations formant l'énoncé et $\mathrm{Pre}(f(x))$ la précondition de f.

Dans les définitions suivantes, on suppose également que f admet un seul argument qui est du type d'intérêt.

Si $\{g_1, \ldots, g_m\}$ est un ensemble de générateurs principaux pour le type abstrait, on appelle **définition explicite** de f un système d'équations $S(f)=\{f(g_i(x))=u_i(x)\}$ dans lequel les expressions u_i contiennent éventuellement des occurrences de f et où x représente l'ensemble des variables pouvant être arguments de g_i.

Adoptons une abréviation. Si A et B sont deux ensembles d'équations on écrira $A \models B$ si chaque équation de B peut être démontrée en utilisant les hypothèses auxiliaires de A.

On dira que $S(f)$ est **à terminaison bornée** si elle permet de calculer $f(u)$ pour toute expression u sans variable vérifiant $\mathrm{Pre}(u)$ autrement dit si, pour toute expression u, on peut trouver une expression v telle que $S(f) \models f(u)=v$.

On dira que $S(f)$ **satisfait l'énoncé** $E(f)$ si et seulement si $S(f) \models E(f)$.

Une condition suffisante (mais non nécessaire) pour qu'une définition soit à terminaison bornée est que pour chaque i, l'argument de f à l'intérieur de $u_i(x)$ soit l'un des arguments de g_i.

On dira que $S(f)$ est une **solution** du problème spécifié par l'énoncé $E(f)$ si $S(f)$ est à terminaison bornée et satisfait l'énoncé $E(f)$.

Le type d'équations introduit pour définir de nouvelles opérations vérifie les critères proposés par Musser. C'est pourquoi nous nous contenterons d'effectuer des preuves de réécriture.

2. REPRÉSENTATION D'UN ENSEMBLE PAR UN DICTIONNAIRE

2.1. Généralités sur les représentations

On appelle **opération généralisée** (sur un type abstrait $\mathscr{T}$) toute expression $u(x_1, \ldots, x_n)$ construite en composant les opérations de base de $\mathscr{T}$.

Dans un processus de représentation, il est courant que plusieurs objets distincts du type cible représentent le même objet du type source. Cela conduit à munir le type cible $\mathscr{C}$ d'un symbole d'équivalence $\sim$, de profil $\mathscr{C} \times \mathscr{C} \to \{vrai, faux\}$.

Étant donné deux types $\mathscr{S}$ (**type source**) et $\mathscr{C}$ (**type cible**), dépendant de mêmes types $\mathscr{T}_1, \ldots, \mathscr{T}_n$, une **prérepresentation** ρ de $\mathscr{S}$ par $\mathscr{C}$ est une application de l'ensemble des opérations de $\mathscr{S}$ dans celui des opérations généralisées de $\mathscr{C}$ associant à chaque symbole une opération de profil similaire.

Si $\mathscr{C}$ est muni d'un prédicat d'équivalence $\sim$, on impose de plus que le symbole d'égalité entre objets de type $\mathscr{S}$ soit représenté par $\sim$. Une prérepresentation ρ se prolonge à l'ensemble des expressions de manière unique en respectant les règles de composition, c'est-à-dire comme un homomorphisme. De plus, si u et v sont des expressions, on pose

$$\rho(u=v)=(\rho(u)\sim\rho(v)) \text{ si } u \text{ et } v \text{ sont de type } \mathscr{S},$$

$$\rho(u=v)=(\rho(u)=\rho(v)) \text{ sinon.}$$

On dit que ρ est une représentation de $\mathscr{S}$ par $\mathscr{C}$ si :

(1) pour tout axiome ρ de $\mathscr{S}$, $\rho(\varphi)$ est un théorème de $\mathscr{C}$;

(2) pour toute opération f de $\mathscr{S}$, $\rho(\text{pre}(f(x)))=\text{pre}(\rho(f)(x))$.

Il résulte de ces conditions que l'équivalence ne peut être choisie de manière quelconque. Si u et v sont des expressions et f une opération dont le type du résultat est un type paramètre, si $u=v$ est un théorème, on a

$$\rho(u)\sim\rho(v) \text{ et } \rho(f)(\rho(u))=\rho(f)(\rho(v)).$$

On peut dire que $\sim$ doit vérifier la condition

« *Pour tout x, y du type d'intérêt,* $x\sim y \Rightarrow \rho(f)(x)=\rho(f)(y)$ ».

Ceci doit être encore vrai pour toute représentation d'une opération généralisée dont le résultat est d'un type paramètre.

Le plus faible prédicat d'équivalence compatible avec la représentation est donc

« $x\sim y \Leftrightarrow$ *pour tout accès généralisé f de* $\mathscr{S}$, $\rho(f)(x)=\rho(f)(y)$ ».

Dans le cas du type **ENSEMBLE**, on peut vérifier que tout accès généralisé peut se simplifier en une expression contenant exclusivement **EXISTE**. Si l'on pose $\rho(\textbf{EXISTE})=\textbf{PRESENT}$ on peut définir l'équivalence de deux dictionnaires E et E' par

$$E\sim E'=\forall z\,(\textbf{PRESENT}(z, E)=\textbf{PRESENT}(z, E').)$$

Cette définition utilise une fois de plus un quantificateur et nous verrons dans la suite comment déduire de celle-ci plusieurs propriétés de l'équivalence.

Construire une représentation d'un type $\mathscr{S}$, c'est :

(*a*) définir un type $\mathscr{C}$, c'est-à-dire des symboles et des axiomes;

(*b*) associer à chaque opération de $\mathscr{S}$ une opération généralisée de $\mathscr{C}$ et en déduire un prédicat d'équivalence.

Généralement cette démarche s'avère trop simpliste. C'est le cas si l'on veut représenter une opération f du type source par une opération f' définie récursivement. Dans ce cas, on introduit dans un premier temps le symbole f' et les traductions des spécifications de f dans le type $\mathscr{C}$. Puis l'on recherche dans $\mathscr{C}$ des définitions récursives de f' satisfaisant les spécifications introduites. On trouvera deux exemples de cette démarche au paragraphe 2.5 et 2.6.

Ces généralités sur les représentations diffèrent quelque peu des concepts proposés par Gaudel et Terrine [15] et par Guttag [17].

D'une part ces auteurs définissent la fonction de représentation du type cible vers le type source.

D'autre part Gaudel et Terrine introduisent effectivement la notion d'équivalence mais sans donner de règle systématique de définition.

2.2. Spécifications du type dictionnaire

Il est courant de ramener la résolution d'un problème sur un ensemble E à la résolution du même problème sur des sous-ensembles (disons E_1, E_2) de E. On insiste habituellement sur le fait que E_1, E_2 doivent être de même taille mais l'idée reste intéressante si l'on partage E de manière aléatoire. Ayant en tête la recherche dichotomique classique nous considérons un élément médian noté **MEDIAN**(E) partageant E en deux sous-ensembles **GAUCHE**(E) et **DROITE**(E) selon un critère à préciser. Réciproquement, à partir d'un élément y et deux ensembles E_1 et E_2 vérifiant un critère convenable, il est possible de reconstituer un ensemble noté **CONCAT** (E_1, y, E_2) ou encore $\langle E_1, y, E_2 \rangle$; on peut bien entendu construire n'importe quel ensemble à partir de l'ensemble **VIDE** en utilisant répétitivement l'opération **CONCAT** et les éléments de **ID**.

On peut alors définir très simplement une opération **PRESENT**. Pour calculer **PRESENT** $(x, \langle E_1, y, E_2 \rangle)$, on compare x et y suivant le critère retenu. Selon le résultat, ou bien $x = y$ (auquel cas x est présent) ou bien l'on oriente les recherches vers E_1 ou E_2. On introduit enfin deux opérations **INSERE** et **EXTRAIT** pour représenter **AJOUTE** et **RETIRE**.

On désigne par **DICT** le type abstrait dont les opérations de base sont **VIDE**, **MEDIAN**, **GAUCHE**, **DROITE**, **CONCAT**, **PRESENT**, **INSERE**, **EXTRAIT**

et un prédicat **ESTVIDE** et dont les axiomes présentés informellement ci-dessus sont explicités dans la figure 1.

(1) **Règle de récurrence** : Pour toute assertion Φ sur DICT :

Si $\vdash \Phi$ (**VIDE**) *et si, pour tout* $E_1, E_2 \in$ DICT, *tout* $x \in$ ID

$\Phi(E_1)$ *et* $\Phi(E_2) \models \Phi(\langle E_1, x, E_2 \rangle)$

alors $\models \Phi$

(2) **Spécifications de GAUCHE, MEDIAN, DROITE**

pre(**GAUCHE**(E)) = pre(**MEDIAN**(E)) = pre(**DROITE**(E)) = *non* **ESTVIDE**(E)

GAUCHE $(\langle E_1, y, E_2 \rangle) = E_1$

MEDIAN $(\langle E_1, y, E_2 \rangle) = y$

DROITE $(\langle E_1, y, E_2 \rangle) = E_2$

ESTVIDE (VIDE) = *vrai*, **ESTVIDE** $(\langle E_1, y, E_2 \rangle)$ = *faux*

(3) **Spécifications de PRESENT**

PRESENT $(x,$ VIDE$) =$ *faux*

PRESENT $(x, \langle E_1, y, E_2 \rangle)$ = **PRESENT** (x, E_1) *ou* $x = y$ *ou* **PRESENT** (x, E_2)

(4) **Spécifications de l'équivalence de deux dictionnaires**

$E \sim E' = \forall z$ (**PRESENT** (z, E) = **PRESENT** (z, E'))

(5) **Spécifications de INSERE et EXTRAIT**

pre(**INSERE**(x, E)) = *non* **PRESENT**(x, E)

pre(**EXTRAIT**(x, E)) = **PRESENT**(x, E)

PRESENT$(z,$ **INSERE**$(x, E)) = z = x$ *ou* **PRESENT**(z, E)

EXTRAIT$(x,$ **INSERE**$(z, E)) \sim$ *si* $x = z$ *alors* E *sinon* **INSERE**$(z,$ **EXTRAIT**$(x, E))$ *fsi*

Figure 1. – Spécifications du type abstrait DICT.

Compte tenu des définitions du paragraphe précédent, l'application définie par ρ(**VIDE**) = **VIDE**, ρ(**EXISTE**) = **PRESENT**, ρ(**AJOUTE**) = **INSERE** et ρ(**RETIRE**) = **EXTRAIT** est évidemment une représentation du type **ENS** par le type **DICT**.

Dans les paragraphes suivants, nous précisons la définition de **PRESENT** et proposons des définitions explicites (et récursives) pour **INSERE** et **EXTRAIT**. Auparavant, précisons en quelques mots les transformations que nous allons utiliser pour passer d'une spécification implicite à une définition explicite.

2.3. Stratégies de construction de définitions explicites

Rappelons tout d'abord l'objectif. Il s'agit de construire un système d'équations de la forme

$$S(f) = \{ f(g_i(x)) = u_i(x) \},$$

pour lequel on puisse prouver :

(*a*) que les assertions de l'énoncé $E(f)$ sont vérifiées;

(*b*) que la précondition Pre (f) est telle que Pre $(f(g_i(x))) \Rightarrow$ Pre $(u_i(x))$ pour tout i;

(*c*) et que le système est à terminaison bornée.

Nous proposons de distinguer deux phases qui dans la réalité seront intimement mêlées.

1re phase : construction d'un système de définitions explicites satisfaisant (*a*).

2e phase : vérifications des conditions (*b*) et (*c*).

Généralement le système trouvé au cours de la première phase ne vérifie les conditions (*b*) et (*c*) que sous certaines hypothèses supplémentaires qui sont déterminées au cours de la seconde phase. On se trouve en présence d'un processus itératif consistant à construire en réalité plusieurs systèmes d'équations $S_1, \ldots, S_k$ cohérents modulo des hypothèses $H_1, \ldots, H_k$. Le processus s'achève lorsque tous les cas sont envisagés.

Le second point important est que même si le problème initial comporte une seule inconnue, il est utile, et parfois indispensable, d'introduire des **inconnues auxiliaires**. Le cas le plus fréquent est le suivant : Soit f une inconnue dont le résultat est du type d'intérêt et g un générateur, on fait l'hypothèse que $f(g(x))$ est de la forme $g'(f_1(x), \ldots, f_m(x))$ où g' est un générateur du type. On transforme ainsi l'énoncé $E(f)$ en un énoncé $E'(f_1, \ldots, f_m)$ $(= E(g'(f_1, \ldots, f_m)))$ qui est tel que $E(f)$ se déduit de $E'(f_1, \ldots, f_m)$ sous l'hypothèse supplémentaire $f(g(x)) = g'(f_1(x), \ldots, f_m(x))$. On dira encore que l'**on a partiellement explicité l'énoncé**.

Généralement on s'efforce de scinder l'énoncé $E'(f_1, \ldots, f_m)$ en plusieurs énoncés indépendants $E_i'(f_i)$. On est ainsi placé en face de plusieurs sous-problèmes que l'on résoud indépendamment.

On peut aussi expliciter l'énoncé sans introduire de nouvelles inconnues, par exemple en posant $f(g(x_1, x_2, \ldots, x_n)) = x_i$ ou bien $f(g(x_1, \ldots, x_n)) = f'(x_i)$. On effectue de telles explicitations chaque fois qu'elles permettent de prouver l'énoncé $E(f)$ en tenant compte des axiomes du type abstrait. Par exemple on peut poser $f(x, E) =$ **INSERE** (x, E) si l'énoncé $E(f)$ est de la forme

$$\textbf{PRESENT}(z, f(x, E)) = (z = x) \text{ ou } \textbf{PRESENT}(z, E).$$

En dehors de cette technique de transformation d'énoncé par explicitation partielle la seule règle utilisée est la règle de réécriture préalablement introduite.

Pour faciliter la lisibilité de ces constructions, on introduit un certain nombre de constantes pour abréger des expressions telles que $fu_1 \ldots u_n$.

2.4. Définition du PRESENT. Complément sur la structure DICT

La spécification proposée pour **PRESENT** dans les axiomes du type **DICT** est déjà une définition explicite.

$$\mathbf{PRESENT}(x, \mathbf{VIDE}) = \textit{faux}, \tag{1}$$

$$\mathbf{PRESENT}(x, \langle E_1, y, E_2 \rangle) = \mathbf{PRESENT}(x, E_1) \textit{ ou } x = y \textit{ ou } \mathbf{PRESENT}(x, E_2). \tag{2}$$

Il est possible, en supposant l'ensemble **ID** muni d'une relation d'ordre totale notée $\leqq$ de transformer cette définition en une définition moins indéterministe, telle que

$$\mathbf{PRESENT}(x, \langle E_1, y, E_2 \rangle) = \mathbf{PRESENT}(x, E_1) \textit{ et } x < y \textit{ ou vrai et } x = y \textit{ ou } \mathbf{PRESENT}(x, E_2) \textit{ et } x > y. \tag{3}$$

Il n'y a actuellement pas d'axiomes permettant de déduire (2) de (3) et il faut donc compléter les spécifications de **DICT**, par exemple par les axiomes

$$\mathbf{PRESENT}(x, E_1) = \mathbf{PRESENT}(x, E_1) \textit{ et } x < y, \tag{4}$$

$$\mathbf{PRESENT}(x, E_2) = \mathbf{PRESENT}(x, E_2) \textit{ et } x > y, \tag{4'}$$

ou si l'on préfère

$$\mathbf{PRESENT}(x, E_1) \Rightarrow (x < y) = \textit{vrai}, \tag{5}$$

$$\mathbf{PRESENT}(x, E_2) \Rightarrow (x > y) = \textit{vrai}. \tag{5'}$$

C'est-à-dire encore, compte tenu de la définition proposée au paragraphe 1.3,

$$E_1 < y \textit{ et } y < E_2. \tag{6}$$

Cette assertion est la plus faible précondition à imposer aux arguments de **CONCAT** et nous la supposerons vérifiée par la suite.

2.5. Définition explicite de INSERE

Abrégeons à partir d'ici **PRESENT** (z, E) en **PR** (z, E) et **INSERE** (x, E) en $E + x$.

Première partie : **construction d'une définition explicite.**

La spécification à satisfaire est

$$\mathbf{PR}(z, E + x) = (z = x) \textit{ ou } \mathbf{PR}(z, E). \tag{1}$$

Introduisons trois inconnues auxiliaires E'_1, y', E'_2 et explicitons partiellement $E+x$ en posant $E+x=\langle E'_1, y', E'_2 \rangle$. (1) se réécrit en

$$\mathbf{PR}(z, \langle E'_1, y', E'_2 \rangle)=(z=x) \text{ ou } \mathbf{PR}(z, E) \tag{2}$$

ou encore, par l'axiome de **PR**, en

$$\mathbf{PR}(z, E'_1) \text{ ou } (z=z') \text{ ou } \mathbf{PR}(z, E'_2)=(z=x) \text{ ou } \mathbf{PR}(z, E). \tag{2'}$$

Premier cas : $E=$**VIDE** :

$$\mathbf{PR}(z, E'_1) \text{ ou } (z=y') \text{ ou } \mathbf{PR}(z, E'_2)=(z=x) \text{ ou } \textit{faux}. \tag{3}$$

Le second membre s'écrit encore *faux ou* $(z=x)$ *ou faux*.

(3) est vérifié à condition de faire les hypothèses $\mathbf{PR}(z, E'_1)=$ *faux*, $\mathbf{PR}(z, E'_2)=$ *faux* et $y'=x$. Les deux premières hypothèses se ramènent à $E'_1=$**VIDE** et $E'_2=$**VIDE**.

Dans le cas $E=$**VIDE**, une définition explicite de $E+x$ est donc

$$\boxed{\begin{array}{l}\mathbf{VIDE}+x=(E'_1, y', E'_2),\\ \quad \textit{avec } E'_1=\mathbf{VIDE},\ y'=x,\ E'_2=\mathbf{VIDE}.\end{array}} \tag{I}$$

Deuxième cas : $E=\langle E_1, y, E_2 \rangle$:

$$\mathbf{PR}(z, E'_1) \text{ ou } (z=y') \text{ ou } \mathbf{PR}(z, E'_2)$$
$$=(z=x) \text{ ou } (\mathbf{PR}(z, E_1) \text{ ou } (z=y) \text{ ou } \mathbf{PR}(z, E_2)). \tag{4}$$

(4) est vérifié si l'on fait les hypothèses

$$\mathbf{PR}(z, E'_1)=(z=x) \text{ ou } \mathbf{PR}(z, E_1),$$
$$y'=y,\ \mathbf{PR}(z, E'_2)=\mathbf{PR}(z, E_2).$$

La première hypothèse est vérifiée si l'on suppose $E'_1=E_1+x$ tandis que la dernière l'est si l'on suppose $E'_2=E_2$.

Une première définition est donc

$$\boxed{\begin{array}{l}\langle E_1, y, E_2 \rangle+x=\langle E'_1, y', E'_2 \rangle,\\ \quad \textit{avec } E'_1=E_1+x,\ y'=y,\ E'_2=E_2.\end{array}} \tag{II}$$

Une deuxième tout à fait symétrique étant

$$\boxed{\begin{array}{l}\langle E_1, y, E_2 \rangle+x=\langle E'_1, y, E'_2 \rangle,\\ \quad \textit{avec } E'_1=E_1,\ y'=y,\ E'_2=E_2+x.\end{array}} \tag{II'}$$

En toute rigueur, il faut aussi considérer d'autres hypothèses telles que

$$y'=x,\ E'_1=E_1+y,\ E'_2=E_2,$$

ou

$$y'=x,\ E_1'=E_1,\ E_2'=E_2+y,$$

Cependant dans la seconde partie où ces hypothèses sont testées, on verra que les solutions (II) et (II') sont suffisantes. Un problème de stratégie se pose ici pour présenter d'abord les solutions les plus « intéressantes ». Un critère peut être de faire apparaître la récursivité, en changeant le minimum d'argument. Ainsi le couple (E_1, x) est plus proche de $(\langle E_1, y, E_2 \rangle, x)$ que le couple (E_1, y).

Deuxième partie : **Calcul des préconditions**

Ce calcul est trivial dans le cas I ($E=$**VIDE**). Lorsque $E=\langle E_1, y, E_2 \rangle$, considérons uniquement la définition (II) qui peut encore s'écrire $\langle E_1, y, E_2 \rangle + x = \langle E_1 + x, y, E_2 \rangle$. Il s'agit de trouver un prédicat $\Phi(x, E)$ tel que l'implication

$$\text{pre}(\langle E_1, y, E_2 \rangle + x) \text{ et } \Phi(x, \langle E_1, y, E_2 \rangle) \Rightarrow \text{pre}(\langle E_1 + x, y, E_2 \rangle)$$

soit vérifiée.

En d'autres termes

$$E_1 < y \text{ et } y < E_2 \text{ et non } \mathbf{PR}\ (x, \langle E_1, y, E_2 \rangle)$$

$$\text{et } \Phi\ (x, \langle E_1, y, E_2 \rangle) \Rightarrow \text{non } \mathbf{PR}\ (x, E_1) \text{ et } E_1 + x < y \text{ et } y < E_2.$$

Il faut développer $E_1 + x < y$ ce qui donne (d'après les calculs effectués au paragraphe 1.3) :

$$E_1 + x < y = E_1 < y \text{ et } x < y.$$

D'autre part, non **PR** $(x, \langle E_1, y, E_2 \rangle)$ est égal à non **PR** (x, E_1) et non $(x=y)$ et non **PR** (x, E_2) et implique donc non **PR** (x, E_1).

Finalement, on peut choisir pour Φ l'assertion $x<y$. De même la définition (II') suppose la précondition $x>y$. Enfin le cas $x=y$ est exclu par la précondition non **PR** $(x, \langle E_1, y, E_2 \rangle)$. On obtient finalement la définition suivante :

$$\langle E_1, y, E_2 \rangle + x = \text{cas } x<y \text{ alors } \langle E_1 + x, y, E_2 \rangle,\ x>y \text{ alors } \langle E_1, y, E_2 + x \rangle \text{ fcas.}$$

Cette définition est à terminaison bornée puisque les arguments de + sont plus simples à droite qu'à gauche.

2.6. Définition explicite de EXTRAIT

On abrège ici **EXTRAIT** (E, x) en $E-x$.

Première partie : **construction.**

La spécification à satisfaire est $E+y-x\sim$ *si* $x=y$ *alors* E *sinon* $E-x+y$ *fsi.*

Nous allons guider notre construction sur une spécification plus restreinte : $E+x-x\sim E$ dont on verra tout à l'heure qu'elle est suffisante. Schématiquement, il s'agit, connaissant une définition explicite de **INSERE**, d'en construire une réciproque.

(*a*) **Cas simples.**

Nous étudions d'abord le cas particulier où $E=E'+x$ (avec $E'=\langle E'_1, y', E'_2\rangle$) et imposons dans ce cas $E'+x-x=E'$.

Soit, en utilisant la définition de **INSERE** trouvée précédemment.

$$\left.\begin{array}{l} cas\ x<y'\ alors\ \langle E'_1+x,\ y',\ E'_2\rangle-x \\ \quad x>y'\ alors\ \langle E'_1,\ y',\ E'_2+x\rangle-x\ fcas \end{array}\right\}=\langle E'_1, y', E'_2\rangle. \qquad (1)$$

Par récurrence on peut encore écrire $E'_1=E'_1+x-x$ et $E'_2=E'_2+x-x$. On obtient, dans le cas $x<y'$:

$$\langle E'_1+x,\ y',\ E'_2\rangle-x=\langle E'_1+x-x,\ y',\ E'_2\rangle$$

et dans le cas $x>y'$:

$$\langle E'_1,\ y',\ E'_2+x\rangle-x=\langle E'_1,\ y',\ E'_2+x-x\rangle.$$

On peut encore généraliser ces équations en posant $E_1=E'_1+x$; $E_2=E'_2+x$ ce qui donne, en remplaçant également y' par y :

$$\begin{array}{l} \langle E_1, y, E_2\rangle-x= \\ \quad cas\ x<y\ alors\ \langle E_1-x, y, E_2\rangle, \\ \qquad x=y\ alors... \\ \qquad x>y\ alors\ \langle E_1, y, E_2-x\rangle\ f cas. \end{array} \qquad (I)$$

Figure 2. – Définition de EXTRAIT : 1re partie.

(*b*) **Cas où l'élément à extraire est le médian.**

Soit à calculer $E-y$ dans le cas où $E=\langle E_1, y, E_2\rangle$. La démarche précédente n'est plus adaptée. Par contre, il est possible de partir de la spécification $E'+y-y\sim E'$, en posant ici $E'=E-y$. La définition de $E-y$ doit permettre de prouver l'équivalence $E-y+y-y\sim E-y$. On peut donc proposer comme spécification initiale l'assertion

$$E-y+y\sim E. \qquad (1)$$

(Remarquons que cette dernière équivalence est un peu différente de la précédente.)

Explicitons partiellement la définition de $E-y$ en posant

$$E-y=\langle E'_1, y', E'_2\rangle$$

et développons la définition de l'équivalence. Il vient, en tenant compte des définitions de $\mathbf{PR}(z, \langle E'_1, y', E'_2\rangle+y)$ et de $\mathbf{PR}(z, \langle E_1, y, E_2\rangle)$:

$$\mathbf{PR}(z, E'_1) \textit{ ou } (z=y') \textit{ ou } \mathbf{PR}(z, E'_2) \textit{ ou } (z=y)$$
$$=\mathbf{PR}(z, E_1) \textit{ ou } (z=y) \textit{ ou } \mathbf{PR}(z, E_2). \quad (2)$$

L'assertion (2) est complètement vérifiée si l'on ajoute par exemple les hypothèses

$$\left.\begin{array}{c}\mathbf{PR}(z, E_1)=\mathbf{PR}(z, E_1),\\ \mathbf{PR}(z, E'_2) \textit{ ou } (z=y')=\mathbf{PR}(z, E_2).\end{array}\right\} \quad (I)$$

ou encore les hypothèses équivalentes

$$\left.\begin{array}{c}E'_1=E_1\\ E'_2+y'\sim E_2.\end{array}\right\} \quad (I')$$

Une définition partiellement explicitée de $\langle E_1, y, E_2\rangle-y$ est donc

$$\begin{array}{c}\langle E_1, y, E_2\rangle-y=\langle E'_1, y', E'_2\rangle,\\ \textit{avec } E'_1=E_1 \textit{ et } E'_2+y'\sim E_2.\end{array} \quad (II)$$

Avant de chercher une solution de (II), calculons les préconditions associées. On obtient :

$$E_1<y<E_2 \textit{ et } \mathbf{PR}(y, \langle E_1, y, E_2\rangle) \Rightarrow E'_1<y'<E'_2 \textit{ et non } \mathbf{PR}(y', E'_2).$$

Le membre gauche se simplifie en $E_1<y<E_2$. De même l'assertion $y'<E'_2$ se réécrit $\forall z(\mathbf{PR}(z, E'_2)\Rightarrow y'<z)$. Pour $z=y'$ on trouve :

$$\mathbf{PR}(y', E'_2) \Rightarrow \textit{faux}$$

soit *non* $\mathbf{PR}(y', E'_2)$. Enfin, l'assertion $E'_1<y'$ est conséquence de

$$E'_1=E_1,\ E_1<y<E_2 \text{ et } \mathbf{PR}(y', E_2).$$

Le calcul des préconditions se réduit finalement à

$$E_1<y<E_2 \Rightarrow y'<E'_2$$

et donc le problème se ramène à calculer E'_2 et y' vérifiant les spécifications

$$\left.\begin{array}{l} E'_2 + y' \sim E_2, \\ y' < E'_2. \end{array}\right\} \qquad \text{(II')}$$

(*c*) *Sous-problème :* **Calcul d'un élément minimum**

Pour un dictionnaire E quelconque, désignons par **MIN**(E) et **REST**(E) les éléments z et E' vérifiant les spécificications

$$\left.\begin{array}{c} E' + z \sim E, \\ z < E'. \end{array}\right\} \qquad \text{(II'')}$$

Traitons successivement les cas $E =$ **VIDE** et $E = \langle E_1, y, E_2 \rangle$. Immédiatement l'on trouve $E' + z \sim$ **VIDE** $=$ *faux*. Il faut donc poser

$$\text{pre}\,(\textbf{MIN}\,(E)) = \text{pre}\,(\textbf{REST}\,((E)) = \textit{non}\ \textbf{ESTVIDE}\ (E)$$

et la solution (II) précédente ne convient que si E_2 n'est pas vide.

Soit maintenant $E = \langle E_1, y, E_2 \rangle$. Posons $E' = \langle E'_1, y', E'_2 \rangle$ et développons l'assertion $E' + z \sim E$:

$$((t = z)\ \textit{ou}\ \textbf{PR}(t, E'_1)\ \textit{ou}\ (t = y')\ \textit{ou}\ \textbf{PR}(t, E'_2))$$
$$= \textbf{PR}(t, E_1)\ \textit{ou}\ (t = y)\ \textit{ou}\ \textbf{PR}(t, E_2)). \quad (3)$$

(3) est satisfaite si l'on adopte les hypothèses $E'_2 = E_2$, $y' = y$ et $E'_1 + z \sim E_1$.

Ainsi, lorsque E_1 est non vide, on peut poser $z =$ **MIN**(E_1) et $E'_1 =$ **REST**(E_1). La condition $z < E'$ se réécrit :

$$z < E'_1 \ \textit{et}\ z < y \ \textit{et}\ z < E_2.$$

La première inéquation résulte des spécifications de **MIN** et **REST**, la seconde des propriétés **PR**(z, E_1) et $E_1 < y$ et la troisième de ce que $y < E_2$.

Lorsque E_1 est vide, on peut encore écrire :

$$(t = z)\ \textit{ou}\ \textbf{PR}(t, E') = (t = y)\ \textit{ou}\ \textbf{PR}(t, E_2)$$

et adopter comme solution $z = y$ et $E' = E_2$.

Le sous-problème posé est donc entièrement résolu par l'équation

MIN$(\langle E_1, y, E_2 \rangle)$, **REST**$(\langle E_1, y, E_2 \rangle) =$
cas **ESTVIDE** (E_1) *alors* y, E_2,
non **ESTVIDE** (E_1) *alors* **MIN**(E_1), $\langle$ **REST** $(E_1), y, E_2 \rangle$ *fcas*.

(*d*) *Un dernier cas :* $\mathbf{E} = \langle E_1, \mathbf{y}, \mathbf{VIDE} \rangle$

Dans ce cas, la spécification de E' peut s'écrire :

$$\textbf{PR}(z, E')\ \textit{ou}\ (z = y) = \textbf{PR}(z, E_1)\ \textit{ou}\ (z = y)$$

et la solution $E' = E_1$ convient trivialement.

On complète ainsi la définition **EXTRAIT**.

$\langle E_1, y, E_2 \rangle - y =$
cas **ESTVIDE** (E_2) *alors* E_1,
non **ESTVIDE** (E_2) *alors* $\langle E_1, \mathbf{MIN}(E_2), \mathbf{REST}(E_2) \rangle$ *fcas*,
avec...

Figure 3. – Fin de la définition de EXTRAIT.

(*e*) Vérification des autres spécifications

La définition explicite proposée en I permet de prouver que de manière générale $E + x - x = E$ (seul le cas $E =$ **VIDE** demande une preuve particulière). On sait de plus que $E - x + x \sim E$ lorsque x est médian de E. Cette assertion peut être aussi prouvée à partir de (I) dans les autres cas. Enfin on montre de manière très simple que $E + x + y \sim E + y + x$ lorsque $x \neq y$. En effet

$$\mathbf{PR}(z, E + x + y) = (\mathbf{PR}(z, E) \text{ ou } (z = x) \text{ ou } (z = y)) = \mathbf{PR}(z, E + y + x).$$

Vérifions maintenant que $E + y - x \sim E - x + y$ lorsque $x \neq y$:

$$E + y - x \sim (E - x + x) + y - x \sim E - x + y + x - x \sim E - x + y.$$

(*f*) Commentaires

La construction de la définition récursive de **EXTRAIT** nous a demandé un peu plus d'imagination que pour **INSERE**. Dans les cas simples, nous avons utilisé un argument d'induction ou de généralisation (en partant de cas particuliers) qui est couramment utilisé par d'autres auteurs (Burstall et Darlington [7]). Dans les cas plus difficiles, nous avons appliqué la spécification de **EXTRAIT** non directement à E, mais à $E - x$, supposant en quelque sorte le problème résolu. De plus une fois obtenue une spécification partiellement explicitée, nous avons immédiatement effectué un calcul de préconditions permettant de renforcer cette spécification. Enfin il est significatif que cette démarche permette de trouver des solutions d'abord dans les cas les plus fréquents (E_2 non vide dans le problème principal, E_1 non vide dans le sous problème) les exceptions recevant ici des solutions plus simples.

3. ÉVALUATION ET AMÉLIORATION DE DÉFINITIONS RÉCURSIVES

3.1. Introduction

Nous allons nous intéresser ici à l'**évaluation abstraite** d'une définition récursive, qui permet de dénombrer le nombre d'utilisation d'une certaine

opération de base du type abstrait dans un calcul. On donnera une technique permettant de déduire systématiquement d'une équation donnée d'autres équations d'évaluation.

Puis, dans l'exemple qui nous intéresse, on effectue des majorations de ces quantités, introduisant pour cela une nouvelle fonction, la hauteur d'un dictionnaire, elle-même définie récursivement. On s'attache ensuite à calculer une définition récursive de l'opération **INSERE** qui minimise la hauteur des arbres résultants. Disposant d'une première définition, il suffit de procéder par équivalence puisque, si E' et E'' sont équivalents et si E' vérifie les spécifications de **INSERE** $(E+x)$, il en est de même de E''.

A cet effet on recherche des couples de dictionnaires équivalents ou, si l'on préfère, des règles de transformations préservant l'équivalence. Il reste alors à s'assurer que ces règles sont suffisantes et à caractériser les dictionnaires obtenus.

Des dictionnaires améliorés ont été proposés pour la première fois par Adel'son-Vel'skii et Landis [1] et portent dans la littérature le nom d'arbres AVL (*voir* aussi le livre de Aho, Hopcroft et Ullman [2] et celui de Knuth [19]). La présentation qui en est faite ici est à notre connaissance relativement originale et les méthodes proposées doivent pouvoir s'étendre à d'autres problèmes de représentations.

3.2. Évaluation

Plaçons-nous dans le cas simple d'une seule inconnue f définie par un système d'équations $S(f)=\{f(g_i(x))=u_i(x)\}$ et soit g une opération de base du type abstrait. On désire définir formellement, pour un x donné, le nombre $C_g(f(x))$ d'utilisation de l'opération g dans le calcul de $f(x)$ [que l'on notera encore $C(f(x))$ ou $\hat{f}(x)$ si g est déterminé par le contexte]. Pour cela associons plus généralement à chaque expression $u(x)$ une expression $\hat{u}(x)$ de type entier représentant le nombre d'utilisation de g dans le calcul de $u(x)$. L'expression $\hat{u}(x)$ est définie par récurrence sur la complexité des expressions : $\hat{x}=0$ si x est une variable

$$\widehat{gu_1 \ldots u_m} = 1 + \sum_{i=1}^{m} \hat{u}_i,$$

$$\widehat{hu_1 \ldots u_m} = \sum_{i=1}^{m} \hat{u}_i \text{ si } h \neq g \text{ et } h \neq f,$$

$$\widehat{fu_1 \ldots u_n} = \hat{f}(u_1, \ldots, u_n),$$

$$\widehat{\textit{si } b \textit{ alors } u_1 \textit{ sinon } u_2 \textit{ fsi}} = \hat{b} + (\textit{si } b \textit{ alors } \hat{u}_1 \textit{ sinon } \hat{u}_2 \textit{ fsi}).$$

Ces formules permettent d'associer au système $S(f)$ un système similaire $S(\hat{f})=\{\hat{f}(g_i(x))=\hat{u}_i(x)\}$ qui est à terminaison bornée si $S(f)$ l'est.

Nous ne voulons pas développer plus les généralités sur ces fonctions d'évaluations mais présenterons seulement un exemple.

Dans le type dictionnaire soit g le prédicat $<$ et $f=$ **PRESENT**. La fonction $\hat{f}$ est définie par les équations

$$\hat{f}(x, \text{VIDE})=0,$$
$$\hat{f}(x, \langle E_1, y, E_2 \rangle)=1+cas\ x<y\ alors\ \hat{f}(x, E_1),$$
$$x>y\ alors\ \hat{f}(x, E_2)\ fcas.$$

Nous pouvons maintenant rechercher soit une borne supérieure soit une valeur moyenne de $\hat{f}(x, E)$ lorsque x parcourt l'ensemble des identificateurs. Contentons-nous ici de calculer une borne supérieure.

On peut écrire :

$$\operatorname*{Max}_x \hat{f}(x, \langle E_1, y, E_2 \rangle)=1+\max\{\operatorname*{max}_{x<y} \hat{f}(x, E_1), \operatorname*{Max}_{x>y} \hat{f}(x, E_2)\}.$$

Mais on peut encore remplacer $\operatorname*{Max}_{x<y} \hat{f}(x, E_1)$ par $\operatorname*{Max}_x \hat{f}(x, E_1)$ et de même

$\operatorname*{Max}_{x>y} \hat{f}(x, E_2)$ par $\operatorname*{Max}_x \hat{f}(x, E_2)$.

(Cela ressort du fait que $E_1<y<E_2$.)

On peut donc écrire ici que $\mathbf{H}=\operatorname*{Max}_x \hat{f}$ est solution du système

$$\begin{cases} \mathbf{H}(\textbf{VIDE})=0, \\ \mathbf{H}(\langle E_1, y, E_2 \rangle)=1+\max(\mathbf{H}(E_1), \mathbf{H}(E_2)). \end{cases}$$

La quantité $\mathbf{H}(E)$ sera encore appelée **hauteur** du dictionnaire E.

3.3. Amélioration de la définition de INSERE : décomposition du problème

Il est naturel de chercher à améliorer la définition de **INSERE** (x, E) de telle sorte que la hauteur des dictionnaires obtenus par insertions successives soit aussi petite que possible. D'une part on peut supposer que $\mathbf{H}$(**INSERE** (x, E)) est plus grande que $\mathbf{H}(E)$. D'autre part on peut montrer, à partir de la définition calculée au paragraphe 2.5 que $\mathbf{H}$(**INSERE** $(x, E)) \leqq \mathbf{H}(E)+1$. Nous allons définir une opération **INSERE** (ou $\pm$ en notation infixée) telle que :

1° la hauteur de $E \pm x$ soit le plus souvent possible égale à celle de E et

2° $E \pm x \sim E + x$.

Compte tenu de la définition de l'équivalence, $\pm$ vérifie la même spécification que + puisque

PRESENT $(z, E+x) = \mathbf{PRESENT}\ (z, E+x) = (z = x)$ ou **PRESENT** (z, E).

Lorsque $E = \mathbf{VIDE}$, la définition de $E \pm x$ n'est pas améliorable et l'on pose

$$\mathbf{VIDE} \pm x = \langle \mathbf{VIDE}, x, \mathbf{VIDE} \rangle.$$

Dans l'autre cas, on peut décomposer le problème en posant :

$$\langle E_1, y, E_2 \rangle \pm x = \mathbf{ECRASE}(\langle E_1, y, E_2 \rangle \underset{\approx}{\pm} x),$$

où :

- **ECRASE** $(E) \sim E$ et $\mathbf{H}\ (\mathbf{ECRASE}\ (E)) < \mathbf{H}(E)$ le plus souvent possible,
- $\langle E_1, y, E_2 \rangle \underset{\approx}{\pm} x = cas\ x < y\ alors\ \langle E_1 \pm x, y, E_2 \rangle$
  $x > y\ alors\ \langle E_1, y, E_2 \pm x \rangle\ fcas.$

3.4. Définition de l'opération ECRASE

Un moyen de réduire la hauteur d'un dictionnaire en ne changeant pas ses éléments consiste à le rééquilibrer, c'est-à-dire à réduire en valeur absolue la différence des hauteurs de ses constituants. Nous avons besoin de définir la **balance** d'un dictionnaire non vide :

$$\mathbf{BAL}(\langle E_1, y, E_2 \rangle) = \mathbf{H}(E_1) - \mathbf{H}(E_2).$$

(*a*) Cas d'un dictionnaire presque équilibré

Lorsque $|\mathbf{BAL}\ (E)|$ est inférieure ou égale à 1, il y a peu de chance de réduire la hauteur de E. On pose donc dans ce cas

$$\mathbf{ECRASE}(E) = E.$$

(*b*) Une première transformation préservant l'équivalence

Pour trouver des couples (E, E') de dictionnaires équivalents, développons E suffisamment pour faire apparaître des réarrangements possibles. La décomposition $E = \langle E_1, y, E_2 \rangle$ n'est pas suffisante. En décomposant E_1 en $\langle E_{11}, y_1, E_{12} \rangle$ on trouve

$$\langle \langle E_{11}, y_1, E_{12} \rangle, y, E_2 \rangle \sim \langle E_{11}, y_1, \langle E_{12}, y, E_2 \rangle \rangle.$$

La justification est standard.

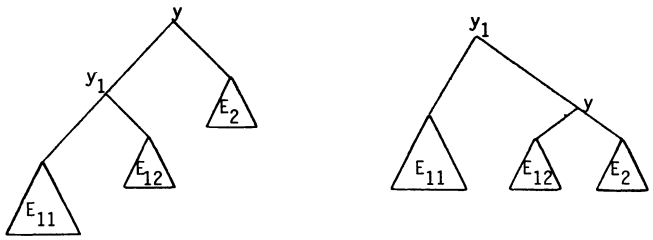

Figure 4. – Une première transformation.

Désignons par **ROTGD** l'opération (de **Rotation Gauche Droite**) définie par

$$\mathbf{ROTGD}(\langle\langle E_{11}, y_1, E_{12}\rangle, y, E_2\rangle) = \langle E_{11}, y_1, \langle E_{12}, y, E_2\rangle\rangle$$

et par **ROTDG** l'opération réciproque.

Déterminons à quelle condition **ROTGD** réduit la hauteur.

En désignant par E, E' l'argument et le résultat de **ROTGD** on peut écrire :

$$\mathbf{H}(E) = 1 + \mathrm{Max}(\mathbf{H}(E_2), \mathbf{H}(E_{11})+1, \mathbf{H}(E_{12})+1),$$
$$\mathbf{H}(E') = 1 + \mathrm{Max}(\mathbf{H}(E_{11}), \mathbf{H}(E_{12})+1, \mathbf{H}(E_2)+1),$$

d'où

$$\mathbf{H}(E') < \mathbf{H}(E) \Leftrightarrow \mathbf{H}(E) = \mathbf{H}(E_{11})+2 \quad \text{et}$$
$$\mathbf{H}(E_{12})+2 < \mathbf{H}(E) \text{ et}$$
$$\mathbf{H}(E_2)+2 < \mathbf{H}(E) \Leftrightarrow \mathbf{H}(E_{12}) < \mathbf{H}(E_{11}) \text{ et } \mathbf{H}(E_2)+2 \leqq \mathbf{H}(E_1).$$

Finalement,

$$\mathbf{H}(E') < \mathbf{H}(E) \Leftrightarrow \mathbf{BAL}(E) \geqq 2 \text{ et } \mathbf{BAL}(E_1) > 0$$

et symétriquement

$$\mathbf{H}(E') > \mathbf{H}(E) \Leftrightarrow \mathbf{BAL}(E) \leqq -2 \text{ et } \mathbf{BAL}(E_2) < 0.$$

Dans le premier cas on peut poser $\mathbf{ECRASE}(E) = \mathbf{ROTGD}(E)$ et dans le second $\mathbf{ECRASE}(E) = \mathbf{ROTDG}(E)$.

(*c*) **Une deuxième transformation**

Pour résoudre le cas où $\mathbf{BAL}(E) \geqq 2$ et $\mathbf{BAL}(E_1) \leqq 0$, développons un peu plus E en posant

$$E_{12} = \langle E_{121}, y_{12}, E_{122}\rangle,$$
$$\mathbf{ROTDG}(E_1) = \langle\langle E_{11}, y_1, E_{121}\rangle, y_{12}, E_{122}\rangle$$

et

$$\mathbf{BAL}(\mathbf{ROTDG}(E_1)) = 1 + \mathrm{Max}(\mathbf{H}(E_{11}), \mathbf{H}(E_{121})) - \mathbf{H}(E_{122})$$
$$\geqq 1 + \mathbf{H}(E_{11}) - \mathbf{H}(E_{122}).$$

Cette dernière quantité est strictement positive à condition que $\mathbf{H}(E_{11}) - \mathbf{H}(E_{122})$ soit positif ou nul. Pour des raisons de symétrie, supposons en fait que $\mathbf{H}(E_{11})$ est supérieure à $\mathrm{Max}(\mathbf{H}(E_{121}), \mathbf{H}(E_{122}))$ donc que $\mathbf{BAL}(E_1)$ est supérieure ou égale à -1. Plus précisément, si $\mathbf{BAL}(E_1) = -1$, $E_1' = \mathbf{ROTDG}(E_1)$ vérifie les propriétés $\mathbf{H}(E_1') = \mathbf{H}(E_1)$ et $\mathbf{BAL}(E_1') = +1$. Donc $E' = \mathbf{ROTGD}(\langle E_1', y, E_2 \rangle)$ vérifie la propriété $\mathbf{H}(E') = \mathbf{H}(E) - 1$.

On peut donc poser, dans le cas $\mathbf{BAL}(E) > 2$ et $\mathbf{BAL}(E_1) = -1$,

$$\mathbf{ECRASE}(\langle E_1, y, E_2 \rangle) = \mathbf{ROTGD}(\langle \mathbf{ROTDG}(E_1), y, E_2 \rangle)$$
$$= \mathbf{DROTGD}(\langle E_1, y, E_2 \rangle).$$

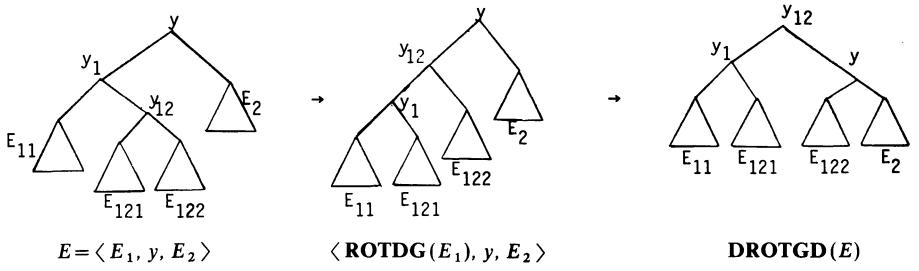

Figure 5. – Une deuxième transformation.

(*d*) **Les cas exclus**

Si l'on regroupe les résultats précédents, on obtient pour **INSERE** la définition donnée dans la figure 6.

Pour achever la construction de **INSERE**, il faut vérifier que le cas « $\mathbf{BAL}(E'') \geqq 2$ *et* ($\mathbf{BAL}(E_1'') = 0$ *ou* $\mathbf{BAL}(E_1'') \leqq -2$) » est exclu. Si l'on choisit E quelconque, cette situation peut se produire et il faut donc supposer que E est un dictionnaire obtenu à partir de **VIDE** par application répétitive de **INSERE**.

Il faut donc déterminer une propriété invariante $\Phi(E)$ telle que :

(1) Φ (**VIDE**) soit vérifiée;

(2) *si* $\Phi(E)$ *alors* Φ(**INSERE** (x, E));

(3) *si* $\Phi(\langle E_1, y, E_2 \rangle)$ *alors* $\Phi(E_1)$ *et* $\Phi(E_2)$,

INSERE$(x,$ **VIDE**$)=\langle$ **VIDE**, $x,$ **VIDE**$\rangle$,

INSERE$(x, \langle E_1, y, E_2 \rangle)=$

cas $x<y$ *alors soit* $E_1''=$**INSERE**(x, E_1) *et* $E''=\langle E_1'', y, E_2\rangle$,

cas $|$**BAL**$(E'')| \leqq 1$ *alors* E'',

BAL$(E'') \geqq 2$ *alors*

cas **BAL**$(E_1'') \geqq +1$ *alors* **ROTGD**(E''),

BAL$(E_1'')=-1$ *alors* **DROTGD**(E''),

fcas,

fcas,

$x>y$ *alors...*

fcas.

Figure 6. – Définition de INSERE

et vérifier de plus que

(4) *si* $\Phi(E)$ *et* **BAL**$(E'') \geqq 2$ *alors* **BAL**$(E_1'') \neq 0$;

(5) *si* $\Phi(E)$ *et* **BAL**$(E'') \geqq 2$ *alors* **BAL**$(E_1'') \geqq -1$,

avec des propriétés symétriques lorsque **BAL**$(E'') \leqq -2$.

Pour que (5) soit vérifiée de manière immédiate, il suffit, compte tenu de (2) et (3) de choisir Φ telle que

$$\Phi(E) \Rightarrow \textbf{BAL}(E) \geqq -1.$$

En effet $E_1''=$**INSERE** (x, E_1) et $\Phi(E) \Rightarrow \Phi(E_1) \Rightarrow \Phi(E_1'')$.

Pour des raisons de symétrie et pour vérifier (1) et (3) on peut prendre pour Φ(E) le plus faible prédicat vérifiant :

$\Phi($**VIDE**$)=vrai$

et $\Phi(\langle E_1, y, E_2\rangle)=|\textbf{H}(E_1)-H(E_2)|<1$ *et* $\Phi(E_1)$ *et* $\Phi(E_2)$.

On écrit encore $\Phi(E)$ sous la forme 'E est un AVL'.

Reste à vérifier les propriétés (2) et (4).

PROPRIÉTÉ 2 : **Si** E **est un** AVL **alors** $E \pm x$ **est un** AVL.

Par hypothèse de récurrence E_1'' et E_2 sont des AVL. Donc si $|$**BAL**$(E'')| \leqq 1$, E'' est un AVL; donc aussi $E \pm x$ puisque $E \pm x = E''$. Sinon **BAL** (E'') est égal à $+2$ puisque l'insertion n'augmente la hauteur que de 1 au maximum.

Deux cas sont à envisager (d'après la propriété 4) :

BAL $(E_1'') = +1$: alors $E' = \mathbf{ROTGD}\ (E'')$ vérifie les propriétés suivantes :

- $\mathbf{H}\ (E') = \mathbf{H}\ (E'') - 1$ [et donc $\mathbf{H}\ (E') = \mathbf{H}\ (E)$],
- **BAL** $(E') = 0$,
- E_1' et E_2' sont des AVL.

Donc $E \pm x$ est un AVL puisque $E \pm x = E'$:

BAL $(E_1'') = -1$: alors $E' = \mathbf{ROTGD}\ (E'')$ vérifie les mêmes propriétés.

PROPRIÉTÉ 4 : **Si** E **est un** AVL **et si** **BAL** $(E'') \geqq 2$ **alors** **BAL** $(E_1'') \neq 0$.

Puisque E est un AVL, **BAL** $(E) \leqq 1$. Donc lorsque **BAL** $(E'') \geqq 2$, $\mathbf{H}\ (E_1'')$ est strictement supérieure à $\mathbf{H}\ (E_1)$. La propriété 4 est donc une conséquence immédiate de la propriété suivante.

PROPRIÉTÉ 4' : **Si** E **est un** AVL **et si** $\mathbf{H}\ (E \pm x) > \mathbf{H}\ (E)$ **alors** **BAL** $(E \pm x) \neq 0$.

Ou bien $E \pm x = E''$ est dans ce cas **BAL** (E'') est différent de 0.

Ou bien $E \pm x \neq E''$ et dans ce cas $\mathbf{H}\ (E \pm x) = \mathbf{H}\ (E'') - 1 = \mathbf{H}\ (E)$.

3.5. Conclusion et nouvelle évaluation

Pour construire la définition de INSERE, nous avons déterminé plusieurs transformations préservant l'équivalence. Une fois que nous avons obtenu une définition « assez riche », nous avons calculé un invariant de l'opération et vérifié à partir de cet invariant que la définition était complète, c'est-à-dire recouvrait tous les cas envisageables.

Pour terminer, donnons une évaluation de INSERE. Il suffit de calculer la hauteur d'un AVL; celle-ci vérifie les équations suivantes :

$$\mathbf{H}\ (\mathbf{VIDE}) = 0, \tag{1}$$

$$\mathbf{H}\ (\langle E_1, y, E_2 \rangle) = 1 + \mathrm{Max}\ (\mathbf{H}\ (E_1), \mathbf{H}\ (E_2)) \leqq 2 + \mathrm{Min}(\mathbf{H}\ (E_1), \mathbf{H}\ (E_2)), \tag{2}$$

puisque $|\mathbf{H}\ (E_1) - \mathbf{H}\ (E_2)|$ est inférieure ou égale à 1.

Définissons le nombre $\mathbf{N}\ (E)$ d'éléments d'un dictionnaire E par les formules

$$\mathbf{N}\ (\mathbf{VIDE}) = 0,$$

$$\mathbf{N}\ (\langle E_1, y, E_2 \rangle) = 1 + \mathbf{N}\ (E_1) + \mathbf{N}\ (E_2),$$

et posons

$$\mathbf{HMAX}\ (N) = \mathrm{Max}\ \{\mathbf{H}\ (E) \mid \mathbf{N}\ (E) \leqq N\}.$$

Remarquons aussi que

$$\mathbf{N}(\langle E_1, y, E_2 \rangle) > 2 \star \text{Min}(\mathbf{N}(E_1), \mathbf{N}(E_2)). \quad (3)$$

On vérifie que

$$\text{Min}(\mathbf{H}(E_1), \mathbf{H}(E_2)) \leqq \text{Min}(\mathbf{HMAX}(\mathbf{N}(E_1)), \mathbf{HMAX}(\mathbf{N}(E_2))) \leqq \mathbf{HMAX}([\mathbf{N}(E)/2]), \quad (4)$$

où $[x]$ désigne le plus grand entier inférieur ou égal à x.

Donc

$$\mathbf{HMAX}(N) \leqq 2 + \mathbf{HMAX}([N/2]) \quad (5)$$

et

$$\mathbf{HMAX}(0) = 0. \quad (6)$$

Remarquons que la fonction Log_+ définie par $\text{Log}_+(N) = \text{Max}(0, \text{Log}(N))$ (où Log désigne le logarithme à base 2) vérifie les relations

$$\text{Log}_+(0) = 0 = \mathbf{HMAX}(0), \quad \text{Log}_+(1) = 0 = \mathbf{HMAX}(1) - 1, \quad (7)$$

$$\text{Log}_+(N/2) = \text{Log}_+(N) - 1 \text{ pour } N \geqq 2. \quad (8)$$

On en déduit que $\mathbf{HMAX}(N)$ est majorée par $2\,\text{Log}_+(N) + 1$ pour tout entier N :

Pour tout AVL E : $\mathbf{H}(E) \leqq 2\,\text{Log}_+(\mathbf{N}(E)) + 1$.

Il s'agit d'une majoration intéressante puisque dans le cas le plus général on peut avoir $\mathbf{H}(E) = \mathbf{N}(E)$ et qu'il est facile de vérifier par ailleurs que $\mathbf{H}(E)$ est toujours supérieure à $\text{Log}_+(\mathbf{N}(E) + 1)$.

4. REPRÉSENTATION DES DICTIONNAIRES PAR DES STRUCTURES D'UN LANGAGE DE PROGRAMMATION

4.1. Présentation d'un langage

On se propose de représenter les dictionnaires par des objets structurés de Algol 68. Un travail semblable pourrait être réalisé dans le cadre de Pascal, encore qu'un certain nombre de restrictions de ce langage rende l'écriture des programmes plus malaisée.

A ce niveau, un traitement rigoureux des problèmes de représentations par des objets d'un langage de programmation nécessiterait de spécifier ces objets

comme des types abstraits, c'est-à-dire encore, de se donner une sémantique d'Argol 68 Un tel objectif dépasse le cadre de cet article [12]

En Algol 68, les dictionnaires peuvent être représentés par des objets de mode **dict**, lui-même défini récursivement par la déclaration

mode dict = rep triplet,

mode triplet = struct (dict gauche, **id** median, **dict** droite)

Les dictionnaires sont donc des **noms** reperant des triplets de sélecteurs respectifs gauche, médian, droite Il est indispensable de considérer les dictionnaires comme des noms pour eviter de traiter des objets de taille arbitraire Le dictionnaire **VIDE** est represente par le nom particulier **nil** qui ne repère rien De plus, si $e1$, $e2$ representent les dictionnaires $E1$, $E2$ et si y est un identificateur, alors tout nom reperant le triplet $e=(e1, y, e2)$ représente le dictionnaire $\langle E1, y, E2 \rangle$

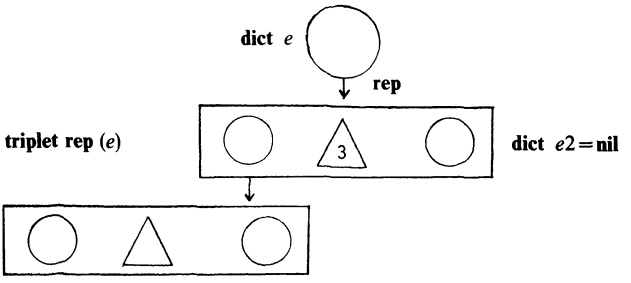

Figure 7. – Exemple d'objets ALGOL 68.

On note **rep** (e) le triplet reperé par un dictionnaire e Une transformation Algol 68, le **dérepérage** permet de confondre dans des conditions de contexte assez large un dictionnaire et le triplet qu'il repere Donc si e repere le triplet ($e1$, y, $e2$), on peut ecrire les equations

$e1 =$ gauche **de** e, $y =$ median **de** e, $e2 =$ droite **de** e

Les modes Algol 68 sont des types abstraits au même sens que **ENS** et **DICT**. Comme tout systeme axiomatique ils possedent des interpretations (ou modeles), concretes Ainsi, un objet de **mode** dict peut être interprete comme un ensemble de points (tous les noms accessibles en utilisant les selecteurs gauche et droite) et d'arêtes c'est-a-dire comme un graphe

Pour qu'un dictionnaire e puisse etre interprete comme un arbre, il est nécessaire que les ensembles de noms associes a gauche **de** e et droite **de** e soient disjoints Il est donc necessaire de pouvoir creer de nouveaux noms ayant une

portée globale. On utilise en Algol 68 un **générateur global**. Précisément, la déclaration **tas triplet** *e* engendre un nouveau nom, de mode **dict**, désigné par l'identificateur *e*.

L'**affectation** (: =) permet de définir la fonction **rep**. Si *e* est un dictionnaire et *t* un triplet, **rep**(*e*) est égal à *t* après l'affectation *e* : = *t*.

Enfin, la **relation d'identité** (: = :) permet de comparer deux noms de même mode.

Après cette présentation, nécessairement incomplète, de quelques éléments d'Algol 68, on peut donner une représentation des opérations sur les dictionnaires par des procédures. On proposera successivement une procédure à résultat calquée sur la définition de **INSERE**, puis une procédure sans résultat procédant par modification de la donnée.

4.2. Représentation de INSERE

INSERE peut être traduite par la procédure Algol 68 de la figure 8.

```
proc insere=(id x, dict e) dict :
  début tas triplet e';
    e' : =si e : = : nil alors (nil, x, nil),
    sinon si x < médian de e,
          alors (insere (x, gauche de e), médian de e, droite de e),
          sinon (gauche de e, médian de e, insere (x, droite de e)),
          fsi,
    fsi;
    e'
  fin.
```

Figure 8. — Procédure ALGOL 68 pour l'insertion.

Dans cette déclaration, le résultat est le nom *e'* qui est la valeur de la phrase située entre début et fin.

Cette déclaration est une traduction très littérale de la définition. Examinons maintenant comment transformer cette déclaration pour obtenir une procédure sans résultat ou, si l'on veut, confondant la donnée initiale et le résultat final. Traduisons, pour cela, l'affectation

e' : = (insere (*x*, gauche **de** *e*), médian **de** *e*, droite **de** *e*)

par le triplet

$$\text{gauche } \textbf{de } e' := \text{insere } (x, \text{ gauche } \textbf{de } e), \quad (1)$$

$$\text{médian } \textbf{de } e' := \text{médian } \textbf{de } e, \quad (2)$$

$$\text{droite } \textbf{de } e' := \text{droite } \textbf{de } e'. \quad (3)$$

Introduisons alors un nouveau nom $e1$ de mode **rep dict** et construisons une procédure insere 1, de paramètres x et $e1$, sans résultat et telle que, si e_1 repère e initialement, alors e_1 repère e' finalement. On peut dire que :

(1) si e_1 repère initialement **nil** alors l'effet de insere 1 (x, e_1) est le même que celui de l'affectation

$$e_1 := \text{tas triplet} := (\textbf{nil}, x, \textbf{nil});$$

(2) si x est plus petit que médian **de** e_1, l'effet de insere (x, e_1) est le même que celui de insere $(x,$ gauche **de** $e_1)$;

(3) et de même si x est plus grand que médian **de** e_1.

Pour exprimer que e_1 repère **nil**, on effectue une relation d'identité entre le nom repéré par e_1 (noté **dict** : e_1) et **nil**. On obtient le texte suivant:

```
proc insere 1=(id x, rep dict e1) neutre :
  si dict : e1 : = : nil alors e1 : = tas triplet : = (nil, x, nil),
  sinon si x < médian de e1 alors insere 1 (x, gauche de e1),
                              sinon insere 1 (x, droite de e1) fsi,
  fsi.
```

5. CONCLUSION

Cet article présente essentiellement un exemple de spécification de types abstraits et de construction d'une fonction de représentation. Depuis sa rédaction initiale, nous nous sommes employés à généraliser les méthodes utilisées.

En ce qui concerne la spécification d'un type abstrait, nous préférons utiliser des règles de réécriture $(u \to v)$ plutôt que des axiomes $(u = v)$. On peut de cette manière réduire grandement le nombre de techniques de transformation d'énoncés et de construction de définitions explicites en se limitant presque exclusivement aux techniques de pliement-dépliement de Burstall et Darlington.

Une approche algébrique nous conduit à distinguer sur l'ensemble des expressions d'un type abstrait plusieurs relations d'égalité. L'égalité forte, ou minimale, engendrée par les règles de réécriture et l'égalité faible ou maximale

qui imposent seulement que les opérations externes donnent le même résultat. Le prédicat d'équivalence que nous avons abondamment utilisé dans la construction puis l'amélioration d'une représentation n'est rien d'autre que cette égalité faible. Dans la même direction, nous étudions une définition des représentations plus large que celle utilisée jusqu'à présent.

Nous développons enfin un système qui puisse transformer des spécifications implicites assez simples en des définitions récursives. Nous pensons utiliser un certain nombre des techniques de Burstall et Darlington [7] et Feather [11] en les adaptant au cas où les données ne sont pas explicites mais implicites.

Diverses équipes travaillent dans des directions similaires ou dans des domaines reliés. A Nancy, une équipe développe un projet centré sur la méthodologie de la programmation, autour du langage Medee [6, 13, 26, 24]. Citons également les travaux de Arsac [3], de Ashcroft et Wadge [4] et le projet CIP de Munich [5]. Toujours à Nancy des études plus théoriques sont également menées sur la sémantique des langages de programmation [12] et le calcul relationnel comme outils de spécification [20].

Gaudel et Terrine utilisent le même cadre équationnel pour développer très systématiquement des représentations de structure de données. Les définitions récursives sont également très utilisées par Darlington [9, 10]. Citons aussi les travaux de Guttag [17] et de Jones [18].

Gaudel utilise également un formalisme voisin pour décrire la sémantique d'un langage source et d'un langage objet et définit un compilateur comme une fonction de représentation entre deux types abstraits [14].

Ce travail a été réalisé dans le cadre de l'équipe Castor de Nancy et je tiens à remercier particulièrement, en dehors des personnes déjà citées, F. Bellegarde, J. Guyard et J. Jaray pour leurs conseils et critiques ainsi que M. Sintzoff qui m'a incité à relier les problèmes de types abstraits et ceux d'évaluation des structures de données et C. Pair pour ses nombreuses suggestions sur les aspects les plus théoriques de ce travail.

BIBLIOGRAPHIE

1. G. M. Adel'son-Vel'skii et Y. M. Landis, *An Algorithm for the Organization of Information*, Soviet Math. Dokl., vol. 3, 1962, p. 1259-1262.
2. A. V. Aho, J. E. Hopcroft et J. D. Ullman, *The Design and Analysis of Algorithms*, Addison-Wesley, Reading, Mass., 1974.
3. J. Arsac, *La construction de programmes structurés*, Dunod, Paris, 1977.
4. E. A. Ashcroft et W. W. Wadge, *Lucid, a Nonprocedural Language with Iteration*, Comm. A. C. M., vol. 20, n° 7, 1977, p. 519-526.

5. F. L. BAUER et H. WOSSNER, *Algorithmic Language and Program Development*, Prentice Hall International, London, 1979.

6. F. BELLEGARDE *et al.*, MEDEE, *A Type of Language for the Deductive Programming Method*, Conference on Reliable Software, German A. C. M. Chapter, Bonn, 1978.

7. R. M. BURSTALL et J. DARLINGTON, *A Transformation System for Developing Recursive Programs*, J. A.C.M., vol. 24, 1977, p. 44-67.

8. R. M. BURSTALL et J. A. GOGUEN, *Putting Theories Together to Make Specifications*, Proc. of I.F.I.P. Conference, 1977, p. 1045-1058.

9. J. DARLINGTON, *Program Transformation and Synthesis: Present Capabilities*, D.A.I. Research Report n° 48, University of Edinburgh, 1977.

10. J. DARLINGTON, *Program Transformation Involving Unfree Data Structures: an Example*, 3e Coll. Int. sur la programmation, Dunod, Paris, 1978, p. 203-217.

11. M. FEATHER, « *ZAP* » *Program Transformation System, Primer and Users' Manual*, D.A.I. Research Report n° 54, University of Edinburgh, 1978.

12. J. P. FINANCE, *Une formulation de la Sémantique des langages de programmation*, R.A.I.R.O., vol. 10, Paris, 1976, p. 8-12.

13. J. P. FINANCE, *De la spécification abstraite d'une donnée à sa représentation en mémoire*, Théorie et techniques de l'Informatique, actes de Congrès de l'A.F.C.E.T., t. 1, 1978, Hommes et Techniques, Paris.

14. M. C. GAUDEL, *A Formal Approach to Translation Specification, Information Processing* 1977, B. GILCHRIST, éd., North Holland, Amsterdam, 1977, p. 123-129.

15. M. C. GAUDEL et G. TERRINE, *Synthèse de la représentation d'un type abstrait par des types concrets*, Théorie et Techniques de l'Informatique, actes du Congrès de l'A.F.C.E.T., t. 1, 1978, Hommes et Techniques, Paris.

16. J. A. GOGUEN, J. W. THATCHER, E. G. WAGNER et J. B. WRIGHT, *Abstract Data Types as Initial Algebras and the Correctness of Data Representations*, Proc. Conf. on Computer Graphics, Pattern Recognition and Data Structure, mai 1975.

17. J. V. GUTTAG, E. HOROWITZ et D. R. MUSSER, *The Design of Data Type Specifications*, in *Current Trends in Programming Methodology*, IV, Data Structuring, R. T. YEH, éd., Prentice-Hall, Engl. Cliffs, New Jersey, 1978.

18. C. B. JONES, *Constructing a Theory of a Data Structure as an Aid to Program Development*, Acta Informatica, vol. 11, 1979, p. 119-128.

19. D. E. KNUTH, *The Art of Computer Programming*, 3, *Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

20. P. LESCANNE, *Un calcul relationnel pour les structures de données*, Rapport 76-R-029, Centre de Recherche en Informatique de Nancy, Nancy, 1976.

21. P. LESCANNE, *Étude algébrique et relationnelle des représentations de types abstraits*, thèse d'état, Nancy, 1979.

22. B. LISKOV et S. ZILLES, *Programming with Abstract Data Types*, SIGPLAN, Notices, vol. 9, n° 4, 1974.

23. Z. MANNA et R. WALDINGER, *Knowledge and Reasoning in Program Synthesis*, Artif. Intel. J., vol. 6, 1975, p. 175-208.

24. C. PAIR, *La construction des programmes*, Rapport 77-R-019, Centre de Recherche en Informatique de Nancy, Nancy, 1977.

25. C. PAIR et M. C. GAUDEL, *Les structures d'information et leurs représentations*, I.R.I.A., Rocquencourt, 1978.

26. A. QUERE, *Construction de Programmes Itératifs dans le cadre du langage MEDEE*, Convention Informatique Latine, Barcelone, 1979.

27. J. L. RÉMY, *Structures d'Information, formalisation des notions d'accès et de modifications d'une donnée*, Thèse 3e Cycle, Université de Nancy I, 1974.

28. W. P. DE RŒVER, *Operational, Mathematical and Axiomatized Semantics for Recursive Procedures and Data Structures*, Rapport ID 1/74, Math. Centrum, Amsterdam, 1974.

29. M. SINTZOFF, *Inventing Program Construction Rules*, Rapport 77-R-011, Centre de Recherche en Informatique de Nancy, Nancy, 1977; in *Constructing Quality Software*, P. G. HIBBARD et S. A. SCHUMAN, éd., North-Holland, 1978.