Programmes à voyager dans le temps

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

Résumé

Nous présentons une implémentation de l'algorithme de différentiation automatique inverse (adjointe), qui nécessite une logique assez tordue : un programme numérique procède jusqu'à la fin, ensuite il renverse le flux de calcul, et les dérivées du résultat par rapport aux variables d'entrée sont calculées par un programme dérivé, de la fin jusqu'au début. Nous présentons un modèle qui effectue ce parcours double en une passe, comme si nous pouvions opérer simultanément avec deux flux antithétiques de dépendances, dont l'un se propage du futur vers le passé. Ceci est une méthodologie de programmation légitime, et dont d'autres instanciations on trouve dans la théorie de compilation, dans la modélisation des réseaux neuronaux et dans la robotique et l'animation. Elle jette une intéressante lumière sur le possible traitement du temps en Informatique.

1 Introduction

Toute modélisation d'un système physique, biologique ou social, demande l'instauration d'une couche d'abstraction. On peut simplifier tout, mais cette couche doit être relativement fidèle vis-à-vis notre

compréhension du sujet, avec ses relations avec le monde. Il faut respecter quelques *universalia*, et un de ces concepts respectés universellement c'est la **causalité**, liée au *consecutio temporis*. Le temps passe, l'état du programme (les valeurs des variables etc.) est déterminé par les états précédents. Cependant, dès le début des paradigmes actuels de structuration des programmes, on veut se libérer des contraintes temporelles, de la « flèche du temps ». Les raisons sont rarement philosophiques, souvent strictement pragmatiques :

– Si on représente le système de manière *statique*, sans le temps en tant que catégorie ontologique déterminant la « réalité des calculs », mais seulement comme un paramètre, il est plus facile de raisonner sur le programme, on l'analyse avec plus de précision.
– Plusieurs algorithmes (p. ex. en Intelligence Artificielle) ont un goût relationnel, statique, assez visible. On établie une relation – peut-être asymétrique, mais aussi atemporelle – entre les entrées et les sorties du programme. Une telle approche a favorisé le développement des langages fonctionnels et logiques. Plusieurs algorithmes de recherche dans ce contexte fonctionnent comme s'ils pouvaient reculer le temps, « oublier » le changement d'état, et recommencer par une autre stratégie. Ceci constitue l'essence de la technique du *backtracking* en Prolog. L'utilisateur peut avoir l'impression que le temps linéaire bifurque, se transforme en une arborescence multi-branche, et on peut suivre une, en retournant d'une autre. Les « mauvaises » sont oubliées, et le programme fournit la première bonne solution trouvée, ou – si on le veut – toutes.
– On peut trouver une situation, quand on a besoin du « temps inversé », ou plus précisément, des flux de dépendances entre les données et entre les états du système, qui soient *antithétiques*. **Ceci est le sujet principal de ce travail**. Nous démontrerons sur un exemple pratique, appartenant au domaine de programmation scientifique comment implémenter de telles entités.

Bref, il est souhaitable de pouvoir « sortir du temps physique », et de le « regarder de l'extérieur ». Ceci peut être considéré comme une démarche philosophique et méthodologique, voir p. ex. [1], mais pour nous ne sera qu'une technique de programmation.

1.1 Backtracking, et évaluation différée

Exemple. Un algorithme combinatoire peut contenir un petit module : le choix d'un objet parmi les éléments d'une liste. Par exemple, le prédicat Prolog **`choix([a,b,c,d],X)`** doit assigner à la variable **`X`** ***toutes*** les valeurs possibles de choix d'un des éléments de la liste constituant le premier paramètre : **`X=a`**, **`X=b`**, etc. La définition du prédicat **`choix(Liste,Résultat)`** qui établit une *relation* entre le premier et le second paramètre, consiste à énumérer les possibilités :

```
choix([X|Reste],X).      Déstructuration du 1er paramètre.
choix([Y|Reste],X):-choix(Reste,X).
    Ignore Y, cherche X dans le Reste
```

ou : soit on prend le premier élément de la liste, soit un des éléments restants, récursivement. Si on *refuse d'accepter* la première proposition, le programme effectue automatiquement le « backtracking », le « retour en arrière temporel », et en propose une autre. Mais Prolog est un langage assez particulier. Sur le plan sémantique – puisque en fin de comptes toutes les branches sont sérialisées, exécutées une après le backtrack de l'autre, les solutions multiples sont ***équivalents à une liste qui les contient toutes***.

Ceci est l'approche utilisé dans les langages plus classiques, dont la sémantique se réduit à la transformation de données par l'application successive des fonctions. Quelques uns de ces langages, comme Haskell [2] profitent du paradigme d'*évaluation paresseuse* (ou différée) qui peut être intuitivement réduite à l'exigence que l'ordinateur n'évalue pas une expression pour en déduire sa valeur que si cette

valeur est nécessaire dans l'immédiat. Sinon, l'expression reste « latente », sous forme de code qui attend d'être exécuté pour qu'il devienne une valeur. Grâce à ce dispositif on peut définir des séquences **infinies** de données, par exemple la liste **`[0,1,2,3,...]`**:

```
nombres = arithseq 0   where
 arithseq n = n : arithseq (n+1)
```

(Ici l'opérateur **`(:)`** ajoute l'argument à gauche à la liste à droite, **`5:[1]`** est égal à **`[5,1]`**. Il est équivalent à **`|`** en Prolog). On voit que la définition de la fonction **`arithseq`** est récursive sans clause terminale, elle « ne s'arrête jamais ». Dans un langage sans évaluation paresseuse ceci déclencherait une exception du débordement de mémoire, mais en Haskell si on a besoin de, disons, 10 éléments, les autres ne sont pas instanciés. On peut aisément construire d'autres exemples. La construction suivante

```
x = 1:y
y = 2:x
```

assigne à **`x`** la liste **`[1,2,1,2,1,...]`**. Grâce à l'évaluation paresseuse on peut utiliser une donnée qui n'a pas encore été *complètement* définie (dans l'exemple ci-dessus, elle n'en sera jamais, puisque elle représente un objet explicitement infini…).

Ainsi, une structure de données apparemment statique, représente *un processus*, une activité qui intuitivement se déroule dans le temps. La génération de la liste est suspendue, et reprise seulement quand on redémarre l'observation des éléments suivants. Si le programme se rend compte que l'utilisateur a besoin de l'onzième élément de la liste, *un* pas de la génération récursive est exécuté, et « le temps s'arrête » de nouveau.

Cette technique peut être utilisée dans un contexte explicitement atemporel, appartenant aux mathématiques standard ; elle permet de traiter des séries infinies et autres structures similaires, mais les structures paresseuses peuvent également représenter « simultanément »

des trajectoires *complètes* résultant de la solution d'une équation différentielle [3], objets qui typiquement sont des résultats des processus itératifs, engendrés un par un. On regarde le système d'un point « en dehors du temps ». Le « présent » n'existe pas, nous pouvons choisir le temps d'observation et d'instancier l'état du système à volonté. Bien sûr, aucun miracle n'a lieu, et notre programme doit dûment et causalement générer tous les états qui précèdent l'état observé.

L'implémentation des structures paresseuses est connue depuis des années '70. Il suffit simplement de compiler un appel fonctionnel, disons, $y \leftarrow f(e)$ **non pas** comme : 1. exécuter le code qui évalue e ; 2. appliquer le code qui réalise f à la *valeur* de e, mais :

1. Lancer la fonction f tout de suite. Si à cause d'une condition e n'est pas utilisé, l'argument ne sera jamais évalué.
2. Quand le code de f a besoin de e, son code est exécuté (et la valeur mémorisé, mais ce n'est qu'une optimisation). La valeur de l'argument est utilisée par la fonction.

Cette possibilité de contrôler le temps d'évaluation par *l'usage, non pas par la définition* des données dans le programme sera encore plus utile dans le contexte suivant. Nous n'aurons pas besoin de structures paresseuses pour simuler un processus infini, cependant le concept de l'évaluation différée sera toujours fondamental.

1.2 Flux antithétiques : quelques exemples

Les interactions fondamentales en physique sont neutres par rapport à l'inversement du temps. Si une théorie prévoit l'existence des phénomènes acausaux, la propagation des particules quantiques contre la flèche du temps, on peut les réinterpréter comme antiparticules qui se comportent bien ; l'émission et l'absorption s'échangent réciproquement. Les détails sont tordus (p. ex., l'énergie d'une particule qui voyage à contre-temps est négative), mais pour les systèmes élémen-

taires la flèche du temps n'est pas significative.

Nous pouvons dessiner le diagramme de Feynman à droite et dire qu'il représente *un* électron (ligne en bas à gauche) qui a décidé un jour de reculer dans le temps (ligne en haut). Il le fait en émettant un photon (ligne sinueuse). Mais pour un observateur normal, c'est un processus d'annihilation d'un électron et d'un positron (en parfait accord avec la flèche du temps, et avec l'énergie positive) avec l'émission du photon. Un tel exemple est impossible dans la physique macroscopique, où les objets peuvent avoir la mémoire.

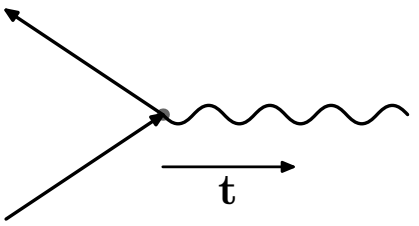

FIG. 1 – *Réaction d'annihilation $e^- e^+ \rightarrow$ photon*

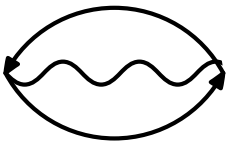

FIG. 2 – *« Boucle temporelle » virtuelle*

Mieux encore, on peut trouver des processus virtuels comme celui à gauche. Un électron venant du futur absorbe un photon, et, bouleversé, commence à se comporter normalement. Un peu plus tard il change d'avis, livre un photon anti-temporel, et se transforme en une particule anti-temporelle également. Ces deux particules ferment la boucle temporelle. Ceci constitue ce que les physiciens appellent les fluctuations du vide, et personne ne trouve cela paradoxal, c'est la création, et anéantissement des particules virtuelles, « normales » vis–à-vis la flèche du temps.

Quand le système devient complexe, quand on introduit des notions d'entropie et d'ireversibilité, le temps devient une catégorie critique. Nous nous sommes habitués à cela. Pour un informaticien l'entropie est l'essence physique de l'information structurelle, donc l'évolution causale des états d'un programme nous semble un paradigme sain, pratiquement le seul possible. Mais le pire pour l'intuition n'est pas le temps qui « marche à l'envers », mais la possibilité que les deux flux temporels, le normal et l'inversé, coexistent. En physique micro-

scopique ceci est conceptuellement réalisable grâce à la réversibilité.

Si l'information peut être crée ou effacée, une attitude cavalière vis-à-vis le temps produit un chaos méthodologique, dont l'issue ne peut être que philosophique ou artistique, voir le livre de science-fiction de Philip Dick [4], où l'auteur suggère, que ressusciter les morts soit une tâche plus facile qu'effacer (« anti-écrire ») un livre… Cependant, l'existence conceptuelle de ces flots antithétiques dans l'informatique moderne est une réalité.

– La compilation d'un programme-source en code exécutable contient la phase du *parsing*, ou l'analyse syntaxique. Une des stratégies du parsing consiste à prendre un par un les symboles atomiques (mots) d'une phrase, et d'assembler d'abord les symboles élémentaires en structures internes représentant les sous-expressions, ensuite les expressions plus grandes, etc. jusqu'à l'objet final : le programme entier. En procédant ainsi on accumule (synthétise) les propriétés (attributs) des entités analysées. Imaginons que notre langage décrit des objets graphiques d'une interface utilisateur. La taille d'un « widget » composite est sûrement déduite, synthétisée à partir des tailles des composantes.

  Mais, souvent on exploite la notion d'*attributs hérités*, contextuels, qui sont « hérités » par les feuilles depuis la racine. On voit clairement que la *position* d'un widget sur la surface du travail dépend de la position de la « boîte englobante », du « widget-père ». Donc, une telle programmation qui assemble et qui affiche les widgets en une passe doit « regarder vers l'avenir ».

– Dans la robotique, ou dans la synthèse des animations, on compose avec les objectifs dictés par le scénario, et la causalité des mouvements des acteurs. En sachant où l'effecteur (la main d'un personnage, etc.) doit arriver, on reconstruit « anti-temporellement » les forces agissant sur les articulations, qui engendraient le mouvement souhaité. Cette technique s'appelle la *cinématique inverse*,

assez classique, mais pas si facile à programmer [5, 6]. Bien sûr, dans la pratique on ne parle pas de « flots d'instructions venant du futur », on calcule tout simplement des inverses des matrices de Jacobi, etc., mais l'image est assez suggestive.

– Dans les modèles d'apprentissage des réseaux neuronaux on trouve des phénomènes du même genre [7, 8]. Une vision du processus d'apprentissage correspond à un flot *des erreurs* contre la flèche du temps. Ces exemples sont proches de la technique décrite ici.

2 Programmation fonctionnelle et états

La programmation des ordinateurs peut être *purement fonctionnelle*, une réalisation du calcul λ de Church. Toutefois, la programmation *réelle* ne se satisfait pas d'opérer avec des structures mathématiques transformées par des fonctions. Pour simuler les systèmes complexes, on a besoin de la notion d'état, de quelque « chose qui change », quand le programme s'exécute. Par exemple, on peut ajouter un compteur qui est incrémenté lors de chaque opération. Dans le monde fonctionnel on procède de manière suivante : au lieu d'opérer avec les valeurs, disons x, auxquelles on applique une fonction : $x \rightarrow f(x)$, on construit des *paires* (x,s), où s est une structure de données qui symbolise l'état. Si s est le compteur, l'application (convenablement généralisée) de la fonction f produit l'objet $(f(x),s+1)$.

La vérité est un peu plus complexe… On ne peut proliférer les compteurs partout, le programme doit assurer que l'état du système existe en un seul exemplaire ! En fait, la valeur x sera « liftée » vers une entité fonctionnelle, $\lambda s \rightarrow (x,s)$ (en Haskell : `\s->(x,s)`). C'est une fonction *qui agit sur l'état* et ici – seulement le répertorie, sans changement. Par contre, une fonction primitive f agissant sur x provoque aussi le changement d'état, elle prend donc deux arguments ; en général toute expression devient une fonction de l'état

initial, et qui retourne une valeur et l'état final, éventuellement modifié. En Haskell nous définissons un opérateur **`(>=>)`** tel, que **`f >=> m`** exécute l'expression **`m`** sur l'état initial, produisant l'état intermédiaire. La fonction **`f`** agit sur la valeur et cet état, et modifie les deux. Voici la définition de cette opération, assez courte :

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

Toutes les opérations sont chaînées séquentiellement, et le programme devient une énorme fonction, ce qui semble bizarre, mais le modèle est parfaitement utilisable. Dans la section (4) il sera modifié de manière à rendre le flux d'états antithétique.

3 Différentiation automatique

Tout le monde scientifique a besoin de calculer les dérivées. On peut les calculer analytiquement ou numériquement (*via* des différences finies). La première méthode est pénible, la seconde – inexacte. Il existe cependant une technique « intermédiaire »: aussi exacte que la technique symbolique, mais purement numérique et automatique. ***On demande à ce que les expressions numériques dans le programme soient enrichies par le calcul de leurs dérivées***, ce qui est pratiquement toujours possible automatiquement, le problème est algorithmiquement trivial. La technologie est bien connue, voir p. ex. [9, 10]. La méthode la plus simple, « directe » consiste à augmenter toute expression e sur le domaine de paires : (e,e') où e' est naturellement la dérivée de e. Quel que soit la complexité du programme, toute manipulation se réduit finalement à la combinaison d'opérations arithmétiques simples sur des sous-expressions. Si, pour simplifier le discours, nous traitons une seule « variable » x de différentiation, son « lifting » est égal, bien sûr, à $(x,1)$. Toute constante c (comme 2, 0.5,

π, etc.) devient $(c,0)$. À présent nous pouvons définir les opérations arithmétiques sur de tels objets : $(e,e') + (f,f') = (e + f,e' + f')$, $(e,e') \times (f,f') = (ef,e'f + ef')$, $\sin(e,e') = (\sin(e), \cos(e) \times e')$, $\sqrt{(e,e')} = (\sqrt{e},e'/2\sqrt{e})$ etc. En général, pour une fonction arbitraire f: $f(e,e') = (f(e),e' \cdot f'(e))$. Une affectation dans le programme : $g \leftarrow f(e_1,e_2,\ldots,e_k)$ engendre l'instruction

$$\frac{dg}{dx} = \frac{\partial f}{\partial e_1}\frac{de_1}{dx} + \frac{\partial f}{\partial e_2}\frac{de_2}{dx} + \cdots + \frac{\partial f}{\partial e_k}\frac{de_k}{dx} \tag{1}$$

et, sachant que les sous-expressions e_i ont déjà été construites avant, si leurs dérivées sont connues également et la fonction f est connue, toute expression g est algorithmiquement calculable avec sa dérivée. La généralisation aux plusieurs variables est simple, il faut prévoir pour chaque variable un vecteur – gradient, de dimension égale au nombre de variables.

3.1 Différentiation automatique inverse

Dans quelques circonstances, surtout quand on a besoin d'un seul résultat qui dépend de plusieurs variables, par exemple : la puissance d'un réacteur, ou la température d'une couche atmosphérique qui dépendent de centaines, voire plus, de paramètres – on procède différemment, pour l'efficacité on n'introduit pas des gradients pour chaque variable dans le programme. Par contre, pour chaque variable x, on introduit son ***adjoint*** $\overline{x}$. L'adjoint de e est spécifié formellement comme $\overline{e} = \partial z/\partial e$, où z dénote le résultat final du programme.

L'enchaînement des dérivées se fait « à l'envers », ce qui sera expliqué ci-dessous. La technique n'est pas facile à implémenter, mais elle est bien maîtrisée aussi, voir p. ex. [11]. Prenons à titre d'exemple le programme suivant :

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

où x est la variable indépendante, et z est le résultat. Tandis que la méthode directe commence par identifier $dy/dx = \cos(x)$, et ensuite $dz/dx = \partial z/\partial x + \partial z/\partial y \times dy/dx$, la méthode inverse introduit d'abord pour toute variable e son adjoint. Il est évident que le résultat du programme augmenté, la valeur de la dérivée dz/dx est la valeur de l'adjoint de la variable indépendante x.

Pour toute affectation $g \leftarrow f(e_1,e_2,\ldots,e_k)$ dans le programme, on ajoute (automatiquement, mais nous verrons que ceci pose des problèmes…) les instructions adjointes, qui ***modifient les adjoints des variables à droite***:

$$\overline{e}_j \leftarrow \overline{e}_j + \overline{g}\frac{\partial f}{\partial e_j} \tag{3}$$

Mathématiquement c'est que l'on appelle une opération *contravariante*, soulignons le fait que c'est **l'usage**, non pas la définition d'un expression e qui définit son adjoint. La dérivation formelle de cette formule peut être trouvée dans [12]. Mais, comment ceci est possible? Quand dans le programme on affecte g pour la première fois, cette variable n'a jamais figuré auparavant dans une autre expression, seulement à partir de ce moment-là que nous avons le droit de l'utiliser. Cependant la valeur de son adjoint est nécessaire pour calculer $\overline{e}_j$. D'autre part, puisque $\overline{g} = \partial z/\partial g$, il faut connaître g pour calculer le résultat final, et donc les adjoints des variables. La situation semble inextricable. La réponse est : faire deux passes sur le programme. D'abord calculer tout dans le programme d'origine, et ensuite renverser le flux de contrôle et exécuter toutes les instructions associées (adjointes) dès la fin jusqu'au début.

Dans notre exemple, nous aurons

$$\begin{array}{rll} z = y^2 - x/y; & \text{donne} & \overline{x} \leftarrow \overline{x} + \overline{z}(-1/y); \\ & & \overline{y} \leftarrow \overline{y} + \overline{z}(2y + x/y^2); \\ y = \sin(x); & \text{donne} & \overline{x} \leftarrow \overline{x} + \overline{y}\cos(x)\,. \end{array} \tag{4}$$

Finalement $\overline{x} = -1/\sin(x) + \cos(x)\left(2\sin(x) + x/\sin(x)^2\right)$ est la valeur de dz/dx. Dans les paquetages « industriels » on effectue ces deux passes, en prévoyant une zone de stockage (une « bande ») pour les instructions adjointes pendant la première phase du programme, quand on ne calcule que les valeurs principales. Nous avons voulu implémenter cet algorithme en ***une passe***, à l'aide de puissantes techniques sémantiques de la programmation fonctionnelle.

4 Modèle fonctionnel de la DAI

Nous allons – comme il a été promis – « pervertir » la gestion d'état dans le cadre de la programmation fonctionnelle. Dans notre modèle, [12], qui est purement fonctionnel, pour chaque expression dans le programme nous introduisons son adjoint ***considéré comme une partie de l'« état » du système***. Cependant, afin d'obtenir le comportement « anti-temporel », comme décrit dans la section précédente, nous allons modifier de manière plutôt drastiquee l'enchaînement des états à l'aide d'un nouveau opérateur **`(>=>)`**, différent de l'enchaînement traditionnel. L'idée est basée sur un travail de Wadler [13]. Il faut imaginer que l'expression composite **`f >=> m`** soit une fonction qui agit sur l'état final, non pas initial. Dans cet état final la fonction f s'applique à une valeur obtenue par l'évaluation de m.

Mais, en agissant ainsi, la fonction f produit accessoirement l'état intermédiaire, et c'est à cet état intermédiaire qu'il faut appliquer m pour récupérer sa valeur. L'expression m produit l'état initial.

Ceci ne peut vraiment être intuitivement compris car l'ordre temporel semble être violé, mais on *peut l'implémenter* en Haskell de manière naturelle :

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

Observons, que m *doit* être exécuté afin de récupérer sa valeur principale x, pour que la fonction f puisse être appelée. Mais il faut appeler cette fonction « avant », pour avoir accès à **`s_mid`**, indispensable pour l'exécution de m. Nous avons deux flots antithétiques de dépendances entre les données.

Comparons ceci avec la section (1.1), le programme qui construit la liste **`[1,2,1,2,1,...]`** avec deux variables x et y qui se référencent réciproquement. La programmation paresseuse permet l'application de tels astuces. La fonction f peut commencer son exécution sans connaître la valeur de ses paramètres. Il nous reste de spécifier « l'état » de façon à permettre au programme de calculer les adjoints durant la seconde, antithétique phase, qui dans la réalité se déroule en parallèle avec la première. Le rapport entre les états et les adjoints est simple. Au moment du démarrage du calcul, quand on évalue, disons, x, l'état initial est constitué de $\overline{x}$. À la fin, en construisant le résultat z, on considère que l'état final soit $\overline{z}$. Le secret de la faisabilité de notre « programme à voyager dans le temps » repose sur les constatations suivantes :

– L'état initial $\overline{x}$ **ne peut** être connu au début des calculs, mais ceci n'est pas grave, car on n'en aura besoin qu'à la fin… Il peut rester « latent ». En fait, l'état initial « sera là », mais différé, sous forme d'un objet paresseux, non-évalué. C'est une « promesse » de livrer la valeur actuelle, quand toutes les dépendances entre variables seront résolues.

– Par contre, l'état final $\overline{z} = \partial z/\partial z \equiv 1$ est trivialement connu à l'avance ! Nous pouvons l'utiliser à n'importe quel moment.

4.1 Adjoints et états

On traitera – seulement pour simplicité – le cas 1-dimensionnel (une variable indépendante). Les expressions (ses valeurs) sont numériques,

appartiennent, disons, au type **a**, où **a** est typiquement **Double**, le type de nombres réels. Les adjoints – naturellement – aussi. Le type lifté, représentant les expressions-transformateurs d'états aura le type **Ld (a->(a,a))**, où la balise **Ld** est là uniquement pour guider les yeux, ce type *est une fonction*. On n'utilisera aucun opérateur **>=>** explicit, mais tout sera réalisé par des opérations arithmétiques standard, utilisables par un utilisateur quelconque, qui n'a même pas besoin de savoir que ces opérations prennent en charge les dépendances croisées entre données.

Les constantes numériques c et les variables x dans un programme augmenté doivent être « liftées » avec des fonctions ci-dessous :

```
lCnst c = Ld (\n->(c,0))    -- Une constante est triviale
lDvar x = Ld (\n->(x,n))    -- L'état est multiplié par 1
```

Voici quelques opérateurs arithmétiques qui réalisent les opérations sur les expressions liftées :

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

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))
```

On peut également prendre une fonction quelconque (codée séparément) **f**, et si sa ***dérivée formelle*** **f'** est connue (est également codée ; par exemple **f** peut être la fonction sinus hyperbolique, alors la dérivée sera le cosinus hyperbolique), alors on peut prévoir un « lifting »

générique, universel :

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

Les constructions ont un goût artificiel et demandent une bonne expérience, mais on peut les enseigner en deux heures aux étudiants en physique, etc. tandis que les détails de l'approche traditionnelle occupent quelques dizaines de pages. Donc, même si l'efficacité de la méthode proposée ici n'est pas idéale, sur le plan méthodologique cette « sortie du temps », et une vision non-orthodoxe de structures de contrôle semble très promettante.

On écrit un programme « presque normal », avec des variables, constantes etc., appartenant au domaine de transformateurs d'états. Le programme s'exécute normalement, mais à la fin on constate qu'il retourne un transformateur d'états, une fonction qui attend *encore* d'être appliquée. On l'applique à 1, à l'adjoint du résultat final : dz/dz, et on récupère $(z,\overline{x})$.

5 Conclusions

Puisque dans le modèle fonctionnel, par excellence statique, le temps n'est pas une catégorie ontologique, nous pouvons faire des expériences assez étonnantes avec la causalité comprise comme une *relation logique*. Le temps qui subit des bifurcations, le temps qui va à l'envers, etc., tout ceci devient techniquement modélisable. Le but de ces modèles est de *faciliter la programmation*. Un chercheur ou ingénieur placé devant un problème logique complexe, surtout dans le domaine de simulation, constate plusieurs embrouillements : des fragments d'un système influencent la totalité, mais le global détermine le comportement des fragments. Le système se comporte téléologiquement, et anticipe ses actions selon un objectif *a priori*, même si physiquement cela est douteux.

Pour organiser un programme de manière propre nous avons besoin de rompre la relation entre le temps physique et le temps simulé, tout en respectant la causalité. Ainsi nous pourrons programmer la consommation d'un « fil d'Ariane » avant que celui-ci sera placé, mais s'assurant *formellement* qu'effectivement il sera déroulé comme il faut plus tard. Ce « plus tard », grâce aux protocoles de la programmation paresseuse devient beaucoup plus contrôlable. Ces techniques sont difficiles… Cependant, au lieu de décrire en termes généraux la méthodologie exposée, nous avons préféré de donner un exemple *très concret*, et utile dans la programmation scientifique. La combinaison de détails techniques et d'une vision non-classique du contrôle a produit un certain abus de constructions par analogie *dans la présentation*. Le lecteur voudra nous en excuser.

Références

[1] Huw Price, *Time's Arrow and Archimedes' Point*, Oxford Univ. Press, (1996).

[2] Site web : **`http://www.haskell.org`** contient les distributions des compilateurs et toute la documentation.

[3] Jerzy Karczmarczuk, *Traitement paresseux et optimisation des suites numériques*, JFLA'2000, Mt St. Michel, pp. 17–30.

[4] Philip K. Dick, *Counter Clock World*, Berkley P.B., (1967). Voir aussi : *The World Jones Made*, où le héros, qui vit simultanément en deux tranches du temps, est obligé de synchroniser sa connaissance du futur *inéluctable*, avec les dépendances entre les processus, qu'il engendre lui-même.

[5] D.E. Whithney, *Resolved Motion Rate Control of Manipulators and Human Prostheses*, IEEE Trans. on Man-Machine Systems, **0**:2, pp. 47-53, **(1969)**.

[6] C. Welman, *Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulations*, Simon Fraser Univ. **(1993)**.

[7] E.A. Wan, F. Beaufays, *Diagrammatic Derivation of gradient Algorithms for Neural Networks*, Neural Computation, (1994), ou autres articles de Eric Wan et Françoise Beaufays.

[8] P. Werbos, *Backpropagation Through Time: What It Does and How to Do It*, Proceedings IEEE, special issue on neural networks **2**, pp. 1550–1560, (1990).

[9] L.B. Rall, *Automatic Differentiation – Techniques and Applications*, Springer Lecture Notes in Comp. Sci., Vol. **120**, (1981).

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

[11] R. Giering, T. Kaminski, *Recipes for Adjoint Code Construction*, ACM TOMS, **24(4)**, (1998), pp. 437–474.

[12] J. Karczmarczuk, *Calcul des adjoints et programmation paresseuse*, Journées JFLA ’2001, Pontarlier, (2001), pp. 145-156.

[13] P. Wadler, *The Essence of Functional programming*, 19’th Symp. on Princ. of Prog. Lang., Santa Fe, (1992).