

Le Système de Burstall-Darlington

un système de transformation de programmes pour passer d'un

programme informatique
à un
programme efficient

1. par des optimisations :

temps exponentiel \longrightarrow temps linéaire
récursion \longrightarrow itération

2. et un langage spécifique : **NPL** avec les caractéristiques suivantes :

- 1er ordre
- typé
- présentation équationnelle

un programme NPL s'écrit sous forme d'un système de *É*quations de Récurrence, des *REQs*, et est composé de :

- *fonctions primitives* : constructeurs, par ex. : $+$, $*$, \dots , $:$ (cons), $@$ (append)
- *variables* : les paramètres
- *symboles de fonctions récursives* : f_i
- *expressions* : combinaisons de f_i et d'*expressions conditionnelles* $x \Rightarrow y|z$
- *expressions gauches (EG)* : $f_i(e_1, \dots, e_n)$
- *expressions droites (ED)*
- *REQs* : $EG \Leftarrow ED$

exemples :

calcul de la factorielle :

```
fac: nb -> nb
fac(0) <= 1
fac(n + 1) <= (n + 1) * fac(n)
```

définition de append :

```
@: liste x liste -> liste
nil @ M <= M
(n : L) @ M <= n : (L @ M)
```

Les transformations sont exprimées sous forme de **Règles d'Inférences** :

1. DEFINITIONS : nouvelles REQs d'EG distinctes de toutes celles déjà existantes
2. INSTANCIATIONS (de variables) : ce sont des dérivations, des exécutions réelles (non symboliques), ce qui impliquent un remplacement de toutes les variables de même nom par la même valeur
3. UNFOLDING :

soit :

$$\begin{array}{l} E \Leftarrow E' \\ F \Leftarrow F' \end{array}$$

avec une occurrence dans F' d'une instance de E

le *unfolding* permet alors d'écrire une nouvelle REQ :

$$F \Leftarrow F'' \text{ t.q. } F'' = F'[E'/E]$$

où $F[x/y]$ est la substitution de x pour toute occurrence de y dans F

exemple :

$$\begin{array}{l} \overbrace{g(x)}^E \Leftarrow \overbrace{k(x, f(x))}^{E'} \\ \overbrace{f(x)}^F \Leftarrow \overbrace{g(h(x))}^{F'} \end{array}$$

clairement, $E, g(x)$, a une occurrence instanciée comme $g(h(x))$ dans F' ; on peut alors *unfold* en :

$$f(x) \Leftarrow k(h(x), f(h(x)))$$

4. FOLDING :

soit :

$$\begin{array}{l} E \Leftarrow E' \\ F \Leftarrow F' \end{array}$$

et une occurrence de E' dans F'

le *folding* permet alors d'écrire une nouvelle REQ

$$F \Leftarrow F'' \text{ t.q. } F'' = F'[E/E']$$

exemple :

$$\begin{array}{ccc} \overbrace{g(x)}^E & \Leftarrow & \overbrace{k(x, f(x))}^{E'} \\ \overbrace{g(c(x))}^F & \Leftarrow & \overbrace{k(h(x), f(h(x)))}^{F'} \end{array}$$

clairement, $k(x, f(x))$ a une occurrence instanciée comme $k(h(x), f(h(x)))$. On peut alors *folder* en :

$$g(c(x)) \Leftarrow g(h(x))$$

5. LOIS : transformation d'une ED par une loi sur les fonctions primitives (associativité, commutativité, ...)

6. ABSTRACTION :

soit :

$$E \Leftarrow E'$$

nouvelle REQ : $E \Leftarrow E'[\dots u_i/F_i \dots]$ where $\langle \dots u_i \dots \rangle = \langle \dots F_i \dots \rangle$

Bien entendu, les transformations doivent préserver la correction des programmes.

Quelques exemples de transformations :

1. *optimisation du parcours* : soit les deux fonctions **double** et **@**

$$\text{double} : \text{liste_de_nombres} \rightarrow \text{liste_de_nombres}$$

$$\text{@} : \text{liste} \times \text{liste} \rightarrow \text{liste}$$

- | | | |
|----|--|-----|
| 1. | double(nil) \Leftarrow nil | DEF |
| 2. | double(n : L) \Leftarrow (2 * n) : double(L) | DEF |
| 3. | nil @ M \Leftarrow M | |
| 4. | (n : L) @ M \Leftarrow n : (L @ M) | DEF |

on veut construire par composition la fonction **f** :

- | | | |
|----|------------------------------------|-----|
| 5. | f(L, M) \Leftarrow double(L @ M) | DEF |
|----|------------------------------------|-----|

inefficace : deux traversées de la liste L; transformons alors :

- | | | |
|----|--|---------------------------|
| 6. | f(nil, M) \Leftarrow double(nil @ M) | Instanciation de L dans 5 |
| | \Leftarrow double(M) | UNFOLD avec 3 |
| 7. | f(n : L, M) \Leftarrow double((n : L) @ M) | Instanciation de L dans 5 |
| | \Leftarrow double(n : (L @ M)) | UNFOLD avec 4 |
| | \Leftarrow (2 * n) : double(L @ M) | UNFOLD avec 2 |
| | \Leftarrow (2 * n) : f(L, M) | FOLD avec 5 |

Ce qui donne une nouvelle définition de **f** avec une seule traversée de la liste **L** :

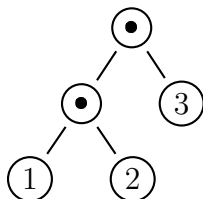
$$\begin{aligned} f(\text{nil}, M) &\Leftarrow \text{double}(M) \\ f(n : L, M) &\Leftarrow (2 * n) : f(L, M) \end{aligned}$$

En Lisp ce sera quelque chose comme :

```
(de f (L M)
  (if (null L) (double M)
      (cons (* 2 (car L)) (f (cdr L) M))))
```

2. optimisation de l'espace utilisé

soit un arbre binaire comme :



qui se construit par : $tree(tree(tip(1), tip(2)), tip(3))$, et la fonction

$$lin : tree \rightarrow liste_de_nombres$$

1. $lin(tip(n)) \Leftarrow n : nil$ DEF
2. $lin(tree(S, T)) \Leftarrow lin(S) @ lin(T)$ DEF

soit la fonction

$$fpos : liste_de_nombres \rightarrow nombre$$

qui ramène le premier nombre > 0 , sinon 0

3. $fpos(nil) \Leftarrow 0$ DEF
4. $fpos(0 : L) \Leftarrow fpos(L)$ DEF
5. $fpos(n+1 : L) \Leftarrow n + 1$ DEF

problème : trouver le premier $tip > 0$ d'un arbre binaire en parcours préfixe. Définissons alors la fonction $g : tree \rightarrow nombre$ comme :

6. $g(T) \Leftarrow fpos(lin(T))$ DEF

inefficace : elle génère trop de append et génère la liste de tous les tips, même si c'est déjà le premier qui est différent de 0. Essayons d'optimiser :

7. $g(tip(0)) \Leftarrow fpos(lin(tip(0)))$ Instanciation de T dans 6
 $\Leftarrow fpos(0 : nil)$ UNFOLD avec 1
 $\Leftarrow fpos(nil)$ UNFOLD avec 4
 $\Leftarrow 0$ UNFOLD avec 3
8. $g(tip(n + 1)) \Leftarrow fpos(lin(tip(n + 1)))$ Instanciation de T dans 6
 $\Leftarrow fpos(n : nil)$ UNFOLD avec 1
 $\Leftarrow n + 1$ UNFOLD avec 5
9. $g(tree(S, T)) \Leftarrow fpos(lin(tree(S, T)))$ Instanciation de T dans 6
 $\Leftarrow fpos(lin(S) @ lin(T))$ UNFOLD avec 2

coincé : faut considérer un LEMME :

$\text{fpos}(L@M) \Leftarrow \text{fpos}(L) = 0 \Rightarrow \text{fpos}(M) \mid \text{fpos}(L)$ démontrable par récurrence

donc Instanciation du LEMME
 9. $g(\text{tree}(S, T)) \Leftarrow \text{fpos}(\text{lin}(S)) = 0 \Rightarrow \text{fpos}(\text{lin}(T)) \mid \text{fpos}(\text{lin}(S))$
ABSTRACTION
 .
 $\Leftarrow (n = 0) \Rightarrow \text{fpos}(\text{lin}(T)) \mid n$ where $n = \text{fpos}(\text{lin}(S))$
 $\Leftarrow (n = 0) \Rightarrow g(T) \mid n$ where $n = g(S)$ FOLD avec 6

Nous obtenons alors la nouvelle définition :

$$g(\text{tip}(0)) \Leftarrow 0$$

$$g(\text{tip}(n + 1)) \Leftarrow n + 1$$

$$g(\text{tree}(S, T)) \Leftarrow (n = 0) \Rightarrow g(T) \mid n \text{ where } n = g(S)$$

Ce qui donne en Lisp quelque chose comme :

```
(de g (s)
  (cond
    ((zerop s) nil)
    ((atom s) s)
    (t (let (n (g (car s)))
        (if (zerop n) (g (cadr s) n))))))
```

3. *transformation récursif* \rightarrow *itératif* par généralisation d'une définition,
c'est-à-dire : par introduction d'une fonction auxiliaire comportant un accumulateur

exemple : $length : liste \rightarrow nombre$

- | | |
|---|-----|
| 1. $length(nil) \Leftarrow 0$ | DEF |
| 2. $length(a : L) \Leftarrow 1 + length(L)$ | DEF |

généralisation de $1 + length(L)$ en $n + length(L)$

- | | |
|---|-----------------------------|
| 3. $l(n, L) \Leftarrow n + length(L)$ | DEF |
| 4. $l(n, nil) \Leftarrow n + length(nil)$ | Instanciation de L dans 3 |
| $\Leftarrow n + 0$ | UNFOLD avec 1 |
| $\Leftarrow n$ | lois sur "+" |
| 5. $l(n, a : L) \Leftarrow n + length(a : L)$ | Instanciation de L dans 3 |
| $\Leftarrow n + (1 + length(L))$ | UNFOLD avec 2 |
| $\Leftarrow (n + 1) + length(L)$ | lois d'associativité de "+" |
| $\Leftarrow l(n + 1, L)$ | FOLD avec 3 |

ce qui livre la nouvelle définition de $length$:

$$length(L) \Leftarrow l(0, L)$$

avec

$$\begin{aligned} l(n, nil) &\Leftarrow n \\ l(n, a : L) &\Leftarrow l(n + 1, L) \end{aligned}$$

en Lisp :

```
(de length (L) (1 0 L))

(de l (n L)
  (if (null L) n
      (1 (1+ n) (cdr L))))
```

4. technique du TUPLING

soit la fonction de Fibonacci $f : \text{nombre} \rightarrow \text{nombre}$

- | | |
|--|-----|
| 1. $f(0) \Leftarrow 1$ | DEF |
| 2. $f(1) \Leftarrow 1$ | DEF |
| 3. $f(x + 2) \Leftarrow f(x + 1) + f(x)$ | DEF |

et la fonction auxiliaire $g : \text{nombre} \rightarrow \text{couple_de_nombres}$

- | | |
|--|---------------------------|
| 4. $g(x) \Leftarrow \langle f(x + 1), f(x) \rangle$ | DEF |
| 5. $g(0) \Leftarrow \langle f(1), f(0) \rangle$ | Instanciation de x dans 4 |
| $\Leftarrow \langle 1, 1 \rangle$ | UNFOLD avec 1 et 2 |
| 6. $g(x + 1) \Leftarrow \langle f(x + 2), f(x + 1) \rangle$ | Instanciation de x dans 4 |
| $\Leftarrow \langle f(x + 1) + f(x), f(x + 1) \rangle$ | UNFOLD avec 3 |
| $\Leftarrow \langle u + v, u \rangle \text{ where } \langle u, v \rangle = \langle f(x + 1), f(x) \rangle$ | ABSTRACTION |
| $\Leftarrow \langle u + v, u \rangle \text{ where } \langle u, v \rangle = g(x)$ | FOLD avec 4 |
| 7. $f(x + 2) \Leftarrow u + v \text{ where } \langle u, v \rangle = \langle f(x + 1), f(x) \rangle$ | ABSTRACTION |
| $\Leftarrow u + v \text{ where } \langle u, v \rangle = g(x)$ | FOLD de 4 |

ce qui donne les nouvelles définitions :

$$\begin{aligned}
 f(0) &\Leftarrow 1 \\
 f(1) &\Leftarrow 1 \\
 f(x + 2) &\Leftarrow u + v \text{ where } \langle u, v \rangle = g(x) \\
 g(0) &\Leftarrow \langle 1, 1 \rangle \\
 g(x + 1) &\Leftarrow \langle u + v, u \rangle \text{ where } \langle u, v \rangle = g(x)
 \end{aligned}$$

la première définition calculait du haut vers le bas, la nouvelle calcule du bas vers le haut, garantissant ainsi un temps d'exécution linéaire.

en Lisp :

```

(de f (x)
  (if (< x 2) 1
      (let (uv (g (- x 2)))
        (let ((u (car uv)) (v (cdr uv)))
          (+ u v))))))
(de g (x)
  (if (zerop x) (cons 1 1)
      (let (uv (g (1- x)))
        (let ((u (car uv) (v (cdr uv)))
              (cons (+ u v) u))))))

```

5. suite de la technique du TUPLING : calcul d'une liste de factorielles

$$\begin{aligned} flist &: \text{nombre} \rightarrow \text{liste_de_nombre} \\ fac &: \text{nombre} \rightarrow \text{nombre} \end{aligned}$$

1. $fac(0) \Leftarrow 1$ DEF
2. $fac(n + 1) \Leftarrow (n + 1) * fac(n)$ DEF
3. $flist(0) \Leftarrow nil$ DEF
4. $flist(n + 1) \Leftarrow fac(n + 1) : flist(n)$ DEF

très inefficace ! Généralisation : $g : \text{nombre} \rightarrow \text{couple} : \text{nombre} + \text{liste}$

5. $g(n) \Leftarrow \langle fac(n + 1), flist(n) \rangle$ DEF
6. $g(0) \Leftarrow \langle fac(1), flist(0) \rangle$ Instanciation de n dans 5
 $\Leftarrow \langle fac(1), nil \rangle$ UNFOLD avec 3
 $\Leftarrow \langle 1, nil \rangle$ UNFOLD avec 1
7. $g(n+1) \Leftarrow \langle fac(n + 2), flist(n + 1) \rangle$ Instanciation de n dans 5
 $\Leftarrow \langle fac(n + 2), fac(n + 1) : flist(n) \rangle$ UNFOLD avec 4
 $\Leftarrow \langle (n + 2) * fac(n + 1), fac(n + 1) : flist(n) \rangle$ UNFOLD avec 2
 $\Leftarrow \langle (n + 2) * u, u : v \rangle \text{ where } \langle u, v \rangle = \langle fac(n + 1), flist(n) \rangle$
ABSTRACTION de 4
 $\Leftarrow \langle (n + 2) * u, u : v \rangle \text{ where } \langle u, v \rangle = g(n)$ FOLD avec 5
8. $flist(n + 1) \Leftarrow u : v \text{ where } \langle u, v \rangle = \langle fac(n + 1), flist(n) \rangle$
ABSTRACTION de 4
 $\Leftarrow u : v \text{ where } \langle u, v \rangle = g(n)$ FOLD avec 5

ce qui nous donne une nouvelle définition qui, comme dans l'exemple précédent, s'exécute en temps linéaire :

$$\begin{aligned} flist(0) &\Leftarrow nil \\ flist(n + 1) &\Leftarrow u : v \text{ where } \langle u, v \rangle = g(n) \\ g(0) &\Leftarrow \langle 1, nil \rangle \\ g(n + 1) &\Leftarrow \langle (n + 2) * u, u : v \rangle \text{ where } \langle u, v \rangle = g(n) \end{aligned}$$

En Lisp ces deux fonctions peuvent s'écrire comme :

<pre>(de flist (n) (if (zerop n) nil (let (uv (g (1- n))) (let ((u (car uv)) (v (cdr uv))) (cons u v))))))</pre>	<pre>(de g (n) (if (zerop n)(cons 1 nil) (let (uv (g (1- n))) (let ((u (car uv)) (v (cdr uv))) (cons (* (1+ n) u) (cons u v))))))</pre>
--	---