Calcul des adjoints et programmation paresseuse

Jerzy Karczmarczuk

Dept. d'Informatique, Université de Caen, France
(`mailto:karczma@info.unicaen.fr`*)*

Résumé

Nous présentons une réalisation purement fonctionnelle et paresseuse de la technique du mode inverse de différentiation automatique. Cette technique, qui permet de calculer de manière précise et efficace les dérivées des expressions numériques dans un programme, est devenue indispensable dans plusieurs branches de programmation scientifique. Le mode inverse ou adjoint demande souvent l'usage des lourds paquetages extérieurs, et du pré-traitement du programme source. Grâce aux techniques de programmation paresseuse, qui rendent possible le codage des programmes «circulaires», nous montrons comment intégrer de manière facile et transparente le codage des dérivées adjointes dans le programme même. Le résultat est utilisable, même si plusieurs optimisations seront nécessaires pour rendre le paquetage utilisable dans des cas plus sérieux. Nous montrons aussi comment l'approche monadique à la programmation fonctionnelle permet d'établir un lien entre des domaines apparemment différents : le calcul des adjoints, et la propagation des attributs durant l'analyse syntaxique.

1. Introduction

1.1. Différentiation automatique

Dans ce travail nous élaborons une technique particulière de différentiation automatique (parfois nommée : *différentiation algorithmique*, ang. : *Computational Differentiation*) des programmes numériques. Notre implantation est purement fonctionnelle, et exploite la sémantique paresseuse de manière assez intense. Le paquetage a été réalisé en Haskell et testé avec l'interprète Hugs.

En général, les outils de différentiation automatique (DA) servent à calculer dans un programme *numérique* les dérivées, gradients, Jacobiens, etc. des expressions par rapport à un ou plusieurs objets considérés comme des *variables* (indépendantes), ce qui permet par exemple de calculer la dérivée d'une procédure par rapport à son paramètre, pour une valeur numérique concrète. Le mot «numérique» signifie ici deux choses :

- aucune manipulation symbolique des expressions n'a lieu, les variables n'ont pas de noms comme dans un programme interprété par un système de calcul formel, et les expressions (p. ex. **`x*y`**) ne sont pas des arbres qui puissent être déstructurés en composantes, ce qui permettrait leur traitement symbolique ;
- on obtient la valeur numérique (réelle, complexe, etc.) de l'expression dérivée, calculée en fonction d'autres valeurs numériques définissant son contexte.

On n'utilise pas des différences finies. Les techniques de DA sont *exactes*, c'est-à-dire, les dérivées sont calculées avec la même précision que toutes les autres expressions numériques, ce qui est déterminé par les propriétés du processeur et les librairies de fonctions sur les nombres flottants. Même si une dérivée s'exprime par une formule plus longue, ce qui implique que la précision numérique peut être légèrement inférieure à celle de l'expression «principale», aucune instabilité, aucune perte de précision spécifique à la méthode n'a lieu.

La DA est un vénérable et important domaine de recherche et d'applications dans le monde d'ingénierie et du calcul scientifique, voir p. ex. [1, 2, 3, 4, 5]. Elle est utilisée pour analyser la stabilité des équations différentielles, ou résoudre quelques problèmes variationnels, et dans de nombreux autres cas. Même dans les mathématiques discrètes la différentiation peut servir comme un outil permettant de calculer les coefficients combinatoires à partir de leur fonction génératrice [6].

La technique est basée sur l'observation suivante : le calcul différentiel est algorithmiquement simple, et toute manipulation des expressions numériques permettant de calculer les dérivées, peut être *facilement* effectuée par le compilateur (ou un logiciel de pré-traitement de la source) capable d'étendre la sémantique du programme original, en accord avec les règles du calcul différentiel. Ainsi, avec pour chaque expression originale on peut calculer simultanément sa dérivée à partir des constantes dont les dérivées sont égales à zéro, et la (ou les) *variables*, dont les dérivées sont triviales aussi. Le programme étendu suit la règle de Leibniz, et, plus généralement – la règle de composition : $(f(g(x))' = f'(g(x)) \cdot g'(x)$, et en composant des expressions plus complexes, calcule leurs dérivées pour les mêmes valeurs numériques de toutes les variables du programme. L'implantation la plus primitive consiste à surcharger l'arithmétique sur des paires de valeurs numériques : (e, e') est une valeur étendue représentant une expression et sa dérivée (pour plus de simplicité nous discutons ici le cas 1-dimensionnel, facile à généraliser).

Dans [7] nous avons défini l'arithmétique sur des séquences infinies $[e, e', e'', e^{(3)}, \ldots]$ représentant les expressions avec *toutes* leur dérivées, ce qui a permis de définir une algèbre différentielle close, c'est-à-dire un domaine qui dispose de la panoplie arithmétique standard, et d'un opérateur de *dérivation* $\mathbf{d}$, linéaire, et satisfaisant la règle de Leibniz : $\mathbf{d}(ef) = e(\mathbf{d}f) + (\mathbf{d}e)f$. Dans ce domaine la «variable» x est une structure équivalente à la liste $[x, 1, 0, 0, \ldots]$, et les constantes c deviennent évidemment des listes $[c, 0, 0, \ldots]$. Voici quelques définitions de l'arithmétique étendue dans ce domaine. Pour $e = (e_0 : \hat{e})$ et $f = (f_0 : \hat{f})$, où l'opérateur $(:)$ est un constructeur de listes (il forme e en ajoutant e_0 devant la liste $\hat{e}$)) nous aurons

$$\begin{array}{rcll}
e + f & = & (e_0 + f_0 : \hat{e} + \hat{f}) & \\
e \cdot f & = & (e_0 \cdot f_0 : e\hat{f} + \hat{e}f) & \\
1/e & = & w \quad \textbf{où} \quad w = (1/e_0 : -\hat{e} \cdot w^2) & \\
\exp(e) & = & w \quad \textbf{où} \quad w = (\exp(e_0) : w\hat{e}) & \\
\sqrt{e} & = & w \quad \textbf{où} \quad w = (\sqrt{e_0} : 1/2 \cdot \hat{e}/w & \\
\log(e) & = & (\log(e_0) : \hat{e}/e) &
\end{array} \tag{1}$$

etc. L'opérateur de dérivation $\mathbf{d} : e \to \hat{e}$ récupère la queue de la liste, et ainsi nous aurons «gratuitement» aussi les dérivées de degré supérieur. Dans un langage paresseux $\mathbf{d}$ n'est pas trivial, car il peut forcer l'évaluation des expressions différées qui constituent cette queue. Notons que les définitions ci-dessus n'ont pas de clauses terminales, elles sont «ouvertes», et plutôt co-récursives que récursives. Les listes sont vraiment traitées comme infinies, ce qui exige l'usage de la sémantique paresseuse, même si dans notre implantation on évite la construction des listes infinies triviales en utilisant une structure de données particulière, où la liste infinie de zéros est remplacée par une constante spéciale. Et tout cas il faut éviter de demander la valeur des éléments dont on n'a pas besoin.

Cette méthode constitue le *mode direct* de la différentiation algorithmique. Le reste du travail décrit une technique alternative, dite «inverse», qui dans quelques contextes semble être plus naturelle, et parfois aussi plus efficace, surtout dans le cas multi-dimensionnel «creux», où toutes les expressions intermédiaires dépendent d'un très petit nombre de paramètres, et où le nombre de résultats finaux est beaucoup plus petit que le nombre de variables indépendantes. (La généralisation de la méthode directe aux structures régulières : tenseurs et formes différentielles en N dimensions peut être trouvée dans l'article [8].)

1.2. Mode inverse de différentiation

Le nombre de résultats d'un programme peut être très inférieur au nombre de variables indépendantes dans des cas comme l'analyse de la sensibilité des processus techniques et naturels (dépendance de la solution finale de l'ensemble des paramètres du système et des conditions initiales). Exemples : météorologie et océanographie, diagnostic des réacteurs nucléaires, développement de la biosphère, etc. Nous commençons par une multitude de paramètres, mais un seul résultat nous intéresse, par exemple la température.

Comme il est précisé dans de nombreux textes sur DA, dans ces circonstances il n'est pas nécessaire de maintenir toutes les dérivées partielles des expressions intermédiaires par rapport aux variables indépendantes. Il suffit de définir pour chaque variable (initiale ou intermédiaire) son **adjoint** $\overline{e}$ – la dérivée du *résultat final* par rapport à cette variable [1]. Il est évident qu'au moment de la définition d'une variable intermédiaire dans le programme, le résultat final n'est pas connu, et l'adjoint ne peut être effectivement calculé. Les paquetages de DA qui exploitent cette technique sont très compliqués., Nous allons montrer que la sémantique paresseuse simplifie l'implantation du mode inverse de manière spectaculaire, même si pour pouvoir l'appliquer aux problèmes réels, un sérieux travail d'optimisation semble encore nécessaire.

Décrivons la dérivation du mode inverse d'une manière plus formelle. Supposons que $(x_1, x_2, \ldots, x_M)$ constitue la collection des variables indépendantes dans le programme. Si à l'aide d'un «oracle» toutes les conditions dynamiques dans le programme sont résolues, les branchements effectués et les boucles aplaties, le programme peut être modélisé par l'enchaînement des définitions fonctionnelles :

$$\begin{array}{rcl} x_{M+1} & \leftarrow & f_{M+1}(x_1, \ldots, x_M)\,, \\ x_{M+2} & \leftarrow & f_{M+2}(x_1, \ldots, x_{M+1})\,, \\ \ldots & & \\ R = x_N & \leftarrow & f_N(x_1, \ldots, x_{N-1})\,, \end{array} \tag{2}$$

où, pour l'homogénéité de la notation toutes les variables intermédiaires portent les noms «x_p». Naturellement, cet ensemble peut être complété par les affectations $x_k \leftarrow f_k()$, où f_k est une fonction constante qui fournit la valeur initiale de la variable en question pour $k \leq M$. Un petit nombre d'équations (2), peut-être seulement la dernière, qui définit R, spécifient les résultats du programme. Les fonctions f sont typiquement très «creuses», et se réduisent aux opérateurs unaires ou binaires. Pour chaque affectation $g \leftarrow f(e_1, e_2, \ldots, e_k)$ les adjoints *des arguments à droite* sont obtenus par

$$\overline{e}_k \leftarrow \overline{e}_k + \overline{g}\frac{\partial f}{\partial e_k}\,. \tag{3}$$

Ceci n'est plus une définition (fonctionnelle) d'une variable intermédiaire, mais une structure impérative : la mise à jour d'une variable existante, puisque $\overline{e}_k$ peut figurer dans plusieurs endroits du programme-source. Pire, même si les instructions adjointes peuvent être facilement compilées dans le contexte de l'instruction originale, *elles ne peuvent y être exécutées*, car $\overline{g}$ sera connu plus tard, quand g sera *utilisé*.

Les paquetages qui exploitent le mode inverse, p. ex. [4, 9] et plusieurs autres, exécutent le programme étendu en deux étapes. D'abord le programme d'origine est exécuté, les valeurs de toutes les variables «normales» calculées, et en parallèle toutes les instructions adjointes sont stockées sur une structure de données linéaire, dite «bande» (ang. *tape*, normalement un fichier). Après avoir calculé le résultat final, la bande est lue et interprétée à l'envers, de la fin jusqu'au début, où se trouvent les premières affectations des adjoints des variables indépendantes. Cette procédure est visiblement lourde, et un pré-traitement important du code-source est nécessaire, si le programme est écrit dans un langage classique impératif.

Par exemple, si on veut calculer $z'(x)$, où z est défini par le programme :

$$y = \sin(x); \quad z = y^2 - x/y\,, \tag{4}$$

il faut d'abord calculer y et z pendant la phase «avant» du programme, ensuite les initialiser les adjoints : $\overline{z} \leftarrow 1$; $\overline{x} \leftarrow 0$; $\overline{y} \leftarrow 0$, et renverser le flot de contrôle.

$$\begin{array}{rcl} z = y^2 - x/y; & \text{donne} & \overline{x} \leftarrow \overline{x} + \overline{z}(-1/y); \\ & & \overline{y} \leftarrow \overline{y} + \overline{z}(2y + x/y^2); \\ y = \sin(x); & \text{donne} & \overline{x} \leftarrow \overline{x} + \overline{y}\cos(x)\,. \end{array} \tag{5}$$

Finalement $\overline{x} = -1/\sin(x) + \cos(x)\left(2\sin(x) + x/\sin(x)^2\right)$ est la valeur de dz/dx.

Voici la dérivation générale de cet algorithme. Les dérivées sont définies par les enchaînements satisfaits par les matrices de Jacobi :

$$\mathbf{J}_{ik} = \frac{dx_i}{dx_k} = \delta_{ik} + \sum_{j=k}^{i-1} \frac{\partial f_i}{\partial x_j}\frac{dx_j}{dx_k}\,. \tag{6}$$

[1] Ceci n'a pas de rapport direct avec les adjoints dans la théorie des catégories

La forme matricielle de cette équation est : $\mathbf{J} = \mathbf{I} + \mathbf{DJ}$, où

$$\mathbf{D}_{ik} = \frac{\partial f_i}{\partial x_k} = \begin{pmatrix} 0 & 0 & 0 & \dots \\ \partial f_2/\partial x_1 & 0 & 0 & \dots \\ \partial f_3/\partial x_1 & \partial f_3/\partial x_2 & 0 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} . \tag{7}$$

Si on commence par x_{M+1}, et si on suit la chaîne jusqu'à x_N, on calcule itérativement $\mathbf{J}_{ik}$, et ceci constitue la méthode directe. Mais on peut (en principe) calculer d'abord les dernières dérivées partielles, et propager les valeurs vers l'«arrière». Sur le plan d'efficacité ceci peut être intéressant, surtout si on a besoin seulement de la dernière ligne de la matrice de Jacobi, c'est-à-dire les éléments $\overline{x}_k = \mathbf{J}_{Nk} = dx_N/dx_k$. Il est évident que les matrices $\mathbf{J}$ et $\mathbf{D}$ commutent, et si l'équation définissant $\mathbf{J}$ est réécrite en sa forme adjointe : $\mathbf{J} = \mathbf{I} + \mathbf{JD}$, ou bien en forme développée :

$$\mathbf{J}_{ik} = \delta_{ik} + \sum_{j=k+1}^{i} \mathbf{J}_{ij}\mathbf{D}_{jk} , \tag{8}$$

la dernière ligne satisfait

$$\overline{x}_k = \delta_{Nk} + \sum_{j=k+1}^{N} \overline{x}_j \mathbf{D}_{jk} . \tag{9}$$

Le calcul des adjoints n'est pas seulement lourd, mais aussi dangereux, car il peut «convertir le temps en espace». Si le programme exécute une boucle, chaque ré-affectation d'une variable intermédiaire (ou création d'une nouvelle instance de cette variable, si la boucle est réalisée par la récursivité terminale), engendre de *nouvelles* instructions adjointes, qui devront être mémorisées sur la «bande». Elle peut devenir très longue, même si le programme est court. Plusieurs optimisations seraient essentielles dans des cas plus complexes [10, 11], mais leur implantation automatique est difficile. Le reste de l'article est consacré à l'implantation fonctionnelle du mode inverse, où la bande est remplacée par une structure fonctionnelle qui cache des expressions différées. Nous voulons implanter ce mode de manière simple, transparente et pratique pour un utilisateur intéressé par le calcul scientifique, mais nous ne discutons pas ces optimisations, sauf une, qui consiste à affaiblir la paresse, et à évaluer da manière stricte toutes les variables intermédiaires qui peuvent être instanciées sans attendre.

2. Application de la sémantique non-stricte

2.1. Transformateurs d'états et programmation paresseuse

Notre approche est basée sur les techniques monadiques de programmation fonctionnelle. Bien sûr, nous ne pouvons pas discuter ici les monades dans leur généralité, rappelons cependant l'idée essentielle de la technique monadique qui permet d'implanter de manière fonctionnelle les effets de bord : la monade ST (*State Transformer*), décrite p. ex. dans [12].

Dans un programme fonctionnel l'évaluation d'une expression est «pure», on obtient une valeur appartenant, disons, à un type dénoté symboliquement par **`a`** (nombres chaînes, listes, etc.), et c'est tout. Pour modéliser la notion d'état qui subit des modifications pendant l'exécution du programme, on remplace les expressions de type **`a`** par des objets fonctionnels de type **`ST a`** $\equiv$ **`s -> (a,s)`**, où **`s`** dénote le type des objets représentant l'état : un compteur incrémenté à chaque évaluation d'une variable, une chaîne de caractères qui contient la trace de l'exécution du programme, les buffers d'entrée/sortie, etc. Si une expression e est neutre par rapport au changement de cet état, elle deviendra une fonction qui laisse l'état intact : **`\s -> (e,s)`**. (Rappelons qu'en Haskell le symbole «**`\`**» dénote λ.) Formellement aucune «transformation» n'a lieu : un opérateur agit sur l'état initial, et créé un autre état, appelé final. Les deux sont des données ordinaires. Toutefois, si le compilateur peut prouver qu'après la création de l'état final, le programme n'accède plus à l'original, il peut optimiser leur gestion en modifiant l'état initial sur place. Ceci est l'essentiel de la formulation monadique de la programmation impérative dans un langage fonctionnel.

Dans le contexte de cet article l'état sera l'ensemble des adjoints. Il sera «caché» dans le programme utilisateur, comme on cache le flot d'entrée dans les définitions des analyseurs syntaxiques monadiques. Cependant, les propriétés de cet état sont très inhabituels.

Toute fonction qui agissait sur les expressions et produisait des valeurs du même type, doit maintenant engendrer les transformateurs d'état. La façon dont elle le fait, mais l'action d'une telle fonction sur un transformateur `m` de type `ST a` peut être définie de manière universelle. Dans la définition ci-dessous l'opérateur `>=>` dénote l'application étendue d'une fonction `f` à un objet de ce type :

```
f >=> m =
 \s_init ->
     let (x,s_mid) = m s_init
         (y,s_final) = (f x) s_mid
     in  (y,s_final)
```

L'interprétation est simple : le résultat est une fonction qui doit agir sur un état initial. D'abord le transformateur `m` agit sur cet état, et engendre un état intermédiaire `s_mid` combiné avec `x` – l'argument effectif de la fonction `f`. Cette fonction agissant sur `x` crée un transformateur qui change l'état intermédiaire en final.

2.2. Propagation de l'état contre la flèche du temps

Dans [12] Philip Wadler a proposé une modification apparemment très bizarre de la composition des objets `ST` – les transformateurs dont l'enchaînement *propage l'état vers l'arrière dans le temps*. Voici la définition de l'application étendue :

```
f >=> m =
 \s_final ->
     let (x,s_init) = m s_interm
         (y,s_interm) = (f x) s_final
     in  (y,s_init)
```

Le résultat est une fonction qui agit sur l'état final. L'opérateur qui touche cet état est le transformateur créé par l'action de la fonction `f` sur son argument : `(f x)`. Ce transformateur rend la valeur `y` qui conceptuellement représente $f(x)$, et accessoirement engendre l'état intermédiaire qui sera consommé par `m` et transformé en l'état initial. Ceci n'est pas un simple changement de noms. Observons que l'argument `x` pour `f` est préparé par `m` qui agit sur l'état intermédiaire créé par `f`. Les définitions ci-dessus sont circulaires, les données sont réciproquement dépendantes, et une telle intrication admet une solution (un programme qui fonctionne) seulement si la sémantique des appels fonctionnels est paresseuse, par exemple si `m` n'a pas immédiatement besoin de `s_interm` pour récupérer et retourner `x`.

L'état qui suit cette règle de propagation sera appelé *état antithétique*, et c'est justement ce qu'il nous faut. Le rapport entre notre calcul des adjoints, et la technique qui exploite ces courants de dépendances antithétiques est le suivant : en effectuant le calcul définissant une variable intermédiaire, par exemple $y = f(x)$, le programme calcule simultanément $\overline{x}$ en fonction de $\overline{y}$ qui appartient au «futur». Mais ce dernier calcul reste latent, car dans le contexte de cette évaluation, personne n'a encore besoin de $\overline{x}$. Quand le programme arrive à la dernière expression qui définit le résultat final, l'état final devient connu, c'est l'adjoint du résultat, égal à 1. Dans la programmation paresseuse l'évaluation est pilotée par les demandes, et c'est à ce moment là, que le programme augmenté peut utiliser la valeur de $\overline{x}$. Toutes les opérations différées seront alors exécutées.

Il est clair pourquoi l'applicabilité de cette méthode est restreinte aux problèmes creux : si une expression $y = f(x_1, \ldots, x_k, \ldots)$ contient plusieurs variables dont elle dépend, ceci engendre plusieurs instructions adjointes, et même pour la stratégie impérative, la complexité pourra devenir quadratique par rapport au programme original. Pour la programmation paresseuse le problème est plus grave, et sera discuté plus tard.

2.3. Programmes circulaires et leur application

A priori, indépendamment de l'idée esquissée ci-dessus, il est difficile de trouver des applications immédiates pour une telle «machine à voyager dans le temps». Elle est loin d'être intuitive, et Wadler lui même n'en a donné aucun exemple concret. Certes, les programmes circulaires qui exploitent la sémantique non-stricte pour optimiser la gestion de données par un programme circulaire sont connus depuis longtemps [13], mais ils ont un goût plutôt artificiel. L'exemple canonique de Bird est une fonction qui effectue *un seul* parcours d'un arbre binaire où les feuilles contiennent des nombres, et qui construit un arbre de structure identique, mais dont les feuilles contiennent la valeur minimale stockée dans l'arbre original. Intuitivement on a besoin de deux parcours. Voici le code en Haskell :

```
data Arb a = Feuille a | Noeud  (Arb a) (Arb a)

arbmin arbre = res where
  (res,x) = abm arbre x              -- dépendance circulaire. Qu'est'ce que le "x"?
abm (Feuille y) x = (Feuille x,y)
abm (Noeud ga dr) x = (Noeud tg td, min xg xd) where
  (tg,xg) = abm ga x
  (td,xd) = abm dr x
```

La réponse à la question dans le commentaire est, naturellement : c'est le minimum des valeurs. La fonction auxiliaire **abm** fait les deux choses à la fois, cherche le minimum, et construit l'arbre. Mais la construction est paresseuse, et quand le processus se termine, le résultat **res** est le *générateur* de l'arbre, un objet fonctionnel qui contient la référence à la variable **x**. Le traitement (p. ex. l'affichage) de ce résultat déclenche la construction physique de l'arbre, ce qui équivaut à second parcours de la structure. Le programme n'est pas plus rapide, l'économie concerne les ressources humaines. Nous aurons le même phénomène dans notre construction.

Comme nous avons dit, les exemples de programmes circulaires sont souvent artificiels. Dans un programme traditionnel les dépendances entre les données et le flot de contrôle sont synchrones, et on n'a jamais besoin de référencer une valeur qui sera définie plus tard. Il existe cependant un contexte, où les dépendances entre les entités sont parfois circulaires : la propagation des attributs sémantiques lors de l'analyse syntaxique d'un programme.

Durant la transformation du programme en arbre syntaxique par un parseur équipé de procédures sémantiques, les attributs hérités (p. ex. le type forcé par une conversion de l'argument, si le résultat d'une opération possède un autre type, ou une information contextuelle comme la position d'un item), descendent de la racine dans la direction des feuilles, tandis que les attributs synthétisés montent dans la direction de la racine. La gestion «naturelle» des attributs préconise l'usage d'un parseur récursif, descendant, ce qui permet de transmettre (par l'intermédiaire des paramètres) les attributs hérités. Si, par exemple, une production syntaxique

$$A ::= \mathbf{Op}\, B \tag{10}$$

est accompagnée d'une règle sémantique $B.i = g(A.t)$, où $B.i$ est un attribut hérité, le parseur descendant de A peut avoir la forme

```
parseur_A(t) =
    ... analyse de Op et B
    ... construction sémantique de B à l'aide de t
    ... construction des attributs synthétisés de A
```

Mais un parseur ascendant, p. ex. LR(1) «ne connaît pas» la racine de l'arbre, il n'est pas récursif. Il se comporte comme un interprète qui «exécute» symboliquement le programme dans l'ordre intuitif. Pour la forme syntaxique conforme à la production (10) il analysera d'abord **Op** et **B**, et ensuite effectuera la construction de **A**. Alors le flot des attributs hérités devient antithétique à la propagation des valeurs gérées par le parseur. Quelques générateurs de parseurs interdisent l'usage des attributs hérités, mais la solution beaucoup plus naturelle et élégante est possible aussi, grâce à la programmation paresseuse.

Thomas Johnsson dans l'article [14] analyse cette solution, et avoue que les sources de son inspiration sont les programmes circulaires de Bird. Analysons la règle syntaxique qui construit une expression à partir d'une sous-expression et d'un opérateur (10).

Cette règle pilote la synthèse des attributs de A, mais c'est également ici que les attributs hérités de B sont construits. Dénotons par A_S un attribut synthétisé attaché à l'expression A, par exemple sa «valeur», et par B_I des attributs hérités. Alors l'ensemble de décorations sémantiques (affectations des attributs) peut être remplacé par la création d'un nouvel attribut synthétisé A_f qui est un objet fonctionnel défini par le programme ci-dessous (en supposant que chaque variable possède un attribut synthétisé et un hérité, pout-être composites) :

$$
\begin{array}{lll}
A_f = \lambda A_I & \rightarrow & \\
& \mathbf{let} & B_S = B_f\,(B_I) \\
& & \{\ldots \text{ déf. des attributs de } A \text{ et du } B_I\ \ldots\} \\
& \mathbf{in} & A_S
\end{array}
$$

où typiquement A_S dépend de B_S, et puisque B_I dépend des attributs de A, les définitions sont croisées. Johnsson utilise l'évaluation paresseuse pour résoudre de manière effective la propagation des attributs, mais il fait mieux : il exploite les attributs comme un paradigme universel, applicable en tant que technique de programmation générale. Il reconstruit dans son article quelques programmes circulaires de Bird avec une simplicité et élégance remarquables. Il est vraiment amusant de voir comment la «perversion» de la monade ST suggère le même style de programmation, mais il est encore plus amusant de découvrir que les programmeurs en Fortran en ont besoin, et qu'ils utilisent des techniques analogues depuis déjà une bonne dizaine d'années, à l'aide des astuces de programmation très pénibles. Les attributs X_I peuvent être considérés ici comme «les adjoints», et la fonction définie ci-dessus comme un transformateur d'états.

Le lecteur intéressé par la gestion paresseuse des attributs trouvera beaucoup d'informations dans la documentation du système de compilation Elegant de Lex Augusteijn [15].

3. Construction du mode inverse de DA

3.1. Arithmétique des adjoints

Le style monadique classique utilise souvent une syntaxe particulière : l'opérateur «*bind*» qui enchaîne les monades (c'est la transposition de notre opérateur **>=>**), ou la forme syntaxique «*do*» qui simule un style impératif de programmation. Nous voulons éviter toute syntaxe spéciale, et écrire les programmes de manière très traditionnelle et fonctionnelle, grâce à la surcharge des opérateurs standard. Ce qui restera de l'idée monadique est le fait que l'état est caché de la surface du programme.

Il n'y aura pas de mise à jour impérative et incrémentale des adjoints dans le programme comme dans (3). Nous construirons seulement les contributions spécifiées par cette équation : $\overline{g}\partial f/\partial e_k$, et le résultat sera directement la somme précisée dans l'équation (9).

Par simplicité nous commençons par la discussion du cas 1-dimensionnel. L'«état final» est l'adjoint du résultat final, c'est à dire 1. l'état initial est l'adjoint de la variable indépendante. Quand la variable de différentiation est utilisée pour la première fois, son adjoint apparaît aussi, mais son statut existentiel est un peu fantomal, il ne sera réellement formé qu'en fin de programme, quand le transformateur final est appliqué à 1.

Dans le cas 1-dimensionnel l'état appartient au même type que toute autre expression, typiquement il s'agit d'un nombre flottant. Le transformateur d'états, et les générateurs des constantes et de la variable de différentiation sont définis par

```
newtype Ldif a = Ld (a->(a,a))

lCnst c = Ld (\n -> (c, 0))
lDvar x = Ld (\n -> (x, n))
```

ce qui est parfaitement intuitif : l'adjoint d'une constante n'apporte rien, et l'adjoint de x engendré par l'instruction $n = x$ est $\overline{x} = \overline{n}$. La construction **newtype** en Haskell définit un type physiquement synonymique avec un autre, ici – avec le type fonctionnel **a -> (a,a)**, mais formellement différent, ce qui est assuré par la présence de la balise **Ld** dans le programme source, mais qui ne laisse pas de trace pendant la compilation. En Haskell il existe une autre méthode de définition littérale des synonymes :

```
type Ldif a = (a->(a,a))
```

mais l'usage de **type** est très restreint, en particulier on ne peut pas définir des opérateurs surchargés pour un tel type (les types-synonymes en Haskell ne peuvent être des instances des classes de types).

Pour des fonctions quelconques, dont toutes les dérivées (formelles) sont connues, nous définissons leur «lifting» générique dans le domaine **Ldif**, **llift** pour les fonctions unaires et**dllift** pour les opérateurs binaires :

```
llift f f' (Ld pp) =
  Ld (\n->let (p,pb)=pp eb
              eb=(f' p)*n in (f p,pb))

dllift op op1' op2' (Ld pp) (Ld qq) =
  Ld (\n->let (p,pb)=pp ep;  (q,qb)=qq eq
              ep=(op1' p q)*n;  eq=(op2' p q)*n
          in (op p q, pb+qb) )
```

ce qui permet immédiatement la construction des fonctions élémentaires étendues, par exemple dans le domaine **Ldif** le logarithme est défini par **log = llift log recip**, et le cosinus par **cos = llift cos (negate.sin)** . Cependant, les opérateurs arithmétiques standard sont un peu optimisés, et leur définitions sont courtes (même si un peu difficiles à lire...)

```
negate (Ld pp)=Ld (\n->let (p,pb)=pp (negate n)
                       in  (negate p,pb))
(Ld pp)+(Ld qq) = Ld (\n ->
  let (p,pb)=pp n;  (q,qb)=qq n
  in  (p+q, pb+qb) )
(Ld pp)-(Ld qq) = Ld (\n ->
  let (p,pb)=pp n;  (q,qb)=qq (negate n)
  in  (p-q, pb+qb) )

(Ld pp)*(Ld qq) = Ld (\n ->
  let (p,pb)=pp (n*q);  (q,qb)=qq (p*n)
  in  (p*q, pb+qb) )
(Ld pp)/(Ld qq) = Ld (\n ->
  let (p,pb)=pp (recip q*n);  (q,qb)=qq eq
      eq=negate (p/(q*q))*n
  in  (p/q, pb+qb) )

recip (Ld pp) = Ld (\n ->
  let (p,pb)=pp eb;  w=recip p
       eb=negate (w*w)*n   in  (w,pb))

exp (Ld pp) = Ld (\n ->
  let (p,pb)=pp (w*n); w=exp p in (w,pb))

sqrt (Ld pp) = Ld (\n ->
  let (p,pb)=pp eb; w=sqrt p
      eb=(0.5/w)*n   in  (w,pb))

-- ... etc. ...
```

Notez la présence de l'addition des dérivées pour tout opérateur binaire. C'est ici que le programme accumule les adjoints et construit la somme (9).

3.2. Comment utiliser le paquetage ?

Supposons que le point de départ soit une fonction numérique, par exemple la définition d'une fonction hyperbolique :

```
ch z = let e = exp z
       in (e + recip e)/2.0
```

Il faut d'abord s'assurer que la fonction est reconnue par Haskell comme polymorphe, c'est-à-dire que toutes les entités, données et opérateurs, sont surchargés. La définition ci-dessus malgré la présence d'une constante explicite 2.0 satisfait cette contrainte, car Haskell surcharge automatiquement les constantes numériques (force leur coercition par un appel implicite de `fromDouble` ou `fromInteger`).

Afin de calculer la dérivée de cette fonction par rapport à son paramètre x pour, disons, $x = 0.5$ il faut

– appeler, et extraire le transformateur du type `Ldif` :
  `res = ch (lDvar 0.5)` ;
– appliquer le résultat à 1 :
  `paireFinale = res 1`

ce qui donne $(1.12762597, 0.521095305)$. Bien sûr, si on applique la fonction `ch` à `lCnst 0.5`, on obtient $(1.127626, 0.0)$.

Dans la programmation pratique on peut combiner l'extraction du transformateur d'états de la structure `Ldif` et son application à 1 dans une fonction d'évaluation, ou peut-être tout simplement dans la fonction d'affichage du résultat final.

4. Généralisations et applications

4.1. Dérivées d'ordre supérieur

La possibilité de calculer les dérivées d'ordre supérieur est assurée «gratuitement» par le polymorphisme de Haskell. Il suffit d'ajouter un niveau supplémentaire d'extension `Ldif` au nombre qui représente la variable de différentiation, d'appeler la fonction concernée, par exemple : `ch (lDvar (lDvar 0.5)`, et en extraire la valeur désirée. Cependant en général le mode inverse n'est pas bien adapté aux dérivées d'ordre supérieur, et leur manipulation devient vite assez pénible, il suffit d'observer que le résultat d'évaluation de `recip (lDvar (lDvar (lDvar 1.0)))` est $(((1.0, -1.0), (-1.0, 2.0)), ((-1.0, 2.0), (2.0, -6.0)))$. L'analyse de l'algorithme et de sa sémantique dans le cas où le type de base n'est plus numérique, mais constitue un transformateur d'états est difficile, observons seulement que la complexité de la structure résultante croît exponentiellement avec le nombre de `lDvar`...

Les paquetages adaptés à Fortran ou C++ d'habitude renoncent à calculer les dérivées d'ordre supérieur à 2. Si l'utilisateur a besoin de ces dérivées dans un programme fonctionnel, nous préconisons l'usage de la méthode directe [7, 8].

4.2. Cas multi-dimensionnel

Dans la section (1.2) nous avons souligné que les techniques de DA en mode inverse sont particulièrement intéressantes dans des cas M-dimensionnels (avec M variables indépendantes, dont les adjoints doivent être calculés) irréguliers, non-géométriques, où les matrices de Jacobi sont creuses. La généralisation de la méthode proposée est directe et naturelle, ce qui permet de calculer les gradients, Jacobiens, Hessiens etc. relativement facilement. Malheureusement l'efficacité de l'algorithme en souffre beaucoup. L'indice k dans l'équation (9) parcourt toutes les dimensions, et si nous voulons garder la simplicité de notre codage fonctionnel, les adjoints seront des vecteurs (très creux), tandis que dans l'approche impérative les variables adjointes restaient scalaires.

Nous avons donc redéfini le type `Ldif`, en remplaçant le type des adjoints par une liste, ou plutôt par un type synonymique à une liste :

```
newtype Adj a = Ad [a]
newtype Ldif a = Ld (Adj a->(a,Adj a))
```

Les adjoints des constantes sont des listes de zéros, et la k-ième variable indépendante x_k est convertie en `Ld (\nn->(xk,Ad[0,0,...,1,0,...]))` (le constructeur `lDvar` prend un paramètre supplémentaire :la position de 1 dans la liste des adjoints).

Le reste du code subit des modifications cosmétiques, par exemple la somme des adjoints demande la construction de l'opérateur `(+)` surchargé qui agit sur les listes additionnant les éléments, et le produit `n*q` où maintenant `n` possède plusieurs composantes, devient `n*>q`, où l'opérateur `(*>)` est défini par

```
Ad n *> q = Ad (map (q*) n)
```

et créé la liste de produits de `q` par les éléments $\mathbf{n_k}$.

Le résultat final est engendré par le transformateur agissant sur la liste `Ad[1,1,...,1]`. Bien sûr, rien n'empêche de calculer les dérivées d'ordre supérieur aussi dans le cas multi-dimensionnel.

4.3. Une optimisation légère

Notre codage des adjoints n'est pas très efficace, même si la complexité de nos algorithmes se comporte formellement de la même façon que dans d'autres implantations du mode inverse. Nous n'avons encore essayé aucune optimisation présente dans des populaires paquetages de DA, la plupart de ces optimisations étant spécifique de la programmation impérative.

L'usage de la mémoire mérite une analyse approfondie qui ne peut être effectué ici. Dans un programme paresseux les expressions différées occupent la mémoire sous forme de *thunks* – fermetures composées du code compilée et de références aux objets appartenant à l'environnement de ce code. Thunks peuvent être combinées avant d'être évaluées, ils remplacent la «bande» des programmes impératifs et leur taille peut devenir très grande lors de leur application – rappelons que *toutes* les opérations différées seront alors exécutées, et il n'y a pas de réutilisation des variables adjointes ; c'est ici qu'une optimisation s'impose.

Nous proposons ici une légère modification du formalisme, ce qui permet d'alléger un peu la surcharge causé par la suspension de *toutes* les opérations. En effet, normalement les valeurs principales peuvent être calculées immédiatement, pendant la phase directe du programme, seulement les adjoints doivent rester sous la forme fonctionnelle, différée. Introduisons donc la structure de donnée suivante et les générateurs primitifs (le cas 1-dimensionnel est présenté) :

```
data Rdif a = Rd a (a->a)

rCnst c = Rd c (\_->0)
rDvar x = Rd x id
```

L'expression `Rd e g` contient directement une valeur numérique `e`, mais son deuxième champ est une «promesse» de fournir l'adjoint quand cette expression sera utilisé dans un contexte où l'adjoint pourra être calculé. Pour récupérer la dérivée du résultat final `Rd r g` il faut appliquer `g` à 1. Le codage qui suit est directement inspiré du modèle précédent, mais vu la représentation fonctionnelle explicite de la partie adjointe, il peut être réalisé dans un langage strict. Bien sûr, le codage reste fonctionnel, la possibilité de créer dynamiquement les fermetures reste indispensable.

L'arithmétique est une simplification du code présenté ci-dessus. Le «lifting» générique prend la forme suivante :

```
rlift f f' (Rd p pr) =
   Rd (f p) (\r->pr(r*f' p))
drlift f f1' f2' (Rd p pr) (Rd q qr) =
   Rd (f p q)
      (\r->pr(r*f1' p q)+qr(r*f2' p q))
```

Notons que l'opérateur `f` est appliqué tout de suite, ce qui élimine les références croisées entre l'expression et son adjoint. Ainsi nous nous sommes éloignées du modèle anti-temporal original. L'économie de mémoire introduite par cette optimisation peut être importante, et le temps de calcul de la partie principale du résultat est réduit aussi, il doit être comparable au temps d'exécution du programme original, sans dérivées. Les opérations arithmétiques étendues deviennent :

```
negate (Rd e _) = Rd (negate e) (\r->(negate r))

(Rd p pr)+(Rd q qr)=Rd (p+q) (\r->pr(r)+qr(r))
(Rd p pr)-(Rd q qr)=Rd (p-q) (\r->pr(r)+qr(negate r))

(Rd p pr)*(Rd q qr)=Rd (p*q) (\r->pr(r*q)+qr(r*p))

(Rd p pr)/(Rd q qr)=
     Rd (p/q) (\r->pr(r/q)+qr(negate r*p/(q*q)))
recip (Rd p pr)=Rd w (\r->pr(negate r*w*w))
                 where w=recip p

exp (Rd p pr) = Rd w (\r -> pr(r*w)) where w=exp p
sqrt (Rd e pr) = Rd w (\r->pr(0.5*r/w))
                 where w=sqrt e
-- et, tout simplement ...
log = rlift log recip
sin = rlift sin cos
cos = rlift cos (negate . sin)
```

5. Conclusions et perspectives

L'importance des techniques fonctionnelles de programmation dans le domaine du calcul scientifique reste toujours relativement faible. La puissance de l'inférence des types, le polymorphisme et les facilités de construction des données sont reconnus, mais trop souvent l'élégance de l'approche fonctionnelle, son affinité avec la formalisation mathématique et sa compacité de codage sont reléguées au second plan, sacrifiées au nom de l'efficacité. En particulier, les techniques qui profitent de la sémantique paresseuse sont d'habitude considérées trop lentes et gourmandes en mémoire, et de plus «non-naturelles». (Les langages fonctionnels qui ont acquis une certaine reconnaissance industrielle, comme Erlang, SML ou CAML sont stricts).

Mais les ressources humaines sont coûteuses également. Dans notre opinion le fait que les techniques fonctionnelles paresseuses constituent un *outil d'algorithmisation* très puissant et économique, et qu'il est possible d'exploiter la sémantique non-stricte de manière non-triviale, peut favoriser l'usage des langages fonctionnels par les physiciens, ingénieurs, etc. Cependant, il va de soi que les compositions des expressions différées *sont* onéreuses. Ceci peut rester acceptable dans le cadre de la compilation des programmes courts, où le nombre d'opérations du parseur est proportionnel à la longueur du programme. Pour les programmes itératifs, par exemple pour la solution des équations différentielles, les *thunks* doivent obligatoirement être réduits le plus tôt possible. Le codage fonctionnel devient moins attractif.

Notre but était plutôt méthodologique, en construisant cette maquette nous voulions montrer et expliquer un style particulier de programmation fonctionnelle dans un contexte scientifique reconnu. Les résultats pratiques nous semblent très prometteurs grâce à la simplicité du codage, et un peu d'exotisme présent dans le modèle de propagation des états contre le flux du temps rend le sujet assez excitant. L'analogie entre notre technique et la propagation des attributs hérités nous semble très importante, et mérite un travail supplémentaire.

Références

[1] L.B. Rall, *Automatic Differentiation – Techniques and Applications*, Springer Lecture Notes in Computer Science, Vol. 120, (1981).

[2] A. Griewank, *On Automatic Differentiation*. Éd. M. Iri and K. Tanabe, *Mathematical Programming : Recent Developments and Applications*, Kluwer, (1989), pp 83–108.

[3] D. Juedes, *A taxonomy of automatic differentiation tools*. Éd. A. Griewank and G. F. Corliss, *Automatic Differentiation of Algorithms : Theory, Implementation, and Application*, SIAM, Philadelphia, Penn., (1991), pp 315–329.

[4] A. Griewank, D. Juedes H. Mitev, J. Utke, O. Vogel, A. Walther, *ADOL-C : A Package for the Automatic Differentiation of Algorithms Written in C/C++*, ACM TOMS, **22**(2) (1996), pp. 131–167, Alg. 755.

[5] Site Web de Argonne National Laboratory (USA), consacré aux techniques de la différentiation algorithmique **`www-unix.mcs.anl.gov/autodiff`**.

[6] Graham Ronald, Knuth Donald, Patashnik Oren, *Concrete Mathematics*, Addison-Wesley, Reading, MA, (1989).

[7] Jerzy Karczmarczuk, *Functional Differentiation of Computer Programs*, Journal of Higher Order and Symbolic Computing – publication en cours. Voir aussi : Proceedings, III ACM SIGPLAN International Conference on Functional Programming, Baltimore, (1998), pp. 195–203.

[8] Jerzy Karczmarczuk, *Functional Coding of Differential Forms*, I-st Scottish Workshop on Functional Programming, Stirling, Septembre 1999.

[9] R. Giering, T. Kaminski, *Recipes for Adjoint Code Construction*, ACM Trans. On Math. Software, **24(4)**, (1998), pp. 437–474.

[10] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software, **1**, (1992), pp. 35–54.

[11] P. Hovland, C. H. Bischof, D. Spiegelman, M. Casella, *Efficient Derivative Codes through Automatic Differentiation and Interface Contraction : an Application in Biostatistics*, Mathematics and Computer Science Division, Argonne National Laboratory, Preprint MCS–P491–0195, (1995).

[12] P. Wadler, *The Essence of Functional programming*, 19'th Symposium on Principles of programming Languages, Santa Fe, (1992).

[13] R.S. Bird, *Using circular programs to eliminate multiple traversals of data*, Acta Informatica **21**(4), pp. 239–250, (1984).

[14] T. Johnsson, *Attribute Grammars as a Functional Programming Paradigm*, Conference on Functional programming Languages and Computer Architecture, Portland, Proceedings : Springer LNCS 274, pp. 154–173, (1987).

[15] P. Jansen, H. Munk, et L. Augusteijn, *An introduction to Elegant*, Documentation, Philips Research Laboratories, Eindhoven, Pays Bas, (1997). Site Web :
`www.research.philips.com/generalinfo/special/elegant/elegant.html`.