Université Denis Diderot
Master 1 MPRI
2006 - 2007

# Travail de recherche encadré : Le calcul incrémental monadique

Edlira Nano

Ce texte est la propriété de l'auteur. Il est régi par les termes de la Licence de Libre Diffusion des Documents, version 1, consultable en ligne sur `http://pauillac.inria.fr/%7Elang/licence/v1/lldd.html`.

# 1 Introduction

Le but de ce travail de recherche encadré est la lecture de l'article *Monads for incremental computing* de M. Carlsson (voir [Car]) ainsi que la réalisation d'une petite librairie en *OCaml* en se basant sur la librairie en *Haskell* présentée dans cet article (voir [Lib]).

## 1.1 Le calcul incrémental

**Définition 1** *Soient :*

- *$f$ un programme et $x$ une donnée en entrée de $f$,*
- *$y$ un changement dans la valeur de $x$ et op une opération qui combine $x$ et $y$ pour produire une nouvelle donnée en entrée de $f$ de valeur $x$ $(op)$ $y$,*
- *$r = f(x)$ le résultat de l'exécution de $f$ sur $x$,*
- *$f'$ un programme qui calcule $f(x\ (op)\ y)$ tel que :*
  1. *le calcul de $f'$ est plus rapide que celui de f pour tout $x$ et $y$,*
  2. *$f'$ utilise $r$.*

*Alors $f'$ est appelé version incrémentale de $f$ sous op.*

On parle alors de calcul incrémental. Parfois, des informations supplémentaires autres que $r$ sont nécessaires à $f'$ pour un calcul incrémental efficace.

Prenons un exemple simple : un programme f calcule la somme des éléments successifs d'une liste $l$ de taille $n$ selon l'algorithme de complexité $\mathcal{O}(n^2)$ suivant :

```
pour i allant de 1 à n
   s = s + l[i]
finpour
retourner s
```

Ainsi :

$$l = [2; 3; 1; 5; 2] \xrightarrow{f} 2 + 3 + 1 + 5 + 2 = 13 = s$$

Lorsque la liste $l$ en entrée est modifiée, par exemple son troisième élément vaut 4 au lieu de 1, pour calculer la nouvelle somme notre version incrémentale $f'$ de $f$ soustrait l'élément sortant 1 de la somme précédemment calculée et rajoute la nouvelle valeur 4 :

$$l = [2; 3; 4; 5; 2] \xrightarrow{f'} 13 - 1 + 4 = 16 = s$$

Le calcul incrémental permet donc, par l'utilisation du résultat du calcul précédent, une mise à jour du résultat bien plus rapide que la réévaluation complète.

## 1.2 Survol des articles étudiés

Le calcul incrémental est utilisé pour l'optimisation des compilateurs, les systèmes interactifs, etc. De nombreuses techniques de calcul incrémental ont été développées. Une de ces techniques est celle utilisant des graphes de dépendance dynamique appelés DDG (pour Dynamic Dependence Graph) et l'algorithme appelé *change propagation*.

Cette technique est présentée dans l'article *Adaptive functional programming* (voir [Acar]). Les auteurs parlent ici de programmation fonctionnelle adaptative. Lorsqu'un programme adaptatif s'exécute, le système sous-jacent représente les données et leurs dépendances d'exécution via un DDG. Lorsque l'entrée du programme change, l'algorithme *change propagation* met à jour le DDG et la sortie du programme en propageant les changements dans ce graphe et en ré-exécutant le code seulement lorsque cela est nécessaire. Les auteurs y définissent trois primitives pour rendre adaptatif tout programme fonctionnel avec appel par valeurs, puis deux opérations : changement de l'entrée et propagation. L'implémentation d'une telle librairie pour ML y est décrite brièvement. Enfin, pour assurer l'usage correct de la librairie, les auteurs construisent un langage purement fonctionnel adaptatif qui donne cette assurance.

Dans *Monads for incremental computation*, l'auteur présente une approche monadique de la technique précédemment décrite, pour des langages purement fonctionnels comme Haskell. La librairie en ML précédente y est traduite selon cette approche en Haskell. L'article montre comment l'utilisation des monades assure l'usage correct de la librairie (sans recourir à des extensions comme dans le cas précédent), mais aussi offre des opportunités d'optimisation du temps du calcul incrémental. Cette librairie est disponible pour téléchargement sur [Lib].

# 2 Librairie adaptative

## 2.1 Présentation

Voici la signature de la librairie adaptative pour ML donnée dans [Acar] :

```
signature ADAPTIVE =
 sig
  type 'a mod
  type 'a dest
  type changeable
  val mod: ('a * 'a -> bool) ->
           ('a dest -> changeable) -> 'a mod
  val read: 'a mod * ('a -> changeable) -> changeable
  val write: 'a dest * 'a -> changeable
  val change: 'a mod * 'a -> unit
  val propagate: unit -> unit
  val init: unit -> unit
 end
```

Une variable modifiable (de type *'a mod*) est crée par *mod*, une valeur y est inscrite par *write*, cette valeur est lue par *read* et changée par *change* et *propagate*. *init* initialise la librairie. Le type *changeable* représente les expressions qui définissent ou mettent à jour une modifiable en se servant (en lisant) d'autres modifiables, dans un style de programmation par continuations appelé CPS (pour Continuation-Passing Style) : elles ne retournent pas une valeur mais prennent un paramètre destination *'a dest* dans lequel écrivent la valeur.

L'usage correct de la librairie demande que :

1. les valeurs des modifiables ne puissent pas être lues directement, mais uniquement lors de la définition d'autres modifiables
2. chaque expression changeable utilise sa destination exactement une fois, lors d'une opération write.

La propriété (1) est assurée par la définition même du type de *read*. C'est pour assurer la propriété (2) que les auteurs présentent dans [Acar] un langage fonctionnel étendu. L'approche monadique dans [Car] assure à elle seule cette propriété.

La fonction *mod* prend deux paramètres : une fonction de comparaison et un initialiseur. La fonction de comparaison est utilisée par l'algorithme *change propagation* pour comparer l'ancienne et la nouvelle valeur de la modifiable, l'initialiseur est celui qui écrit la valeur de la modifiable. La fonction

*read* prend en paramètres une modifiable et un *reader*, fonction de corps changeable qui est appliquée à la modifiable. Illustrons avec un exemple :

```
let val m  = mod (op=) (fn d => write(d, 1))
    val n  = mod (op=) (fn d => write(d, 2))
    val mn = mod (op=) (fn d =>
                        read (m, fn v =>
                              read (n, fn w =>
                                    write (d, v * w))))
in change (m, 3);
change (n, 7);
propagate()
end
```

La modifiable $m$ est ici définie avec la valeur 1, $n$ avec la valeur 2, $mn$ est définie ayant la valeur de $m$ multipliée par celle de $n$. L'opération de comparaison (op=) vérifie si la nouvelle valeur de la modifiable est différente de l'ancienne. Lorsque les valeurs de $m$ et $n$ sont changées, $mn$ doit être recalculée puisque elle utilise les valeurs de $m$ et $n$. Le recalcul est fait en appliquant $fn\ v => read(n, fn\ w => write(d, v * w))$ aux valeurs modifiées de $m$ et $n$. Ainsi $mn$ vaudra après propagation $3 * 7 = 21$.

## 2.2 DDG et time-stamps

Dans l'exemple précédent, si seule la valeur de $m$ était modifiée, il serait inutile de recalculer la valeur de $n$ lors de la propagation. Pour être efficace l'algorithme du *change propagation* doit donc garder une trace des dépendances mutuelles des modifiables, de l'ordre d'évaluation des différents *read* en excluant ceux qui ne doivent pas être réévalués. C'est ce à quoi vont servir les DDG.

Un DDG est un graphe dirigé acyclique crée dynamiquement lors de l'évaluation d'un programme adaptatif. Chaque modifiable y est représentée par un nœud et chaque arc représente un *read*. Lors de l'évaluation d'un *read* la modifiable qui est lue devient le nœud source et la modifiable dans laquelle le *reader* finit par écrire devient le nœud destination. On dit qu'un *read* $r$ est contenu dans un autre *read* $r$' si l'arc de $r$ a été crée durant l'évaluation de $r'$. Ceci défini une hiérarchie entre les *read*, hiérarchie qui est représentée dans *change propagation* par l'utilisation des time-stamps.

Les time-stamps représentent différents "moments" d'un découpage du temps d'exécution de notre programme adaptatif. Chaque nœud et chaque arc du DDG possède son time-stamp, qui correspond au temps de son exécution dans l'ordre séquentiel d'exécution du programme. Le time-stamp d'un

nœud est généré par *mod* après évaluation de l'initialiseur, le time-stamp d'un arc est le time-stamp de la modifiable source du *read* que cet arc représente. On appelle *start time* le time-stamp d'un arc (i.e. le time-stamp de la modifiable source) et *stop time* le time-stamp de la modifiable écrite dans le *read* (i.e. le time-stamp de la modifiable destination). On définit alors le *time-span* d'un arc comme le couple $(start\ time, stop\ time)$. Le time-span définit la hiérarchie des *read* comme suit :

**Définition 2** *Soient $r$ un* read *de time-span $t_r = (start_r, stop_r)$ et $r'$ un* read *de time-span $t_{r'} = (start_{r'}, stop_{r'})$. $r$ est contenu dans $r'$ si et seulement si : $start_{r'} < start_r < stop_r < start_{r'}$*

## 2.3 Algorithme *change propagation*

Voici l'algorithme en pseudo-code :

```
Change propagate
    I : ensemble des modifiables changées
    G = (V, E) : DDG
1 Q := ensemble des arcs sortants de chaque v dans I
2 pour Q non vide faire
3    e:= delete_min(Q)
4    (start, stop) := time-span(e)
5    V := V - {v dans V tq start < time-stamp(v) < stop}
6    E' := {e' dans E tq start < time-stamp(e') < stop}
7    E := E - E'
8    Q := Q - E'
9    v' := apply(reader(e), value(source(e)))
10   if v' # value(dest(e)) then
11      value(dest(e)) = v'
12      Q := Q + arcs_sortants(dest(e))
```

Étant donnés un DDG G et un ensemble I de modifiables dont la valeur a changé (par *change*), l'algorithme du *change propagation* propage ces changements et met ainsi à jour le DDG. On ne réévalue que les *read* affectés par les changements. Conformément au DDG ces *read* correspondent à tous les arcs sortants des modifiables changées (appartenant à I donc). On place ces arcs dans une queue de priorité Q (ligne 1). La priorité Dans Q étant définie selon l'ordre des time-stamp des arcs, on ne risque pas d'évaluer un read avant d'évaluer les reads contenus dans celui-ci (ces derniers ayant un time-stamp plus petit). Après la réévaluation du reader (ligne 9) l'algorithme

vérifie si la valeur de la modifiable destination a changé (ligne 10) en utilisant la fonction de comparaison passée en argument à *mod*. Si cette valeur a changé tous les *read* de cette modifiable doivent être réévalués, on ajoute donc les arcs sortants de la modifiable destination à la queue (ligne 12) après avoir écrite cette nouvelle valeur dans la modifiable (ligne 11).

On remarque de plus que chaque itération dans l'algorithme met à jour un arc $e$ en réévaluant son reader seulement après avoir enlevé de la queue de priorité et du DDG tous les nœuds et tous les arcs contenus dans $e$ (lignes 5 à 8). Pour comprendre la raison de ceci prenons un fragment d'un hypothétique programme :

```
let val m     = mod (op=) (fn d => write(d, 1))
    val n     = mod (op=) (fn d => write(d, 1))
    val mdivn = mod (op=) (fn d =>
                           read (n, fn n =>
                             if n = 0 then write (d, -1)
                             else read (m, fn m =>
                                        write (d, m / n))))
```

Lors de la création de *mdivn* la librairie ajoute les deux arcs correspondant aux *read* de $m$ et de $n$ de destination *mdivn*. L'arc du *read(m)* est de time-stamp plus petit car ce *read* est contenu dans le *read(n)*. Supposons maintenant que la valeur de $n$ change pour devenir 0 et celle de $m$ devienne 4. La librairie va insérer l'arc partant de $n$ à destination de *mdivn* ainsi que celui partant de $m$ à *mdivn* à la queue Q. Si lors de l'évaluation du *read* de $m$ on n'enlevait pas de la queue Q tous les *read* contenus dans *read(n)* (i.e. le *read(m)* ici), ce ne serait pas juste une perte de temps que de réévaluer le *read(m)*, cela provoquerait une erreur de division par 0.

## 2.4 Travail effectué

Une implémentation incomplète de la librairie *change propagation* est présentée dans [Acar]. Nous avons "traduit" cette librairie en OCaml et implémenté la queue de priorité Q ainsi qu'une structure de données permettant de gérer les time-stamps. Ces différents fichiers se trouvent en annexe A. En annexe B cette librairie est testée sur quelques exemples.

Le fichier *time.ml* implémente les time-stamps comme une référence vers une liste triée de flottants. Cette liste est initialisée avec l'élément 0, le time-stamp le plus grand pouvant être attribué est 1. Lorsqu'un nouveau time-stamp est attribué, il est calculé comme étant la moyenne des deux plus grands time-stamps déjà existants. En effet lorsqu'une modifiable $m$, dont la valeur est utilisée dans un *read* par une autre modifiable $n$, voit sa valeur

changée par un *change*, un time-stamp lui est attribué. Ce time-stamp doit être inférieur au time-stamp de la modifiable destination $n$ qui a déjà été attribué lors de la création de $n$. Ce time-stamp doit également être supérieur au time-stamp d'une éventuelle modifiable $p$ dont la valeur est utilisée pour $n$ mais est lue par un *read* avant que $m$ soit lue. D'où le choix de la moyenne de ces deux time-stamps. L'assurance que les deux derniers time-stamps correspondent bien aux time-stamps qui devront encadrer notre time-stamp à insérer, viens du fait que la liste des time-stamps est maintenue dynamiquement. En effet lors de l'évaluation d'un *read* $r$, l'algorithme *change propagation*, plutôt que d'enlever les arcs contenus dans $r$ de la queue de priorité et du DDG, efface tous les time-stamps (fonction *splice_ out*) contenus dans le time-span de $r$.

Le fichier *q.ml* gère la queue de priorité des arcs affectés. On appelle arcs affectés les arcs sortant des modifiables dont la valeur a été changée par *change*. Les opérations d'insertions, effacement, cardinal y sont implémentées.

Le fichier *lib.ml* représente la librairie avec les opérations d'initialisation *init*, de création de modifiable *modif*, de lecture *read*, d'écriture *write*, et l'algorithme *change propagation*.

## 2.5 Limitations de la librairie OCaml

Considérons l'exemple suivant :

Fichier incorrect_mplus1.ml :

```
let _ =
  let m =
    modif (=) (function d -> (write d 1; write d 5)) in
  let mplus1 =
    modif (=) (function d ->
 read m (function v -> write d (v+1))) in
    change m 7;
    propagate();
    Printf.printf "m=%d mplus1=%d\n" (!(m.value) ()) (!(mplus1.value) ())
```

On remarque à la troisième ligne que la destination est utilisée deux fois lors de deux *write* consécutifs, ce qui viole notre propriété (2) d'usage correct de la librairie (qui exige qu'une expression changeable utilise sa librairie exactement une fois et ceci lors d'une opération *write*). Pourtant la librairie OCaml ne réagit pas à cette violation. Il en est de même lorsqu'une destination n'est jamais utilisée.

Pour pallier ce problème les auteurs présentent dans [Acar] une extension de langage fonctionnel avec appel par valeurs dont la sémantique force l'usage correct.

# 3 Approche monadique de la librairie

Dans [Car] l'auteur traduit la librairie adaptative de [Acar] dans un langage purement fonctionnel, Haskell. Ceci est fait en définissant deux monades : le monade $C$ représentant les expressions de type changeable et le monade A qui implémente les opérations *change* et *propagate*. L'utilisation des monades va assurer à elle toute seule l'usage correct de la librairie.

## 3.1 Les monades

Les langages fonctionnels purs comme Haskell, contrairement au langages simplement fonctionnels comme OCaml, n'offrent pas directement la facilité d'utilisation des fonctionnalités impures : gestion d'erreurs avec les exceptions, instructions d'entrées-sorties, ou tout simplement incrémentation d'une variable globale. L'utilisation de telles fonctionnalités nécessite des modifications importantes du programme en Haskell. Les monades servent justement à faciliter la modification de tels programmes et à augmenter leur modularité.

Voici la définition d'un monade en notation Haskell comme elle est donné dans *The essence of functional programming* (voir [Wad]) :

**Définition 3** *Un monade est un triplet* $(\mathbf{M}, \mathbf{unitM}, \mathbf{bindM})$ *avec* $\mathbf{M}$ *un constructeur de type,* **unitM** *et* **bindM** *deux fonctions de types polymorphiques :*

- $\mathbf{unitM} \ :: \ \mathbf{a} \rightarrow \mathbf{M\ a}$
- $\mathbf{bindM} \ :: \ \mathbf{M\ a} \rightarrow (\mathbf{a} \rightarrow \mathbf{M\ b}) \rightarrow \mathbf{M\ b}$

*unitM* est parfois appelé *return*, *bindM* est parfois noté $>>=$ ou $*$.

Lors de la conversion d'un programme en forme monadique, une fonction de type de départ $\mathbf{a} \rightarrow \mathbf{b}$ sera convertie en une fonction de type $\mathbf{a} \rightarrow \mathbf{M\ b}$. On voit alors que **unitM** correspond à la fonction identité. **bindM** quant à elle représente la composition monadique. En effet : soient deux fonction $\mathbf{k} \ :: \ \mathbf{a} \rightarrow \mathbf{b}$ et $\mathbf{h} \ :: \ \mathbf{b} \rightarrow \mathbf{c}$ qu'on peut composer en écrivant :

```
\a -> let b = k a in h b,
```

la composé est donc de type $\mathbf{a} \rightarrow \mathbf{c}$. Sous forme monadique, deux fonctions $\mathbf{k} \ :: \ \mathbf{a} \rightarrow \mathbf{M\ b}$ et $\mathbf{h} \ :: \ \mathbf{b} \rightarrow \mathbf{M\ c}$ sont composées par **bindM** :

```
(\a -> k a 'bindM' (\b -> h b))
```

qui est de type $\mathbf{a} \rightarrow \mathbf{M\ c}$.

Trois lois monadiques demandent que **bindM** soit associative et admette **unitM** comme élément neutre à gauche et à droite.

D'autres opérations primitives peuvent être définies dans un monade selon l'utilisation recherchée. Ainsi en Haskell les monades implémentent les listes, les tableaux, gèrent les entrées-sorties, etc.

La librairie adaptative étant écrite en CPS, considérons en exemple le monade $C$ des continuations :

```
type C a    = (a -> Answer) -> Answer
unitC a     = \c -> c a
m 'bindC' k = \c -> m (/a -> k a c)
```

Le type *Answer* représente le résultat final d'un calcul. En CPS, une valeur $a$ (de type $a$) est représentée par une fonction qui prend en paramètre une continuation $c$ (de type $a \rightarrow Answer$), applique la continuation à la valeur $a$, et produit comme résultat $c\ a$ (de type $Answer$). Ainsi, dans le monade $C$, $unitC\ a$ est la représentation CPS de $a$. Avec $m :: C\ a$ et $k :: C\ b$, l'action de $m$ *'bindC'* $k$ se lit : soit $c$ la continuation courante, évaluer $m$ et mettre le résultat dans $a$, appliquer $k$ à $a$ avec la continuation $c$.

Dans [Wad] l'auteur montre qu'il existe une correspondance étroite entre les continuations et le monade $C$.

## 3.2 Monades C et A

Voici l'interface simplifiée de la librairie adaptative en Haskell donnée dans [Car] :

```
newtype A a
newtype C a
newtype Mod a
instance Monad A
instance Monad C

class Monad m => Newmod m where
  newModBy :: (a -> a -> Bool) -> C a -> m (Mod a)
instance NewMod A
instance NewMod C

newMod :: (NewMod m, Eq a) => C a -> m (Mod a)
```

```
readMod :: Mod a -> C a

change :: Mod a -> a -> A ()
propagate :: A ()
```

Nous avons remarqué précédemment que la mise à jour des modifiables est faite dans un style de programmation CPS : les expressions changeables prennent en paramètre une destination *'a dest* (paramètre de continuation ici) dans laquelle elles écrivent la valeur mise à jour de la modifiable. C'est ainsi que le monade $C$ précédent va être utilisé dans [Car] pour représenter les expressions changeables. Le type de ce monade est précisémment :

```
(a -> Changeable) -> Changeable
```

L'opération *read*, qui est de type ml :

```
'a mod * ('a ->changeable) -> changeable
```

se traduit ainsi immédiatement en une opération du monade C appelée *readMod* de type **Mod a** $\rightarrow$ **C a**.

Le passage de la continuation au monade $C$ dans [Car] permet de cacher le paramètre de continuation *'a dest* dans le monade. En cachant cette destination, sa possibilité d'utilisation directe par l'utilisateur de la librairie est éliminée.

C'est aussi pourquoi l'opération *write* n'apparaît plus dans l'interface Haskell de la librairie. En effet, il est souhaitable que *write* fasse partie de la continuation pour pouvoir la "cacher" de l'utilisateur, pour que ce dernier ne puisse pas écrire plus d'une fois ou aucune fois dans une modifiable. L'opération *write* va donc se trouver cachée dans l'opération *newMod* de création de modifiables de sorte que la propriété (2) d'usage correct de la librairie se trouve forcément respectée. Ainsi la version monadique en Haskell de l'exemple incorrect donné à la section 2.5 est (comme donné dans *Car*) :

```
do m      <- newMod (return 1)
   mplus1 <- newMod (do v <- readMod m
                        return (v + 1))
   change m 2
   propagate
```

On remarque qu'aucun *write* n'apparaît dans le code, et que le paramètre destination a disparu. Il est alors évident que l'usage incorrect présenté dans l'exemple de la section 3.1 est ici impossible, la propriété (2) est toujours assurée.

Le monade $A$ implémente les opérations adaptatives *change* et *propagate*. De plus l'opération *newMod* y est surchargée pour permettre la création de modifiables dans les deux monades, $A$ et $C$. L'algorithme de *change propagation* implémenté est le même que celui de la librairie ML.

## 3.3 Travail effectué

Nous avons implémenté en OCaml l'approche monadique de la librairie adaptative. Les différents fichiers se trouvent en Annexe C. Cette implémentation est basée sur le monade $C$ uniquement, le monade $A$ n'a pas été construit.

Le module *Monad* comporte une interface pour les monades et une implantation en monade C du monade de continuation. Les fonctions monadiques *return* et *bind* ont été implémentées en se basant sur les exemples d'utilisation donnés en section 2.2 dans [Car].

Les fichiers *types.ml*, *time.ml* et *q.ml* restent les mêmes que précédemment. Le fichier *monadic_lib.ml* implémente la nouvelle librairie. Regardons de plus près ce qui a changé.

La fonction *new_mod* est presque la même que *modif* (à des renommages de fonctions auxiliaires près qui rappellent la version Haskell) sauf les deux dernières lignes :

```
let modif cmp f =
 ...
f(m);
m
```

```
let new_mod cmp f =
  ...
f (fun v -> !(m.wrt) v);
C.return m
```

Revenons à la version Haskell pour comprendre. La fonction de création de modifiables en Haskell *newMod* peut-être exprimée par rapport à *mod* :

```
newMod :: (a -> a -> Bool) -> C a -> C (Mod a)
newMod cmp c = mod cmp (\d -> c (\a -> write d a))
```

Le comparateur cmp est le même mais l'initialiseur a changé. La valeur $a$ de type $a$ qui représente la valeur à écrire dans la modifiable, est maintenant codée dans le monade C, c'est un type $C\ a$. A la ligne $f(fun\ v \rightarrow !(m.wrt)\ v)$ on récupère dans la variable $v$ cette valeur codée et on lance son écriture dans la modifiable en appelant l'initaliseur f comme dans le cas précédent. Et enfin le type de retour de la fonction est désormais monadique d'où le dernier return.

La fonction d'écriture *write* a disparu, la lecture, *change* et *propagate* restent inchangées.

Maintenant l'exemple de la section 3.4 peut s'écrire en OCaml :

Fichier mplus1_monadic.ml :

```
let (>>=) = C.bind
let return = C.return
let (>>) = C.(>>)
let eval m = m (fun x -> x)

let prog =
  new_mod (=) (return 1) >>= fun m ->
  new_mod (=) (read_mod m >>= fun v ->
               return (v + 1)) >>= fun mplus1 ->
  change m 2;
  propagate ();
  Printf.printf "m=%d mplus1=%d\n" (!(m.value) ()) (!(mplus1.value) ());
  return ()

let _ = eval prog
```

et son exécution produit le même résultat que son homologue non-monadique, i.e. imprime $m = 2$ $mplus1 = 3$ sur la sortie standard. Nous avons mis en annexe C l'équivalent de l'exemple de la section 2.1.

# 4 Conclusion

Lors de ce travail de recherche encadré la notion de calcul incrémental a été étudiée à travers la technique de *change propagation*. L'étude de [Haskell] a permis dans un premier temps de se familiariser avec le langage Haskell et tout particulièrement avec la notion de monade. Une idée plus précise de l'intérêt du travail effectué dans [Car] a ainsi pu se mettre en place.

Dans un deuxième temps, l'étude de [Acar], article sur lequel [Car] s'appuie, a permis une compréhension approfondie de l'algorithme de *change propagation*. Cette compréhension a été illustrée par l'implémentation en OCaml de la librairie présentée dans cet article. Cette librairie a ainsi pu être testée sur des exemples.

Dans un troisième temps, le retour à [Car] et l'étude de la librairie qui y est présentée (voir [Lib] a permis de comprendre l'implémentation de la librairie sous forme monadique. Le monade $C$ a été implémenté et la librairie a été réécrite en fonction de ce monade. L'usage correct a ainsi pu être assuré.

Il reste maintenant à réfléchir sur les avantages de l'utilisation du monade $A$ et à implémenter ce monade dans notre librairie OCaml pour mieux rendre compte de son éventuel intérêt.

# Références

[Acar] U. Acar, G. Blelloch et R. Harper. Adaptive functional programming. http://ttic.uchicago.edu/~umut/papers/toplas06.html.

[Car] M. Carlsson. Monads for incremental computing, www.cse.ogi.edu/~magnus/papers/icfp-2002.pdf.

[Haskell] A gentle introduction to Haskell, version 98, chapitres 1 - 9. http://www.haskell.org/tutorial/.

[Hugs] The Hugs 98 interpreter. www.haskell.org/hugs/.

[Lib] M. Carlsson. Implementation of the library in Haskell. http://www.cse.ogi.edu/~magnus/Adaptive/.

[Liu] Y. Liu, S. Stoller et T. Teitelbaum. Discovering auxiliary information for incremental computation, p. 1. www.cs.sunysb.edu/~stoller/papers/POPL96.pdf.

[OCaml] E. Chailloux, P. Manoury et B. Pagano. Developpement d'applications avec Objective Caml, chapitre 14. O'Reilly, 2000.

[Wad] P. Wadler. The essence of functional programming. http://homepages.inf.ed.ac.uk/wadler/topics/monads.html.

# A Librairie OCaml

Le fichier **time.ml** implémente les time-stamps :

```
type t = float

let init () =
  0.

let rec last_els () =
  let rec aux = function
    [] | [_] -> failwith "last_els"
    | [x; y] -> (x, y)
    | h :: t -> aux t in
  aux !time_stamps

let mean_last_els () =
  let (x, y) = last_els () in
    (x +. y)/.2.

let next_time () =
  match !time_stamps with
    [] -> 0.
    | [x] -> x +. 1.
    | l -> mean_last_els ()

let insert time =
  time_stamps := List.sort compare (!time :: !time_stamps);
  time := next_time ();
  !time

let splice_out t1 t2 =
  time_stamps := List.filter (fun t -> not (t1 <= t && t <= t2)) !time_stamps

let is_spliced_out t =
  not (List.mem t !time_stamps)
```

Le fichier **types.ml** définit les types utilisés :

```
type reader = unit -> unit
type time_span = Time.t * Time.t
type edge = {reader: reader; time_span: time_span}
type 'a node = {value: (unit -> 'a) ref; wrt: ('a -> unit) ref;
out_edges: edge list ref}
type 'a dest = 'a node
type 'a modif = 'a node
type changeable = unit
```

Le fichier **q.ml** implémente la queue de priorité :

```
open Types

module S = Set.Make(struct
             type t = Types.edge
             let compare e1 e2 =
       compare (fst e1.time_span) (fst e2.time_span)
   end)

let empty = S.empty

let is_empty = S.is_empty

(* [delete_min q] deletes the edge with the least time-stamp from
   the queue and returns the couple (edge deleted, new queue) *)
let delete_min q =
  let edge = S.min_elt q in
  let nq = S.remove edge q in
    (edge, nq)

let insert = S.add

let cardinal = S.cardinal

let iter = S.iter
```

Le fichier **lib.ml** implémente la librairie :

```
open Types
open Debug

exception Unset_Modif

(* current_time is used to generate the next time-stamp *)
let current_time = ref(Time.init())

(* pq is the priority queue of affected edges, initially empty *)
let pq = ref(Q.empty)

(* initializing the library *)
let init () =
  current_time := Time.init();
  pq := Q.empty

(* creation of modifiables : cmp is the comparison function,
   f is the initialiser function *)
let modif cmp f =
  (* initially *)
  let value = ref(function _ -> raise Unset_Modif) in
  let wrt = ref(function v -> raise Unset_Modif) in
  let out_edges = ref [] in
  let m = {value= value; wrt= wrt; out_edges=out_edges} in
    (* when a value v is written with write *)
  let change t v =
    (* if v is different from the existing one *)
    (if cmp v (!value ())
     then
       (* then nothing is done *)
       ()
     else
       (* else the new value v is written, *)
       (value := (function _ -> v);
(* the out_edges of m are inserted in the priority queue, *)
List.iter (function x -> pq := Q.insert x !pq) !out_edges;
(* the out_edges list is emptied so to avoid re-inserting the
   same edges during next iteration *)
out_edges := []));
```

```
    current_time := t in
  let write v =
    value := (function _ -> v);
    (* generation of the node's time-stamp *)
    wrt := change (Time.insert(current_time)) in
    (* effective writing of the modifiable's value *)
    wrt := write;
    (* the initialiser is called *)
    f(m);
    (* the modifiable is returned *)
    m

(* writing the value v in a modifiable (which is a dest type) d *)
let write d v =
  !(d.wrt) v

(* reading a modifiable n and applying the reader f to it *)
let read n f =
  (* generating the start time of the read = time-stamp of the read *)
  let start = Time.insert(current_time) in
  let rec run () =
    (* preparing to apply the reader f *)
    f (!(n.value)());
    (* updating the out_edges of the modifiable n being read by adding
       the current read to it *)
    n.out_edges :=
      {reader=run; time_span=(start, !current_time)} :: !(n.out_edges) in
    (* applying the reader *)
    run ()

(* changing the value v of modifiable l by explicitely writing to it *)
let change l v = write l v

(* auxiliary fonction for propagation *)
let rec propagate_aux () =
  (* if nothing to evaluate *)
  if (Q.is_empty(!pq))
  then
    (* then nothing is done *)
    ()
  else
```

```
    (* else consider the first edge to be evaluated *)
    let (edge, npq) = Q.delete_min(!pq) in
    let {reader=f; time_span=(start, stop)} = edge in
      (* remove all other edges from evaluation priority queue *)
       pq := npq;
      (* if the time-stamp of the edge was deleted on previous iteration,
 i.e. this edge is not to be re-evaluated *)
       if (Time.is_spliced_out start)
       then
 (* then continue propagating on other edges *)
 propagate_aux ()
       else (
 (* else delete edges contained in the current edge, *)
 Time.splice_out start stop;
 current_time := start;
 (* evaluate edge by calling its reader *)
 f();
 propagate_aux())

(* propagating the changes *)
let propagate () =
  let ctime = !current_time in
    propagate_aux();
    current_time := ctime
```

# B Tests de la librairie Ocaml

L'exécution de l'exemple suivant, fichier **mplus1.ml** :

```
let _ =
  let m =
    modif (=) (function d -> (write d 1)) in
  let mplus1 =
    modif (=) (function d ->
 read m (function v -> write d (v+1))) in
    change m 2;
    propagate();
    Printf.printf "m=%d mplus1=%d\n" (!(m.value) ()) (!(mplus1.value) ())
```

imprime : m=2 mplus1=3.

Celle de l'exemple suivant, fichier **mtimesn.ml** :

```
let _ =
  let m =
    modif (=) (function d -> (write d 1)) in
  let n =
    modif (=) (function d -> (write d 2)) in
  let mn =
    modif (=) (function d ->
 read m (fun v ->
     (read n (fun w -> write d (w * v))))) in
    change m (int_of_string Sys.argv.(1));
    change n (int_of_string Sys.argv.(2));
    propagate();
    Printf.printf "%d*%d=%d\n" (!(m.value) ()) (!(n.value) ()) (!(mn.value) ())
```

avec comme arguments 7 et 5, imprime : $7 * 5 = 35$

Celle de l'exemple suivant, fichier **fact.ml** :

```
let rec fact n =
  if (n = 0)
  then
    1
  else
    n * fact (n-1)

let factorial n p =
    modif (=) (fun d -> read p (fun p ->
      if (not p)
      then
write d 1
      else
read n (fun n -> write d (fact n))))
let _ =
  let n = modif (=) (fun n -> write n 0) in
  let p = modif (=)
    (fun p -> read n (fun n -> write p (n >= 0))) in
    change n (int_of_string Sys.argv.(1));
    propagate ();
    Printf.printf "%d\n" (!((factorial n p).value) ())
```

avec comme argument 3, imprime : 6.

# C Librairie monadique

Le fichier **monad.ml** implémente le monade C :

```
(* Interface of a monad and its principal operations *)
module Monad
 (M :
    sig
      type 'a t
      val return : 'a -> 'a t
      val bind : 'a t -> ('a -> 'b t) -> 'b t
      val fail : string -> 'a t
    end) =
struct
  type 'a t = 'a M.t
  let bind = M.bind
  let (>>=) = bind
  let return = M.return
  let fail = M.fail
  let (>>) m k = m (>>=) (fun _ -> k)
  let lift f m = m (>>=) (function x -> return (f x))
end

(* The PreContMonad module implements the continuation monad *)
module PreContMonad (T : sig type t end) =
struct
  type 'a t = ('a -> T.t) -> T.t
  let return a = fun f -> f a
  let bind m f = fun c -> m (fun h -> f h c)
  let fail s = failwith s
end

(* The C module represents the changeable monad (which is a particular case
   of the continuation monad) *)
module PreChangeableMonad = PreContMonad (struct type t = Types.changeable end)

module ChangeableMonad = Monad(PreChangeableMonad)

module C = ChangeableMonad
```

Le fichier**monadic_lib.ml** implémente la librairie monadique :

```
open Types
open Debug
open Monad

let (>>=) = C.bind

exception Unset_Modif

(* current_time is used to generate the next time-stamp *)
let current_time = ref(Time.init())

(* pq is the priority queue of affected edges, initially empty *)
let pq = ref(Q.empty)

(* initializing the library *)
let init () =
  current_time := Time.init();
  pq := Q.empty

(* creation of modifiables : cmp is the comparison function,
   f is the initialiser function *)
let new_mod cmp f =
  let value = ref(function _ -> raise Unset_Modif) in
  let wrt = ref(function v -> raise Unset_Modif) in
  let out_edges = ref [] in
  let m = {value= value; wrt= wrt; out_edges=out_edges} in
  let write_again t v =
    (if cmp v (!value ())
     then
       ()
     else
       (value := (function _ -> v);
List.iter (function x -> pq := Q.insert x !pq) !out_edges;
(* the out_edges list is emptied so to avoid re-inserting the
   same edges during next iteration *)
out_edges := []));
    current_time := t in
  let write_first v =
    value := (function _ -> v);
```

```
    (* generation of the node's time-stamp *)
    wrt := write_again (Time.insert(current_time)) in
    (* effective writing of the modifiable's value *)
    wrt := write_first;
    (* the initialiser is called *)
    f (fun v -> !(m.wrt) v);
    (* the C modifiable is returned *)
    C.return m

(* reading a modifiable n and applying the reader f to it *)
let read_mod m f =
  (* generating the start time of the read = time-stamp of the read *)
  let start = Time.insert(current_time) in
  let rec run () =
    (* preparing to apply the reader f *)
      f (!(m.value) ());
    (* updating the out_edges of the modifiable n being read by adding
       the current read to it *)
      m.out_edges :=
        {reader=run; time_span=(start, !current_time)} :: !(m.out_edges) in
    (* applying the reader *)
    run ()

(* changing the value v of modifiable l *)
let change l v =
  !(l.wrt) v

(* auxiliary fonction for propagation *)
let rec propagate_aux () =
  (* if nothing to evaluate *)
  if (Q.is_empty(!pq))
  then
    (* then nothing is done *)
    ()
  else
    (* else consider the first edge to be evaluated *)
    let (edge, npq) = Q.delete_min(!pq) in
    let {reader=f; time_span=(start, stop)} = edge in
      (* remove all other edges from evaluation priority queue *)
       pq := npq;
      (* if the time-stamp of the edge was deleted on previous iteration,
```

```
 i.e. this edge is not to be re-evaluated *)
       if (Time.is_spliced_out start)
       then
 (* then continue propagating on other edges *)
 propagate_aux ()
       else (
 (* else delete edges contained in the current edge, *)
 Time.splice_out start stop;
 current_time := start;
 (* evaluate edge by calling its reader *)
 f();
 propagate_aux())

(* propagating the changes *)
let propagate () =
  let ctime = !current_time in
    propagate_aux();
    current_time := ctime
```

L'exemple suivant, **monadic_mtimesn.ml** :

```
let (>>=) = C.bind
let return = C.return
let (>>) = C.(>>)
let eval m = m (fun x -> x)

let prog =
  new_mod (=) (return 0) >>= fun m ->
  new_mod (=) (return 0) >>= fun n ->
  new_mod (=) (read_mod m >>= fun m ->
               read_mod n >>= fun n ->
               return (m * n)) >>= fun mn ->
  change m 3;
  change n 4;
  propagate ();
  Printf.printf "%d*%d=%d\n" (!(m.value) ()) (!(n.value) ()) (!(mn.value) ());
  return ()

let _ = eval prog
```

imprime sur la sortie standard : $3 * 4 = 12$.