Un cadre pour la définition récursive des structures de données

Jean-Louis Giavitto[1] & Olivier Michel[2]

1: LRI, umr 8623 du CNRS,
Université de Paris-Sud, bâtiment 490
91460 Orsay cedex
`giavitto@lri.fr`

2: LaMI, ep 738 du CNRS,
Université d'Évry Val d'Essonne, Bd. F. Mitterrand
91025 Evry cedex
`michel@lami.univ-evry.fr`

Résumé

Les langages fonctionnels comme `ML` permettent de définir des fonctions par des équations récursives. Nous présentons ici les difficultés soulevées par l'extension de cette approche à la définitions de structure de données, et nous présentons un cadre général permettant de formaliser ce problème. La définition récursive de vecteurs est utilisée pour illustrer les notions introduites.

Le principal problème, si on veut éviter les définitions partielles et les stratégies d'évaluation paresseuse, est de déterminer un ordonnancement des calculs permettant de résoudre simplement l'équation récursive. Pour cela nous introduisons les notions de maximalité et de calculabilité par préfixe d'un programme et nous donnons une condition permettant de vérifier qu'un programme calculable par préfixe calcule une structure de données totale.

Dans le cas des vecteurs, nous pouvons vérifier cette condition par un critère syntaxique simple.

1. Introduction

Le support de la notion de fonction par les langages de la famille `ML` est l'un des plus puissant que l'on puisse trouver parmi les langages de programmation et représente indéniablement un succès. Notre opinion est que ce succès repose sur les quatre points caractéristiques suivants :

1. la notion de fonction en `ML` repose sur un objet mathématique bien défini et étudié (les λ-*expressions*) ;
2. la définition d'une fonction se fait par un système d'équations de la forme : *variable* $= \lambda$-*expression* ;
3. on détermine automatiquement à la compilation des propriétés sur les fonctions définies qui seront utiles pour la vérification de la correction de leurs usages et pour leur implémentation (*inférence de type*) ;
4. on peut inférer à partir d'une définition de fonction, un ordonnancement des calculs qui permettent de résoudre les équations (stratégie d'évaluation normalisante et traduction en termes de *combinateurs*).

En effet, les point 1 et 2 permettent de raisonner algébriquement sur les programmes fonctionnels (on peut par exemple remplacer un nom de fonction par sa définition). Et les points 3 et 4 permettent de relier la notion mathématique de fonction à sa représentation informatique et d'assurer une implémentation efficace.

Notre idée est d'étendre cette approche aux structures de données (en assurant les quatre points précédents). Cette démarche est un travail de longue haleine : nous ne proposons ici que quelques définitions cadres ; cependant, nous illustrons et développons complètement cette approche pour le cas particulier des vecteurs.

Dans le reste de cette section, nous illustrons les problèmes spécifiques posés par la définition récursive de structures de données, en considérant les listes infinies et les vecteurs.

1.1. Exemples de définitions récursives de structures de données

Un exemple de définition récursive de liste infinie

Dans les premières versions de `CAML` [Ler93], on peut définir des structures de données par des définition récursives. Par exemple :

$$\texttt{let rec}\ z = 1 :: z\,;;$$

qui définit la suite infinie $1 :: 1 :: 1 :: \ldots$ (en `CAML`, l'opérateur :: est le constructeur qui ajoute un élément à une liste). Cette définition n'est que l'écriture, en syntaxe `CAML`, de l'équation :

$$z = 1 :: z \tag{1}$$

où la solution z est à rechercher parmi les listes (finies et infinies) d'entiers. Cependant, une définition à peine plus compliquée échoue ; le programme :

$$\texttt{let rec}\ inc\ z = (1 + \texttt{hd}\ z) :: inc(\texttt{tl}\ z)\,;;$$
$$\texttt{let rec}\ iota = 1 :: (inc\ iota)\,;;$$

provoque le message d'erreur suivant sur la définition de $iota$: « `This kind of expression is not allowed in right-hand sides of "let rec"` ». En effet, `CAML` n'accepte pas la définition récursive de structures de données si elle implique l'application d'une fonction (ici la fonction inc s'applique à une liste infinie et retourne une liste dont les éléments correspondent aux éléments de l'argument incrémenté de un). Seul un terme impliquant uniquement des constructeurs peut apparaître en partie droite.

Ce type de définition est par contre accepté par un langage à évaluation paresseuse comme `Haskell` [H+96]. On peut par exemple en `Haskell` définir la liste suivante :

$$\texttt{let rec}\ z = 1 :: \texttt{tl}\ z\,;; \tag{2}$$

mais cette définition « boucle » : plus exactement, l'évaluation de l'expression $\texttt{hd}\,(\texttt{tl}\ z)$ qui doit retourner le deuxième élément de z, provoque un débordement de pile. Remarquons que toutes les suites qui débutent par la valeur 1 sont solutions de l'équation (2).

Les suites infinies ne sont pas les seul types de données que l'on peut définir par des équations. Par exemple, en `Haskell` tout type de données algébrique peut donner lieux à des équations récursives.

L'équation (2) montre que, sans précaution particulière, la définition récursive de structures de données permet de définir des objets *partiels* (i.e. des structures de données où certains éléments ne sont pas définis) exactement comme la définition de fonctions récursives permet de définir des fonctions partielles.

Cependant, contrairement à la stratégie adoptée par `Haskell`, nous voulons **déterminer à la compilation** les définitions de structures de données qui conduisent à des **programmes ne se**

terminant pas, ou bien à des structures de données *partielles*, afin de les rejeter. En effet, la notion habituelle de structures de données implique que chaque élément de la structure ait une valeur bien définie. La solution adoptée par `CAML`, à savoir, se restreindre aux expressions ne comportant que des constructeurs, permet de satisfaire cette contrainte mais est brutale : elle écarte trop de définitions intéressantes.

Un exemple de définition récursive de vecteur

La structure de vecteur est fondamentalement différente de la structure de liste infinie. Par exemple, il n'y a pas de définition des vecteurs sous la forme d'un type algébrique (avec un constructeur qui serait l'équivalent d'un :: pour les listes). On peut cependant les définir récursivement. Voici un exemple :

$$iota = [|\ 0\ |] \ \diamond\ ([|\ 1; 1\ |] \ +\ iota : 2) \tag{3}$$

Dans cette équation, les vecteurs littéraux se notent comme en `CAML` en listant les éléments entre [| et |]. L'opérateur $\diamond$ représente la concaténation des vecteurs, l'opérateur $+$ l'addition élément par élément, et l'expression $A : n$ correspond à la coupure d'un vecteur A aux n premiers éléments. Ce sont des opérations polymorphes qui acceptent des vecteurs de taille différentes.

On peut vérifier que le vecteur $V = [|\ 0; 1; 2\ |]$ est une solution de cette équation. En effet,

$$\begin{array}{l} V = [|\ 0; 1; 2\ |] \\ \Rightarrow V : 2 = [|\ 0; 1\ |] \\ \Rightarrow 1 + (V : 2) = [|\ 1; 2\ |] \\ \Rightarrow 0 \diamond (1 + (V : 2)) = [|\ 0; 1; 2\ |] \ = \ V \end{array}$$

Comment calculer la solution d'une équation récursive ?

Comment calculer le vecteur solution V ? L'équation (3) peut se traduire en un système d'équations qui définit les éléments de *iota*. En effet, en supposant que nous sachions que la solution est un vecteur de trois éléments, on peut écrire (la numérotation des éléments d'un vecteur commence à 1) :

$$iota = [|\ iota_1; iota_2; iota_3\ |]$$

d'où, évaluant la coupure dans le membre droit de l'équation (3) :

$$[|\ iota_1; iota_2; iota_3\ |] = [|\ 0\ |] \ \diamond\ ([|\ 1; 1\ |] + [|\ iota_1; iota_2\ |])$$

ce qui peut se réécrire en

$$\left\{ \begin{array}{lcl} iota_1 & = & 0 \\ iota_2 & = & 1 + iota_1 \\ iota_3 & = & 1 + iota_2 \end{array} \right.$$

Ce système d'équations entre les éléments du vecteur *iota* n'est pas récursif et on peut résoudre ce système d'équations par une série de substitutions. Remarquons que cette situation est d'une certaine manière similaire à ce qui se passe pour le calcul de `factorielle(`n`)`. La définition de la fonction `factorielle` (resp. du vecteur *iota*) est récursive, mais le calcul de la valeur que prend cette fonction pour un entier donné (resp. le calcul de la valeur d'un élément du vecteur *iota*) ne boucle pas.

La question qui se pose est de savoir **dans quel ordre faut-il faire ces substitutions ?** Dans la définition de *iota*, on peut vérifier que l'équation définissant l'élément d'indice i dépend, pour $i > 1$, uniquement de la valeur l'élément d'indice $(i - 1)$, et la valeur de l'élément d'indice 1 ne dépend d'aucun autre élément. On peut donc résoudre ce système uniquement par substitution des valeurs des éléments en calculant les membres droit des équations dans l'ordre des indices croissants. Le calcul de *iota* peut donc naturellement se traduire par le code (en `C`) :

```
iota[0] = 0;
for (i = 1; i<3; i++)    iota[i] = 1 + iota[i-1];
```

Notons que dans cette boucle, l'accès à la valeur d'un élément ne se fait que si cet élément possède déjà une valeur.

Là aussi notre approche s'écarte de la solution adoptée par `Haskell` : notre stratégie d'évaluation n'est pas paresseuse et nous voulons, avant le calcul de la solution disposer d'un ordonnancement de tous les sous-calculs nécessaires. De plus, nous ne voulons pas permettre n'importe quel ordonnancement : *il faut qu'il soit* « compatible » *avec la structure de données définie*. Par exemple, l'ensemble des élément ayant une valeur définie au cours du calcul de *iota* forme toujours un vecteur. Autrement dit, nous voulons que lors du calcul d'une structure de données s, tout résultat intermédiaire soit encore un s.

Ce n'est pas toujours le cas, l'équation (4) (en page 109) en donne un exemple pour les vecteurs.

Cette contrainte est forte, mais son expression est naturelle dans un cadre où on veut s'assurer que tous les éléments de la structure de données ont une valeur bien définie et où les calculs doivent pouvoir se faire suivant un ordre simple. On peut interpréter cette contrainte comme le fait que les calculs respectent un schéma de « récursion primitive » liée à la structure de données.

Subsidiairement, pour la résolution de l'équation (3), il faut déterminer le nombre d'élément du vecteur solution, ce qui correspond à un problème d'**inférence de type**[1]. L'inférence du type de la solution de l'équation (1) est un problème résolu pour les types de données algébriques. Cependant, l'exemple des vecteurs montre que les structures de données ont besoin de *types plus riche*.

1.2. Plan de l'article

Le reste de ce travail s'organise comme suit. La section suivante 2 décrit un cadre théorique général permettant de formaliser la définition récursive de structures de données et leur résolution. La section 3 présente une instance du cadre formel précédent pour le cas particulier des vecteurs. La section 4 présente rapidement un prototype mettant en œuvre les notions et résultats précédents en `Mathematica`. Nous finissons par une présentation rapide de travaux apparentés et par les perspectives ouvertes par cette recherche.

2. Formalisation

2.1. Notion de structure de données

Le concept de structure de données est central en informatique mais sa caractérisation n'est pas aisée : certaines approches (par exemple la théorie des domaines) privilégient la construction de structure de données sophistiquées à partir de structures de données plus simples ; d'autres approches (comme les spécifications algébriques) se concentrent sur la définition des opérations permises sur une structure de données.

Nous nous reposerons sur l'idée intuitive qu'une structure de données s est une « organisation » ou un « arrangement » O d'éléments pris dans un ensemble V. Il est usuel alors de considérer la paire

1. Le type vecteur en `CAML` ne comporte pas d'indication du nombre d'éléments du vecteur. Cela permet par exemple de créer des matrices triangulaires comme vecteur de vecteurs. Mais cela ne permet pas de déterminer à la compilation que la concaténation de deux matrices v_1 et v_2 est bien définie : pour cela il faut déterminer que la longueur de v_1 est supérieure à la longueur de v_2.

$s = (O, V)$ et d'énoncer que s est une structure O de V : par exemple une *liste* d'*entiers*, un *vecteurs de 5* éléments *float*, etc.

Habituellement, la paire s est considérée dans le cadre de la théorie des ensembles. Ainsi, l'ensemble $\mathcal{G}$ des graphes orientés simples (i.e. sans arc multiple) peut être défini comme $\mathcal{G} = \{ (O, V) : O \subseteq V \times V \}$ où V est l'ensemble des sommets des graphes simples. Cette approche considère également la structure O et l'ensemble V ce qui ne promeut pas l'idée qu'une structure O est un ensemble de « places » ou de « positions » occupées par des éléments de V.

C'est ce dernier point de vue que nous voulons adopter. Dans ce cadre, un type de donnée est un couple $(\mathcal{O}, V)$ et une structure de donnée s de type $(\mathcal{O}, V)$ est un triplet $(f, \mathcal{O}, V)$ où f est une fonction totale d'un $O \in \mathcal{O}$ dans V : $f \in O \mapsto V$.

L'ensemble V joue le même rôle que précédemment : c'est l'ensemble des valeurs qui peuvent être « stockées » dans la structure de données. L'ensemble $\mathcal{O}$ est l'ensemble des organisations O possibles (pour le type $\mathcal{T}$). Une structure de données s de type T est une valuation des positions d'une organisation donnée O de $\mathcal{O}$, c'est à dire qu'elle associe à chaque position o de O une valeur $f(o)$ de V.

2.2. Notion de sous-structure

L'ensemble des organisations $\mathcal{O}$ n'est pas sans structure : une structure de données possède naturellement une notion de sous-structure qu'on obtient en « oubliant » certaines positions. Voici deux exemples :

1. Un arbre possède des sous-arbres (les fils de la racine) ;
2. Un vecteur de taille n possède naturellement comme sous-vecteurs tous les vecteurs de taille $m < n$ obtenus en oubliant les $(n - m)$ derniers éléments.

Le premier exemple montre que la notion de sous-structure peut se définir à partir de la spécification récursive du type de données. Le deuxième exemple montre que cette notion peut être aisément étendue à d'autres types de données. C'est à cette notion de sous-structure que nous voulons rattacher la notion de « récursion primitives » : une définition de données est récursive primitive si le calcul de la structure peut se faire uniquement à l'aide des sous-structures.

Nous modéliserons la notion de sous-structure en munissant l'ensemble $\mathcal{O}$ d'une relation d'ordre partielle $\sqsubseteq_{\mathcal{O}}$ (on omet l'indice s'il n'y a pas d'ambigüité). Afin qu'une définition inductive sur les structures soit valide, on demandera à ce que $(\mathcal{O}, \sqsubseteq)$ soit un domaine (cf. [Mos90]) bien fondé. Le plus petit élément de $\mathcal{O}$ est noté $\perp_{\mathcal{O}}$. Si $O \sqsubseteq O'$, alors les positions de O doivent pouvoir être considérées comme des positions de O'. Par suite, si $O \sqsubseteq O'$ il existe une injection $j_{O \mapsto O'}$ de O dans O'. Pour des raisons techniques nous allons supposer que $\mathcal{O}$ possède un plus grand élément $\top_{\mathcal{O}}$. Cet ensemble possède lui-même une certaine structure : c'est un domaine (dans la pratique, il suffira par exemple qu'il soit dénombrable). D'où la définition :

DÉFINITION 1 (Type et structure de données). Un type de donnée est un couple $(\mathcal{O}, V)$ où V est un ensemble de valeurs et $\mathcal{O}$ un domaine bien fondée possédant un plus grand élément, un domaine noté $\top_{\mathcal{O}}$, et tel que pour toute paire $O \sqsubseteq_{\mathcal{O}} O'$ d'éléments de $\mathcal{O}$, il existe une injection $j_{O \mapsto O'}$. Par convention, $j(O)$ désigne $j_{O \mapsto \top}(O)$. Une structure de données s de type $(\mathcal{O}, V)$ est une fonction totale d'un $O \in \mathcal{O}$ dans V. O est appelé l'*organisation* de s. Les éléments de O sont appelés les *positions* de s. Si s et s' sont des structures de données d'organisation O et O', et si $O \sqsubseteq O'$ et si $s = s' \circ j_{O \mapsto O'}$, alors on dit que s est une sous-structure de s'.

Exemple des arbres binaires. En `CAML` le type correspondant à un arbre binaire décoré se spécifierait par : `type 'a B = Leaf of 'a | Node of 'a * ('a B) * ('a B)`. Pour notre objectif, le type correspondant est $\mathcal{B}_\alpha = (\mathcal{O}, \alpha)$ où α est le type de la décoration. Un élément $O \in \mathcal{O}$ est un sous-ensemble de $\Omega = \{1, 2\}^*$ qui vérifie la propriétés suivante : si $\omega.\omega' \in O$ alors $\omega \in O$. Une position est un mot, la lettre `1` indiquant qu'il faut sélectionner le sous-arbre gauche et la lettre `2` le sous-arbre droit. Le mot vide est noté Λ. Par exemple, l'arbre binaire décrit en CAML par le terme `Node (5, Leaf 6, Node (7, Leaf 8, Leaf 9))` correspond à l'organisation $\{\Lambda, 1, 2, 2.1, 2.2\}$ et est décrit par la fonction $\{\Lambda \mapsto 5, 1 \mapsto 6, 2 \mapsto 7, 2.1 \mapsto 8, 2.2 \mapsto 9\}$. On adjoint à $\mathcal{O}$ ainsi construit l'élément Ω qui servira de plus grand élément. La relation $\sqsubseteq_{\mathcal{O}}$ correspondant à cette notion d'arbre binaire est la relation suffixe : on la définit par $O \sqsubseteq_{\mathcal{O}} O'$ s'il existe $\omega' \in O'$ tel que $\omega'.O \subset O'$. L'injection $j_{O \mapsto O'}$ est définie par : $j_{O \mapsto O'}(\omega) = \omega'.\omega$.

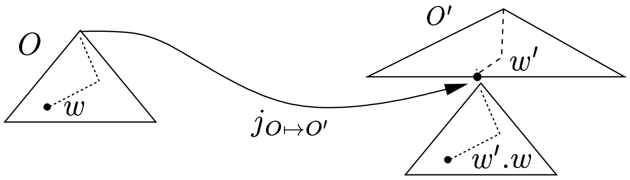

FIG. 1 – *Structure d'arbre binaire et de sous-arbre*

2.3. Notion de définition récursive

Pour simplifier la présentation, nous ne nous intéresserons dans cet article qu'aux programmes P comportant une seule définition. Les résultat présentés ici ont été étendus aux systèmes d'équations, mais cela complique considérablement certaines définitions.

Une définition récursive est une équation de la forme $X = \varphi_{\mathtt{P}}(X)$ où X est une structure de données de type $(\mathcal{O}_{\mathtt{P}}, V_{\mathtt{P}})$ (nous omettrons les indices s'il n'y a pas d'ambigüité). Cette équation peut admettre zéro, une ou plusieurs solutions.

Afin de s'assurer qu'une solution existe toujours, et afin de permettre le choix d'une solution particulière s'il y en a plusieurs, nous allons nous interesser à un problème légèrement différent : trouver la solution X dans l'ensemble $\mathcal{T}_{\mathtt{P}} = \top_{\mathcal{O}} \to V_{\perp}$ des *fonctions partielles* de $\top_{\mathcal{O}}$ dans $V_{\perp}$.

Les définitions et résultat suivants sont classiques en sémantique dénotationelle, cf. [Mos90]. Si E est un ensemble, $E_{\perp}$ désigne le domaine plat construit à partir de E. L'ensemble $\mathcal{T}_{\mathtt{P}}$ est un domaine munit de l'ordre de Scott et φ est une fonction de $\mathcal{T}$ dans lui-même. Si on impose que φ soit continue, alors l'équation P admet une plus petite solution que nous noterons $\mathtt{fix}\,\varphi$ ou bien $\mathtt{fix}\,\mathtt{P}$. De plus, on peut calculer $\mathtt{fix}\,\mathtt{P}$ car $\mathtt{fix}\,\mathtt{P} = \bigsqcup_n \varphi^n(\perp_{\mathcal{T}})$.

Cette solution peut être considérée comme une « structure de données partielle ». En effet, une fonction partielle f de $\mathcal{T}$ définit une unique fonction partielle $f_O = f \circ j_{O \mapsto \top}$ sur $O \in \mathcal{O}_{\mathtt{P}}$ (l'injection existe puisqu'on a $O \sqsubseteq \top_{\mathcal{O}_{\mathtt{P}}}$). Dans la suite on notera f à la place de f_O quand le domaine de définition peut être inféré par le contexte.

2.4. Notion de maximalité

Notre problème à présent est de déterminer si $\mathtt{fix}\,\mathtt{P}$ est bien une structure de données, c'est à dire si c'est une fonction *totale* d'un $O \in \mathcal{O}_{\mathtt{P}}$.

Si f est une fonction d'un domaine D dans un domaine E alors $\mathtt{Def}(f)$ dénote le domaine de définition de f, i.e : $\mathtt{Def}(f) = \{ d \in D \ : d \neq \perp_D \ \wedge \ f(d) \neq \perp_E \}$. Traditionellement, on dit que f est une fonction totale si $\mathtt{Def}(f) = D - \{ \perp_D \}$.

DÉFINITION 2 (Programme maximal). Soit $\mathtt{P} : X = \varphi_{\mathtt{P}}(X)$ une définition récursive sur un type de donnée $(\mathcal{O}, V)$. $\mathcal{T}_{\mathtt{P}}$ désigne $\top_{\mathcal{O}} \to V_{\perp}$. La fonction $\varphi_{\mathtt{P}}$ est une fonction continue de $\mathcal{T}_{\mathtt{P}}$ dans lui-même.

On dit que P est maximal s'il existe un $O \in \mathcal{O}$, tel que $j(O) = \mathtt{Def}(\mathtt{fix}\,\varphi_{\mathtt{P}})$. Si P est maximal, on dit que P définit la structure de données $s = (\mathtt{fix}\,\varphi_{\mathtt{P}}) \circ j_{O \mapsto \top}$.

En effet si P est maximal, alors en posant $f = \mathtt{fix}\,\mathtt{P}$ et O tel que $j(O) = \mathtt{Def}(\mathtt{fix}\,\mathtt{P})$, on peut voir que f_O est un élément maximal parmi les fonctions *partielles* de $O \to V_{\perp}$ pour l'ordre de Scott.

2.5. Inférence de l'organisation de la solution d'une définition récursive

Pour prouver que $\mathtt{Def}(\mathtt{fix}\,\mathtt{P})$ est un élément de $\mathcal{O}$, nous allons essayer d'inférer un O à partir du programme P et de montrer que $\mathtt{fix}\,\mathtt{P}$ est maximal sur ce O.

Un élément O de $\mathcal{O}$ ne correspond généralement pas à un type au sens usuel mais à une notion plus précise. C'est l'ensemble $\mathcal{O}$ qui correspond à un type CAML. Nous allons cependant traiter les O comme des types et nous voulons inférer pour chaque programme P une organisation $O_{\mathtt{P}}$ tel que : $\mathtt{fix}\,\mathtt{P}|_{j(O_{\mathtt{P}})} = \mathtt{fix}\,\mathtt{P}$ (la notation $f|_{E'}$ désigne la restriction de la fonction f sur E à $E' \subseteq E$).

Pour cela, nous supposerons qu'à chaque programme P on peut associer un ensemble d'équations $\widetilde{\mathtt{P}}$ dont la solution est à chercher dans $\mathcal{O}$. Nous dirons que P *est typable* si $\mathtt{fix}\,\widetilde{\mathtt{P}}$ existe et que l'inférence des types est *correcte* si pour tout programme typable P, on a $\mathtt{Def}(\mathtt{fix}\,\mathtt{P}) \subseteq j(\mathtt{fix}\,\widetilde{\mathtt{P}})$. Dans la suite, nous supposons que l'inférence des types est correcte.

Notons que si on dispose d'une inférence de type correcte, et que $O = \mathtt{fix}\,\widetilde{\mathtt{P}}$ est fini, alors le problème de déterminer si un programme $\mathtt{P} : X = \varphi(X)$ est maximal est théoriquement résolu : il suffit de résoudre l'équation sur $j(\mathtt{fix}\,\widetilde{\mathtt{P}})$ plutôt que sur $\top_{\mathcal{O}}$. Autrement dit, la solution de $X = \varphi(X)$ est une fonction récursive s dont on veut calculer les images pour tous les $o \in O$. Ce calcul peut se faire en s'assurant qu'on ne boucle pas dans $\mathtt{fix}\,\widetilde{\mathtt{P}}$ (qui est fini). Si on detecte un bouclage lors du calcul de $s(o)$, alors $s(o) = \perp$. On peut donc calculer $\mathtt{Def}(s)$ et déterminer ainsi si P est maximal.

Nous voulons écarter cette approche : d'abord le domaine $\mathtt{fix}\,\widetilde{\mathtt{P}}$ peut être très grand ; ensuite cette méthode revient essentiellement à calculer la structure de données et donc ne peut pas s'effectuer lors de la compilation ; enfin, cette méthode ne permet pas de déterminer si on peut calculer par un ordonnancement simple la structure de données : on considère qu'une structure de données est « juste » une fonction sur un domaine fini.

2.6. Calculabilité par préfixe

Arrivé à ce point, nous voulons :

1. déterminer si $\mathtt{fix}\,\mathtt{P}$ est maximal sur $\mathtt{fix}\,\widetilde{\mathtt{P}}$;
2. déterminer un ordonnancement simple des calculs.

Il nous faut préciser cette notion d'ordonnancement « simple ».

Nous savons calculer la solution d'un programme P en calculant $\varphi_{\mathtt{P}}^n(\perp)$. Comme le domaine d'arrivée des fonctions que nous considérons pour le calcul de la solution est un domaine plat (c'est $V_{\perp}$), si la position o a une valeur définie pour $\varphi_{\mathtt{P}}^n(\perp)$, alors il en sera de même pour tout $\varphi_{\mathtt{P}}^m(\perp)$ avec $m \geq n$. On n'a donc pas besoin de recalculer la valeur $\varphi_{\mathtt{P}}^m(\perp)(o)$ une fois qu'on l'a obtenue. On peut donc réaliser le calcul de $\varphi_{\mathtt{P}}^t(\perp)$ en calculant à l'étape t uniquement les valeurs associées aux positions de $\delta_t = \mathtt{Def}(\varphi_{\mathtt{P}}^t(\perp)) - \mathtt{Def}(\varphi_{\mathtt{P}}^{t-1}(\perp))$.

Le problème est de caractériser simplement les ensembles δ_t ce qui permettrait d'ordonnancer aisément l'évaluation de la valeur des positions o. Cette approche est en général trop difficile : la « forme » du domaine de définition de $\varphi_{\mathtt{P}}^t(\perp)$ peut être complexe.

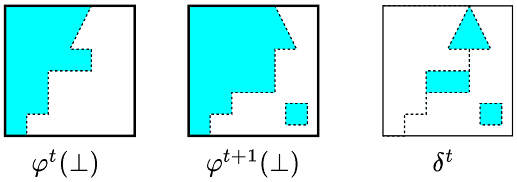

FIG. 2 – *Croissance du domaine de définition entre deux itérations du calcul de* $\mathtt{fix}\,\varphi$.

Notre idée est d'essayer de caractériser une classe de programmes pour lesquels on peut « oublier » des positions du domaine de définition de $\varphi_{\mathtt{P}}^t(\perp)$, sans que cela nuise au résultat final. En oubliant ainsi certaines parties du domaine de définition à chaque itération, on impose une forme simple à δ^t, compatible avec la notion de sous-structure de données.

Pour exprimer cette notion de forme simple compatible avec la notion de sous-structure, il faut introduire la notion de préfixe.

DÉFINITION 3 (Préfixe d'un ensemble et d'une fonction). Soit E un sous-ensemble de $\top_{\mathcal{O}}$. Les préfixes de E sont les éléments $O \in \mathcal{O}$ tels que $j(O) \subseteq E$. Nous voulons choisir un préfixe particulier. Nous munissons donc $\top_{\mathcal{O}}$ d'une relation d'ordre *total* $\preccurlyeq$ qui est *compatible* avec $\sqsubseteq_{\mathcal{O}}$ à savoir : $O \sqsubseteq O' \Rightarrow j(O) \preccurlyeq j(O')$. On définit $\lfloor E \rfloor = \max_{\preccurlyeq}\{j(O) : O \in \mathcal{O} \wedge j(O) \subseteq E\}$.

Si $f \in \mathcal{T}$, on note $\lfloor f \rfloor = \lfloor \mathtt{Def}(f) \rfloor$ et $\Box f = f|_{\lfloor f \rfloor}$.

Nous pouvons à présent formaliser la notion d'ordonnancement simple par la notion de calculabilité par préfixe.

DÉFINITION 4 (Calculabilité par préfixe). Un programme $\mathtt{P} : X = \varphi(X)$ est calculable par préfixe si et seulement si $\mathtt{fix}\,\varphi = \mathtt{fix}(\Box \circ \varphi)$.

Cette définition se justifie intuitivement en remarquant qu'un programme calculable par préfixe, peut calculer la structure de données solution en n'utilisant que des sous-structures de données correspondant aux préfixes des itérées, et sans faire appel à une structure plus générale.

L'exemple (4) du paragraphe 3.4 montre que la propriété de maximalité de la solution ne coïncide pas avec la propriété de calcul par préfixe : il y a des programme maximaux qui ne sont pas calculables par préfixe.

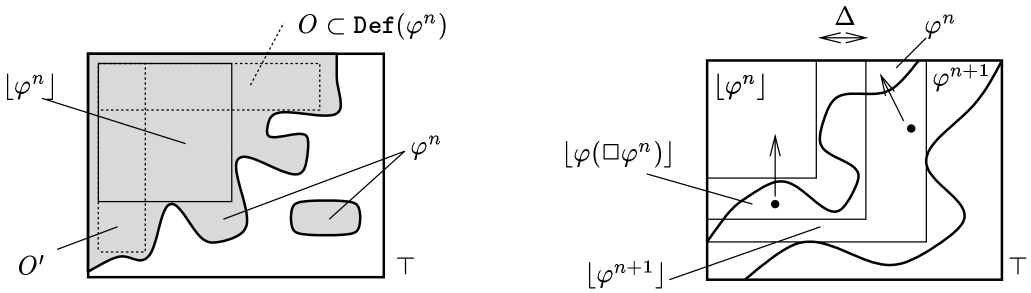

FIG. 3 – *Préfixe des itérations de* φ. *Dans la schéma,* φ^n *abrège* $\varphi^n(\perp)$.

2.7. Croissance du domaine de définition et calculabilité par préfixe

Nous voulons à présent relier la propriété d'être calculable par préfixe et la croissance du domaine de définition entre deux itérées. Cette notion de croissance peut se formuler ainsi.

DÉFINITION 5 (Programme Δ-croissant). Un programme $\mathtt{P} : X = \varphi(X)$ est Δ-croissant, avec Δ une fonction de $\mathcal{O}$ dans $\mathcal{O}$, *continue* pour la relation $\sqsubseteq$, si et seulement si : $\forall U \in \mathcal{T},\ j(\Delta(\lfloor U \rfloor)) \sqsubseteq \lfloor \varphi(U) \rfloor$.

$\mathcal{O}$ est un domaine et Δ est continue. Par suite $\mathtt{fix}\,\Delta$ existe. Un résultat important qui justifie nos définitions est que :

THÉORÈME 1 (Maximalité, calcul par préfixe et Δ-croissance). Un programme typable $\mathtt{P} : X = \varphi(X)$ est maximal et calculable par préfixe s'il existe une fonction $\dot{\varphi}$ telle que $\mathtt{P}$ est $\dot{\varphi}$-croissant et que $\mathtt{fix}\,\widetilde{\mathtt{P}} \sqsubseteq \mathtt{fix}\,\dot{\varphi}$. Dans ce cas, nous dirons que $\mathtt{P}$ est *$\dot{\varphi}$-progressif.*

3. Définition récursive de vecteurs

La section précédente vient d'esquisser un cadre général pour la définition récursive des structures de données. Pour un programme donné $\mathtt{P}$, il faut à présent exhiber une fonction $\dot{\varphi}_{\mathtt{P}}$ tel que $\mathtt{P}$ soit $\dot{\varphi}_{\mathtt{P}}$-progressif.

Dans cette section, nous développons la démarche proposée au cas des vecteurs. Dans la section précédente, nous avons travaillé uniquement au niveau sémantique. Nous allons à présent nous intéresser aussi aux expressions. L'idée est de passer automatiquement de l'expression du programme $\mathtt{P}$ à l'expression d'une fonction $\dot{\varphi}$. La fonction $\dot{\varphi}$ obtenue est telle que $\mathtt{P}$ est $\dot{\varphi}$-croissant. Mais cela ne suffit pas pour qu'il soit $\dot{\varphi}$-progressif.

Nous proposons alors un critère, sous la forme d'une fonction $\mathtt{prog}$ s'appliquant à l'expression de $\dot{\varphi}$, qui donne une condition suffisante pour qu'un programme $\mathtt{P}$ soit $\dot{\varphi}$-progressif.

3.1. Domaines sémantiques

Le type $(\mathcal{V}, V)$ représente le type de nos vecteurs d'éléments de V. L'ensemble $\mathcal{V}$ est défini par :

$$\mathcal{V} = \{\ \{1, \ldots, n\} : n \in \mathbb{N}^+\ \} \cup \{\ \mathbb{N}^+\ \}$$

où $\mathbb{N}^+ = \mathbb{N} - \{0\}$. L'ensemble $\mathcal{V}$ est ordonné par inclusion : $\sqsubseteq_{\mathcal{V}}$ coïncide avec $\subseteq$. Cette relation est une relation d'ordre totale sur $\mathcal{V}$ et donc on a $\preccurlyeq\, = \,\subseteq$ puisque $\preccurlyeq$ doit être compatible avec $\sqsubseteq$. Les fonctions $j_{O \mapsto O'}$ correspondent simplement à l'identité. L'élément maximal $\top_{\mathcal{V}}$ est l'ensemble $\mathbb{N}^+$ qui est construit comme l'union des $O \in \mathcal{O}, O \neq \mathbb{N}^+$.

Intuitivement, un vecteur est une fonction totale de $\{1, \ldots, n\}$ dans V. Un vecteur s est un sous-vecteur de s' si $s = s'|_{\{1,\ldots,n\}}$, autrement dit, s est un sous-vecteur de s' si on peut obtenir s en tronquant s'. Le domaine $\mathcal{T}_{\mathcal{V}}$ (des fonctions partielles de $\top_{\mathcal{V}}$ dans $V_\perp$) est l'ensemble des vecteurs partiels à valeur dans V.

3.2. Expressions et programmes sur les vecteurs

Une expression de vecteurs est définie par la grammaire suivante :

$$EXP \ ::= \ [\!|\, v_1; \ldots; v_p \,|\!] \ \big| \ f(e_1, \ldots, e_p) \ \big| \ e_{+p} \ \big| \ e : p \ \big| \ e_1 \diamond_p e_2 \ \big| \ id$$

où $p \in \mathbb{N}^+$, $id \in VAR$ un ensemble de variables, les $v_i \in V$, $e, e_i \in EXP$ et où $f \in F$ un ensemble de fonctions *strictes* de V^p dans V. Intuitivement, la construction 1 permet de construire extensionellement des vecteurs, la construction 2 correspond à l'extension d'une fonction scalaire aux vecteurs, la construction 3 correspond au `eoshift` de `Fortran`, la construction 4 est la coupure et la construction 5 la concaténation des vecteurs[2].

Les symboles fonctionnels des termes de EXP sont interprétés de la manière suivante dans $\mathcal{T}_\mathcal{V}$:

1. $[\!|\ v_1; \ldots ; v_p\ |\!] = \lambda o.$suivant (o) cas 1: v_1 ; ... ; cas p: v_p ; et sinon $\perp$
2. $f(e_1, \ldots, e_p) = \lambda o.f(e_1(o), \ldots e_p(o))$
3. $e_{+p} = \lambda o.e(o + p)$
4. $e : p = \lambda o.$si $o \leq p$ alors $e(o)$ sinon $\perp$
5. $e_1 \diamond_p e_2 = \lambda o.$si $o \leq p$ alors $e_1(o)$ sinon $e_2(o - p)$

Un programme P sur les vecteurs est un couple $x = e$ où x est une variable et où e est une expression où seule la variable x apparaît. Ce couple est interprété comme une équation $X = \varphi_\mathtt{P}(X)$ à résoudre dans $\mathcal{T}_\mathcal{V}$ en posant $\varphi_\mathtt{P} = \lambda x.e$.

LEMME 1. Si P est un programme sur les vecteurs, alors $\varphi_\mathtt{P}$ est une fonction continue. La fonction $\varphi_\mathtt{P}$ étant continue, $\mathtt{fix}\,\varphi_\mathtt{P}$ existe et se calcule comme $\bigsqcup_n \varphi_\mathtt{P}^n(\perp)$.

3.3. Inférence de type pour les vecteurs

L'ensemble $\{1, \ldots, n\}$ peut se coder par le nombre n et l'ensemble vide $\emptyset$ peut se coder par 0. On peut donc en particulier représenter le domaine de définition d'un vecteur par un entier qui correspond au nombre d'éléments du vecteur et que nous appelerons sa *longueur*. Cette représentation plus concise permet de simplifier la présentation de l'ensemble d'équations $\widetilde{\mathtt{P}}$ associé à un programme P.

Pour construire $\widetilde{\mathtt{P}}$, l'idée est la suivante. À chaque variable x de P on associe une variable n^x et à chaque sous-expression e d'une partie droite d'une équation de P, on associe une variable n^e. Ces variables sont des variables de type et la valeur de ces variables sera le type de la sous-expression associée dans le programme P. Informellement, la valeur de n^x dans la solution $\mathtt{fix}\,\widetilde{\mathtt{P}}$ sera la longueur de la valeur de x dans $\mathtt{fix}\,\mathtt{P}$.

DÉFINITION 6 ($\widetilde{\mathtt{P}} = \mathbf{L}(\mathtt{P})$). L'ensemble d'équations $\widetilde{\mathtt{P}} = \mathbf{L}(\mathtt{P})$ qui permet de calculer les valeurs n^x est construit en parcourant P et en ajoutant des équations pour chaque sous-expressions e de P. Une équation de $\mathbf{L}(\mathtt{P})$ est de la forme $n^e \doteq \mathbf{L}(e)$. La fonction $\mathbf{L}$ est définie inductivement sur la structure de P :

$$\begin{array}{rcl}
\mathbf{L}(\mathtt{P}) & = & \{\, \mathbf{L}(x = e)\ :\ (x = e) \in \mathtt{P} \,\} \\
\mathbf{L}(x = e) & = & \{\, n^x \doteq n^e \,\} \cup \mathbf{L}(e) \\
\mathbf{L}(x) & = & \emptyset \\
\mathbf{L}([\!|\ v_1; \ldots ; v_p\ |\!]) & = & \{\, n^{[\!|\ v_1; \ldots ; v_p\ |\!]} \doteq p \,\} \\
\mathbf{L}(f(e_1, \ldots, e_p)) & = & \{\, n^{f(e_1,\ldots,e_p)} \doteq n^{e_i}\ :\ 1 \leq i \leq p \,\} \cup \bigcup_{1 \leq i \leq p} \mathbf{L}(e_i) \\
\mathbf{L}(e_{+p}) & = & \{\, n^{e_{+p}} \doteq n^e - p \,\} \cup \mathbf{L}(e) \\
\mathbf{L}(e : l) & = & \{\, n^{e:l} \doteq l \,\} \cup \mathbf{L}(e) \\
\mathbf{L}(e_1 \diamond_p e_2) & = & \{\, n^{e_1 \diamond_p e_2} \doteq p + n^{e_2} \,\} \cup \mathbf{L}(e_1) \cup \mathbf{L}(e_2)
\end{array}$$

2. Cette définition peut sembler à première vue peu commode, puisqu'il faut expliciter le paramètre p. Mais la fonction @ définie par $f@g = \lambda X.$si $X \leq \max \mathtt{Def}(f)$ alors $f(X)$ sinon $g(X - \max \mathtt{Def}(f))$ (qui consiste à « rajouter » les élément de g « derrière » le dernier élément de f) n'est pas une fonction continue. En effet, si on suppose que $x = \perp$ et que $y = [\!|\ 1\ |\!]$, alors, pour $x' = [\!|\ 666\ |\!]$, on a $x \sqsubseteq x'$ mais $x@y = [\!|\ 1\ |\!] \not\sqsubseteq [\!|\ 666, 1\ |\!] = x'@y$ ce qui montre que @ n'est pas monotone et donc pas continue. Cependant, on peut poser par convention $u \diamond v = u \diamond_{\mathbf{n}^\mathbf{u}} v$ où $\mathbf{n}^\mathbf{u}$ est la longueur du vecteur u inférée par le système présenté à la section 3.3.

Remarquons que :

- $\mathbf{L}(\mathtt{P})$ est un ensemble d'équations, pas un système (i.e. plusieurs équations peuvent contraindre la même variable). Par exemple l'équation $x = x + [\mid 1 \mid]$ va générer deux équations différentes portant sur la même variable $n^{x+[\mid 1 \mid]}$:

$$\mathbf{L}(x = x + [\mid 1 \mid]) = \{ n^x \doteq n^{x+[\mid 1 \mid]},\ n^{x+[\mid 1 \mid]} \doteq n^x,\ n^{x+[\mid 1 \mid]} \doteq n^{[\mid 1 \mid]},\ n^{[\mid 1 \mid]} \doteq 1 \}$$

- Les équations qui apparaissent dans $\mathbf{L}(\mathtt{P})$ sont toutes des équations linéaires (et les coefficients de la matrice de l'application linéaire sont tous des 0 ou des 1). Il existe des méthodes très efficaces pour déterminer si cet ensemble d'équations admet zéro, une ou plusieurs solutions et pour le résoudre. Voir par exemple [GSM92, Gia92].

Définition 7 (Programme typable). Nous dirons qu'un programme $\mathtt{P}$ est typable si $\mathbf{L}(\mathtt{P})$ admet une unique solution qui vérifie :

- pour toute sous-expression e de $\mathtt{P}$, les nombres n^e existent et sont des quantités strictement positives ;
- pour toute sous-expression $e : p$ de $\mathtt{P}$, on a $n^e \geq p$;
- pour toute sous-expression $e_1 \diamond_p e_2$ de $\mathtt{P}$, on a $n^{e_1} \geq p$.

Si $\mathtt{P} : x = e'$ est typable, $\mathbf{n^e}$ désigne la valeur de la variable n^e dans l'unique solution de $\mathbf{L}(\mathtt{P})$ (l'expression e est une sous-expression de e'). De plus, on note $\mathtt{fix}\,\widetilde{\mathtt{P}}$ l'ensemble $\{1, \ldots, \mathbf{n^x}\}$.

Nous avons le résultat suivant qui justifie la définition de la fonction $\mathbf{L}$:

Lemme 2 (Correction du typage géométrique). Si $\mathtt{P} : x = e$ est un programme typable, alors $\mathtt{Def}(\mathtt{fix}\,\varphi) \subseteq \mathtt{fix}\,\widetilde{\mathtt{P}}$.

3.4. Exemples

Quelques exemples de typages géométriques

1. Le programme $\mathtt{P} : x = x_{+3}$ n'est pas typable car le système $\widetilde{\mathtt{P}}$ n'admet pas de solution. En effet, on a : $\widetilde{\mathtt{P}} = \mathbf{L}(\mathtt{P}) = \{n^x \doteq n^{x_{+3}},\ n^{x_{+3}} \doteq n^x - 3\}$.
2. Le programme $\mathtt{P} : x = [\mid 1; 2 \mid] \diamond_2 x_{+2}$ n'est pas typable car le système $\widetilde{\mathtt{P}}$ admet une infinité de solutions. En effet, $\widetilde{\mathtt{P}} = \mathbf{L}(\mathtt{P}) = \{n^x \doteq n^{[\mid 1;2 \mid] \diamond_2 x_{+2}},\ n^{[\mid 1;2 \mid] \diamond_2 x_{+2}} = 2 + n^{x_{+2}},\ n^{[\mid 1;2 \mid]} \doteq 2,\ n^{x_{+2}} \doteq n^x - 2\}$ et cet ensemble d'équations est équivalent (après une série de substitution) à : $\{n^x = n^x,\ n^{[\mid 1;2 \mid] \diamond_2 x_{+2}} = n^x,\ n^{[\mid 1;2 \mid]} \doteq 2,\ n^{x_{+2}} \doteq n^x - 2\}$ qui admet une infinité de solution.
3. Le programme $\mathtt{P} : x = x : 4$ est typable car le système $\widetilde{\mathtt{P}}$ admet une solution unique (mais ce programme ne définit pas une structure de données car il n'est pas maximal).

Un programme typable et maximal, mais qui n'est pas calculable par réfixe

Voici un exemple de programme typable, maximal, et qui n'est pas calculable par préfixe :

$$\mathtt{Q} : \quad x = \left([\mid 1 \mid] \diamond_1 x_{+3} \diamond_2 [\mid 2 \mid] \diamond_3 x\right) : 4 \tag{4}$$

Ce programme est typable et on a $\mathbf{n^x} = 4$. La solution $\mathtt{fix}\,\mathtt{Q}$ peut se calculer par itération de $\varphi_{\mathtt{Q}}$:

$$\begin{array}{lcl lcl}
\varphi^1(\bot) & = & [\mid 1; \bot; 2; \bot \mid] & (\Box \circ \varphi)^1(\bot) & = & [\mid 1; \bot; \bot; \bot \mid] \\
\varphi^2(\bot) & = & [\mid 1; \bot; 2; 1 \mid] & (\Box \circ \varphi)^2(\bot) & = & [\mid 1; \bot; \bot; \bot \mid] \\
\varphi^3(\bot) & = & [\mid 1; 1; 2; 1 \mid] & (\Box \circ \varphi)^3(\bot) & = & [\mid 1; \bot; \bot; \bot \mid]
\end{array}$$

$\mathtt{Q}$ n'est pas calculable par préfixe, car $\mathtt{fix}\,\varphi \neq \mathtt{fix}(\Box \circ \varphi)$. Par contre, $\mathtt{Q}$ est un programme maximal puisque $\mathtt{Def}(\mathtt{fix}\,\varphi) = \mathtt{fix}\,\widetilde{\mathtt{P}} = \{1, \ldots, 4\} \in \mathcal{V}$.

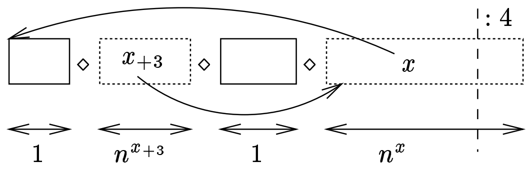

FIG. 4 – *Construction de l'exemple (4) qui est typable, maximal mais non calculable par préfixe.*

3.5. Fonction de croissance associées à un programme

Nous allons inférer une fonction $\dot\varphi_{\mathtt{P}}$ particulière pour chaque programme P, qui sera appelée *la progression de* P.

DÉFINITION 8 (Fonction $\dot\varphi_{\mathtt{P}}$ associée à un programme P). Au programme typable $\mathtt{P} : x = e$ on associe l'équation $\dot{\mathtt{P}} : \dot x \doteq \dot e$ où $\dot x$ prend sa valeur dans $\mathbb{N}$ et $\dot e = \mathbf{Pr}(e)$. On note $\dot\varphi_{\mathtt{P}}$ la fonction de $\mathbb{N}$ dans $\mathbb{N}$ définie par $\dot\varphi_{\mathtt{P}} = \lambda \dot x.\dot e$.

La fonction **Pr** est définie par cas :

$$
\begin{aligned}
\mathbf{Pr}(\mathtt{P}) &= \left\{ \dot x \doteq \mathbf{Pr}(e) \ : \ (x = e) \in \mathtt{P} \right\} \\
\mathbf{Pr}(x) &= \dot x \\
\mathbf{Pr}([\!|\, v_1; \ldots ; v_p \,|\!]) &= p \\
\mathbf{Pr}(e_{+p}) &= \max(0, \mathbf{Pr}(e) - p) \\
\mathbf{Pr}(e : p) &= \min(p, \mathbf{Pr}(e)) \\
\mathbf{Pr}(f(e_1, \ldots, e_p)) &= \min\left\{ \mathbf{Pr}(e_i) : 1 \le i \le p \right\} \\
\mathbf{Pr}(e_1 \diamond_p e_2) &= \mathbf{Pr}(e_1) \underset{p}{\odot} \mathbf{Pr}(e_2)
\end{aligned}
$$

où la fonction $\odot_p$ est définie par : $q \odot_p q' =$ si $q < p$ alors q sinon $p + q'$.

LEMME 3. On a les trois résultats suivants :

1. Si P un programme typable, la fonction $\dot\varphi_{\mathtt{P}}$ est croissante.
2. Un programme typable P est $\dot\varphi_{\mathtt{P}}$-croissant.
3. Soit P un programme typable, la fonction $\dot\varphi_{\mathtt{P}}$ admet $\mathtt{fix}\,\widetilde{\mathtt{P}}$ comme point-fixe.

Exemple. La fonction $\dot\varphi_4$ associée au programme (4) est définie par l'équation :

$$
\begin{aligned}
&\dot x \doteq \dot\varphi_4(\dot x) = \min(4,\ (\text{si } A < 3 \text{ alors } A \text{ sinon } 3 + \dot x)) \\
&\text{avec } A = \text{si } B < 2 \text{ alors } B \text{ sinon } 3 \\
&\text{et } B = 1 + \max(0, \dot x - 3)
\end{aligned}
$$

Les itérées de $\dot\varphi_4$ sont : $\dot\varphi_4(0) = 1$ et $\dot\varphi_4^2(0) = \dot\varphi_4(1) = 1$. Remarquons que $\mathtt{fix}\,\dot\varphi_4 = 1 < \mathtt{fix}\,\widetilde{\mathtt{P}}$ mais $\dot\varphi_4(4) = 4$. On a donc bien $\mathtt{fix}\,\widetilde{\mathtt{P}}$ solution de $\dot{\mathtt{P}}$ (mais ce n'est pas la plus petite).

3.6. Programme progressif

Pour déterminer si un programme est maximal et calculable par préfixe, le théorème 1 nous dit qu'il suffit de vérifier que $\mathtt{fix}\,\dot\varphi_{\mathtt{P}} \ge \mathtt{fix}\,\widetilde{\mathtt{P}}$.

Cette propriété peut se montrer simplement en calculant les itérées $\dot\varphi_{\mathtt{P}}^n(0)$. Si on tombe sur un point fixe $q \le \mathtt{fix}\,\widetilde{\mathtt{P}}$ on sait que le programme P n'est pas progressif. Sinon, on rencontre un q tel que $q \ge \mathtt{fix}\,\widetilde{\mathtt{P}}$ et on sait alors que le programme est maximal et calculable par préfixe.

Cependant, le calcul des itérées peut prendre beaucoup de temps comme le montre l'exemple (où m est un paramètre entier arbitraire) :

$$x = ([\mid 1 \mid] \diamond_1 x) : m$$

dont la fonction $\dot\varphi = \lambda n. \min(m, n+1)$ prend m itérations pour converger.

Nous voulons déterminer directement si $\texttt{fix}\,\dot\varphi_{\mathtt{P}} \geq \texttt{fix}\,\widetilde{\mathtt{P}}$ sans calculer les itérées. Cela n'est pas évident à montrer dans le cas d'un système, car si P est un programme $\dot\varphi$-croissant, on n'a pas nécessairement $\dot\varphi(q) > q$ comme le montre l'exemple suivant :

$$\begin{cases} x = \varphi_1(x, y) = ([\mid 0 \mid] \diamond_1 x) : \ 2 * m \\ y = \varphi_2(x, y) = y_{+m} \end{cases} \tag{5}$$

On peut vérifier que $\varphi_2(\perp, \perp) = \perp$ (pour tout m) et donc ce programme n'admet pas une fonction $\dot\varphi$ dont les composantes sont strictement croissantes. Notons que ce programme est calculable par préfixe, mais ce n'est qu'à partir de l'itération m (qui peut être arbitrairement grand) que y prend une valeur différente de $\perp$.

Cependant, la vérification que la fonction $\dot\varphi$ est croissante sur $\texttt{fix}\,\widetilde{\mathtt{P}}$ est relativement simple dans le cas d'un programme réduit à une équation.

DÉFINITION 9 (Programme progressif). Nous dirons qu'un programme P est progressif s'il est typable et si le système $\dot{\mathtt{P}}$ est progressif.

Un système $\dot{\mathtt{P}} : \dot x = \dot\varphi(\dot x)$ est progressif si et seulement si : $\texttt{prog}(\emptyset, 0, \dot x) > 0$. La fonction `prog` est définie par cas sur les équations et les expressions $\dot e$:

$$\begin{aligned}
&\texttt{prog}(S, q, \dot x) = \text{si } (\dot x \in S) \text{ alors } q \text{ sinon } \texttt{prog}(S \cup \{\dot x\}, q, \dot e) \text{ où } \dot e \text{ tel que } (\dot x \doteq \dot e) \in \dot{\mathtt{P}} \\
&\texttt{prog}(S, q, p) = p \\
&\texttt{prog}(S, q, \max(0, \dot e - p)) = \texttt{prog}(S, q - p, \dot e) \\
&\texttt{prog}(S, q, \min(\dot e_1, \ldots, \dot e_p)) = \min\{\, \texttt{prog}(S, q, \dot e_1), \ldots, \texttt{prog}(S, q, \dot e_p) \,\} \\
&\texttt{prog}(S, q, \dot e_1 \underset{p}{\odot} \dot e_2) = \min(\texttt{prog}(S, q, \dot e_1),\ \texttt{prog}(S,\ \max(p, p+q), \dot e_2))
\end{aligned}$$

Voici le théorème qui justifie ces définitions :

THÉORÈME 2. Un programme progressif est maximal et calculable par préfixe.

4. Réalisations

Les notions et résultats précédents ont été généralisés afin de s'appliquer à des systèmes d'équations et mis en œuvre dans un « notebook » `Mathematica`[3]. Il est possible de

- définir un système d'équations,
- inférer le type de chaque sous-expression du système,
- de calculer la solution par point-fixe,
- de calculer les itérées de la fonction préfixe $\Box\varphi$ associée au système,
- de construire la fonction $\dot\varphi$ et de calculer son plus petit point-fixe,
- de vérifier en calculant `prog` qu'un système est progressif,
- de générer un code impératif qui calcule les vecteurs solutions du système par une série de boucles d'itérations.

3. Ce notebook, ainsi que les démonstrations des résultats présentés dans cet article, sont disponibles à la page `http://www.lri.fr/~giavitto/DefRec.html` .

Le dernier point correspond à notre objectif finale de « compilation » de la définition récursive. Intuitivement, si un programme `P` d'une équation est progressif, alors il peut se calculer par une seule boucle d'itérations qui balaye les éléments du vecteurs solution de taille $\texttt{fix}\,\widetilde{\texttt{P}}$ dans l'ordre des indices croissants.

5. Conclusions et perspectives

5.1. Intégration dans le cadre des langages fonctionnels

Nous avons présenté un cadre formel permettant de définir des structures de données par des équations. Notre cadre se distingue du traitement habituel des structures de données dans le cadre fonctionnel par les points suivants : (1) notre notion de structure de données n'est pas restreinte aux types algébriques ; (2) la notions de structure de données que nous développons formalise naturellement la notion de sous-structure de données ; (3) contrairement aux fonctions, une structure de données doit être un objet « complet » et « simplement » calculable. (Cette dernière condition est importante : elle permet d'assurer par exemple des coût d'accès bien définis à une structure de données). Nous avons exprimé les conditions permettant de s'assurer que le calcul de la solution d'une équation pouvait se faire en respectant la notion de sous-structure.

Si nous nous restreignons aux types de données algébriques (nous avons illustré une définition possible pour les arbres binaires au paragraphe 2.2, mais tout type algébrique est susceptible d'un traitement similaire), la stratégie d'évaluation obtenue est à mi-chemin entre les restrictions imposées par `CAML` et la liberté accordée par `Haskell`. Le gain par rapport à la stratégie d'évaluation paresseuse d'`Haskell` est la détermination statique (donc efficace) des ressources et de l'ordonnancement des calculs. Les techniques que nous avons développer peuvent donc s'intégrer comme une optimisation de l'implantation d'un langage fonctionnel.

Nous avons illustré les concepts présentés sur le cas des vecteurs, qui n'est pas un type algébrique, pour une notion de sous-vecteur correspondant à la troncature aux n premiers éléments. Il est à noter que la définition récursive d'un tableau n'est pas possible à proprement parler en `Haskell` et passe par le détour suivant : la fonction `array` qui permet de créer un tableau prend en argument une liste d'associations qui décrit la valeur associée à un index, et cette liste peut se définir récursivement dans une construction en compréhension. Toute construction de tableau passe donc par la construction préliminaire d'une liste. Au passage nous avons esquissé un système permettant d'inférer le type d'un vecteur comprenant le nombre d'éléments du vecteur (ce qui n'existe pas ni en `CAML`, ni en `Haskell`) et fournit un critère permettant de vérifier que la définition satisfait bien aux conditions énoncées.

5.2. Travaux apparentés

Considérer une structure de données indépendemment des valeurs qui la composent est une idée centrale dans la théorie des *espèces de structures* [BL97]. Ce formalisme repose sur la théorie des catégorie pour capturer l'idée de structures isomorphes quand on ignore la nature de ses éléments. Mais l'objectif des outils développés est de permettre l'énumération de structures combinatoires.

L'idée de séparer contenu et contenant d'une structure de données est aussi une idée mise en pratique dans [Jay95]. B. Jay développe le concept de « polymorphisme de forme » (*shape polymorphism*) afin supporter le typage des fonctions qui ne modifient que l'organisation des données (l'organisation du résultat de l'application d'une telle fonction ne dépend que de l'organisation de l'argument). La fonction `map` ou `fold` sont des exemples de telles fonctions.

Les *champs de données*, étudiés par exemple par B. Lisper [Lis93], considèrent une généralisation des tableaux où l'ensemble des indices est $\mathbb{Z}^n$. L'implémentation efficace des opérations sur les champs de données pose la question de la détermination de leur domaine de définition [LC94] ; le peu de résultats existants s'appuie sur le modèle polyédrique développé en parallélisation automatique. Plus généralement, nous avons introduit la notion de champ de données sur un groupe, ou GBF [GMS95, Gia98], afin de généraliser la notion de champs de données à des structures de données plus complexe qu'un tableau. L'originalité des GBF est de considérer le groupe des déplacements entre les positions d'une organisation. Comme pour les champs de données standards, les fonctions considérées sont partielles, ce qui marque une grande différence avec l'objectif adopté ici.

Les travaux sur l'inférence de la géométrie d'un tableau étaient peu nombreux en 1990, quand nous avons développé un algorithme pour 81/2. On pouvait citer simplement les travaux de [Tha91]. Depuis le sujet a connu un certain développement [Ken94, Vio97, JS96].

Les *travaux pionniers* présentés dans [Sij89] introduisent la notion de définition récursives *productives* : la définition récursive de la liste l, spécifiée par $l = \varphi(l)$, est productive si chaque élément de l peut être calculé en un temps fini. Pour les listes, et plus généralement pour les structures de données, la notion de productivité coïncide avec la notion d'élément maximal. Une fonction φ est dite productive si l'image d'un élément productif est productif.

Notre notion de calculabilité par préfixe est plus fine que la notion de productivité : l'exemple (4) est productif mais pas progressif. Cela est dû au fait que nous avons pour objectif d'assurer un ordonnancement des calculs à la compilation. La notion de progression que nous avons introduite pour les programmes d'une équation sur les vecteurs correspond à une notion plus forte que la notion de I-productivité sur les listes infinies. Contrairement à cette dernière, nous pouvons la généraliser aux systèmes d'équations et le prédicat `prog` permet de tester si un programme est progressif. La productivité des définitions récursives de listes infinies est aussi abordé sous l'angle du modèle de calcul à flot de données dans [Wad81] et [LM87].

Ces travaux sont surtout motivés par l'analyse des définitions récursives de suites infinies et les outils développés se focalisent sur cette structure de données. L'aspect « borné » des structures de données usuelles est donc ignoré.

5.3. Perspectives

Nous voulons bien sur étendre le cadre présenté ici à d'autres structures de données. Nous voulons aussi considérer plusieurs ordres $\sqsubseteq_{\mathcal{O}}$ différents pour la même structure de données (par exemple la définition $riota = (riota_{+1} \diamond^1 n)$: n permet de calculer le vecteur $iota$ « en partant de la fin », en adoptant une définition adaptée pour $\diamond^p$ et une notion de sous-structure « suffixe » des vecteurs).

Mais au delà de ces développements, d'interessantes questions se posent. D'abord le sujet n'est pas sans rappeler les structures de données concrètes introduites pour étudier la séquentialité [Cur86]. Peux t'on utiliser les outils conçus dans ce domaine pour développer notre approche ?

Notre objectif n'est pas de décrire un ordonnancement totalement séquentiel du calcul, mais de savoir si un ordonnancement des calculs compatible avec la structure de $\mathcal{O}$ aboutit à la construction du même objet final que l'itération du point-fixe. Cela amène plusieurs questions : peut on relier la notion de calculabilité par préfixe à l'ordre induit par $\sqsubseteq_{\mathcal{O}}$ sur $\top_{\mathcal{O}}$ à travers les fonctions j ? Cela amène aussi à étudier plus précisément la structure de $\mathcal{O}$ (formellement, $\mathcal{O}$ exhibe un structure topologique de complexe simpliciale abstrait, ce qui ouvre la voie à une vision géométrique des structures de données).

Références

[BL97] G. Bergeron, F. ansd Labelle and P. Leroux. *Combinatorial species and tree-like structures*, volume 67 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 1997. isbn 0-521-57323-8.

[Cur86] Pierre-Louis Curien. *Categorical combinators, sequential algorithms and functional programming*. Research notes in theoretical computer science. Pitman, 1986. Second, revised edition, Birkhauser (1993).

[Gia92] J.-L. Giavitto. Typing geometries of homogeneous collection. In *2nd Int. workshop on array manipulation, (ATABLE)*, Montréal, 1992.

[Gia98] Jean-Louis Giavitto. Rapport scientifique en vue d'obtenir l'habilitation à diriger des recherches, May 1998. http://www.lri.fr/~giavitto/Export/habilitation_anglais.ps.gz.

[GMS95] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. Jr. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *Lecture Notes in Computer Science*, pages 209–215, Beaune (France), 2-4 October 1995. Springer-Verlag.

[GSM92] J.-L. Giavitto, J.-P. Sansonnet, and O. Michel. Inférer rapidement la géometrie des collections. In *Workshop on Static Analysis, Bordeaux*, 1992.

[H+96] Paul Hudak et al. *Report on the programming language HASKELL a non-strict, purely functional language, version 1.3.* Yale University, CS Dept., May 1996.

[Jay95] C. Barry Jay. A semantics of shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[JS96] C. B. Jay and M. Sekanina. Shape checking of array programs. Technical Report 96.09, University of Technology, Sydney, 1996.

[Ken94] Andrew Kennedy. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362, Edinburgh, U.K., 11–13 April 1994. Springer.

[LC94] B. Lisper and J.-F. Collard. Extent analysis of data fields. Technical Report TRITA-IT R 94:03, Royal Institute of Technology, Sweden, January 1994.

[Ler93] Xavier Leroy. The caml light system, release 0.6 documentation and user's manual. Technical report, INRIA, September 1993.

[Lis93] B. Lisper. On the relation between functional and data-parallel programming languages. In *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures.* ACM, ACM Press, June 1993.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proc. of the IEEE*, 75(9), September 1987.

[Mos90] P. D. Mosses. *Handbook of Theoretical Computer Science*, volume 2, chapter Denotational Semantics, pages 575–631. Elsevier Science, 1990.

[Sij89] B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.

[Tha91] S. Thatte. A type system for implicit scaling. *Science of computer programming*, 17:217–245, 1991.

[Vio97] Eric Violard. Typechecking of PEI expressions. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing, Third International Euro-Par Conference*, number 1300 in LNCS, pages 521–529, Passau, Germany, August 1997. Springer-Verlag.

[Wad81] W. W. Wadge. An extensional treatment of dataflow deadlock. *Theoretical Computer Science*, 13(1):3–15, 1981.