Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle

Etienne Duris

Thèse de doctorat de l'Université d'Orléans

Orléans - 5 Octobre 1998.

UNIVERSITE D'ORLEANS

INRIA

$\boxed{\text{Cadre général}}$

Comparer différents styles de programmation

et leurs approches des problèmes à résoudre

But : faire émerger de nouvelles solutions

– programmation déclarative, dirigée par la structure

$\Rightarrow$ expressivité des spécifications

$\Rightarrow$ transformations des programmes

Travaux menés dans la thèse

Grammaires attribuées *versus* **programmation fonctionnelle**

- **Grammaires attribuées dynamiques**
  - → autorisent les spécifications sur des structures abstraites.
  - → expressivité d'un langage fonctionnel du 1er ordre.
- **Panorama de méthodes de déforestation fonctionnelles**.
  - → présentation, comparaisons, limites.
- **Composition symbolique**
  - → étend la composition descriptionnelle des gram. attr.
  - → utilisable sur des programmes fonctionnels.
  - → résoud une classe de problèmes durs.

Plan de l'exposé

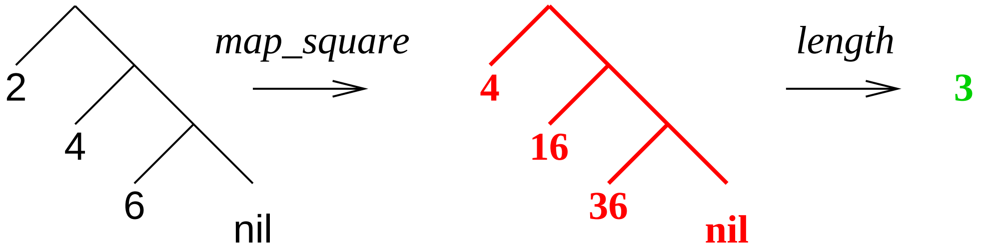
- La **déforestation**, c'est quoi ?
- Principales méthodes en **programmation fonctionnelle**
- Les **grammaires attribuées** comme alternative
- La **composition symbolique**
- Résultats - conclusions - prespectives

Structure de donnée intermédiaire

$$
\begin{array}{l}
map\_square\ x = \texttt{case}\ x\ \texttt{of} \\
\quad cons\ head\ tail \rightarrow \\
\quad\quad cons\ (square\ head)\ (map\_square\ tail) \\
\quad nil \rightarrow nil
\end{array}
$$

$$
\begin{array}{l}
length\ x = \texttt{case}\ x\ \texttt{of} \\
\quad cons\ head\ tail \rightarrow \\
\quad\quad 1 + length\ tail \\
\quad nil \rightarrow 0
\end{array}
$$

Si on applique la composition $length \circ map\_square$ sur une liste...



$\boxed{\text{La déforestation}}$

Dans certaines compositions

$f \circ g$

g produit une

structure de donnée intermédiaire

qui est consommée par f.

- Fondamental en programmation modulaire
- Dramatique du point de vue de l'efficacité

Déforestation : Transformer $f \circ g$ en h,
où h ne construit plus de structure intermédiaire.

Plan de l'exposé

- La déforestation, c'est quoi ?
- **Principales méthodes en programmation fonctionnelle**
- Les grammaires attribuées comme alternative
- La composition symbolique
- Résultats - conclusions - perspectives

Déforestation en programmation fonctionnelle

Transformations par dépliage/pliage [Burstall&Darlington 77]

$\longrightarrow$ Algorithme de déforestation [Wadler 88]

Opérateurs algébriques de récursion [Meijer&Fokkinga&Paterson 91]

$\longrightarrow$ Règle élimination `foldr/build` [Gill&Launchbury&Peyton-Jones 93]

$\longrightarrow$ Algorithme de normalisation des *folds* [Sheard&Fegaras 93]

$\longrightarrow$ Hylomorphismes sous forme de triplets [Takano&Meijer 95]

$\longrightarrow$ Déf. rec. $\longrightarrow$ hylomorphismes [Hu&Iwasaki&Takeishi 96]

Déforestation par dépliage/pliage

$$\texttt{let}\ lms\ x = length\ (map\_square\ x)$$

$$\begin{array}{l} \texttt{let}\ lms\ x = \texttt{case}\ x\ \texttt{of} \qquad \text{dépliage} \\ \quad cons\ head\ tail \rightarrow \\ \qquad length\ (cons\ (square\ x)\ (map\_square\ tail)) \\ \quad nil \rightarrow length\ (nil) \end{array}$$

$$\begin{array}{ll} \texttt{let}\ lms\ x = \texttt{case}\ x\ \texttt{of} & \text{application aux constructeurs} \\ \quad cons\ head\ tail \rightarrow & \\ \qquad 1 + length\ (map\_square\ tail) & [\ \equiv 1 + (lms\ tail)\ ] \\ \quad nil \rightarrow 0 & \text{pliage} \end{array}$$

$$lms\ x \equiv length\ x$$

Méthodes algébriques

- Transformations guidées par la structure des données
  $\longrightarrow$ remplacer les constructeurs par l'effet de la fonction

- Foncteur $\equiv$ abstraction d'un schéma de récursion
  à la fois celui de la structure de données et celui de la fonction

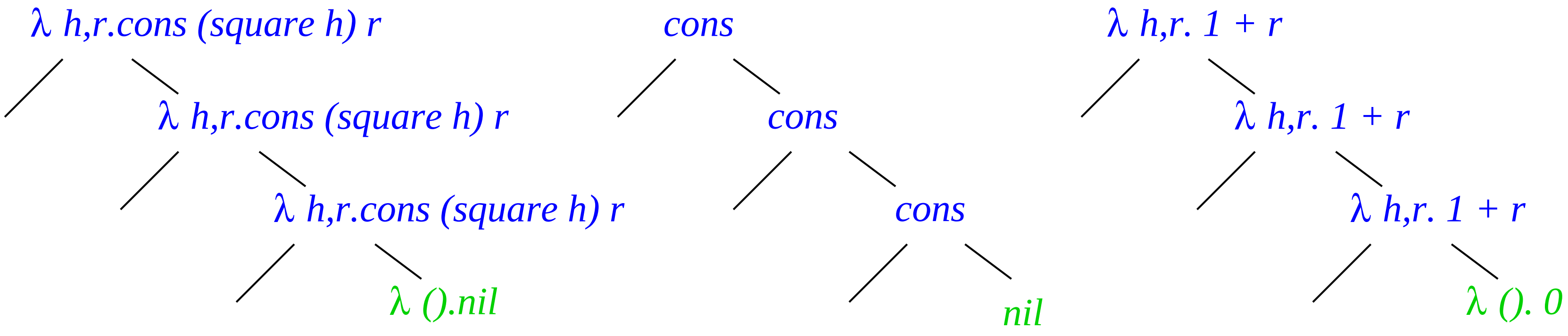
- Type $\Rightarrow$ Foncteur $\Rightarrow$ opérateur de contrôle générique

L'opérateur *fold* sur des listes

$$(fold^{list}\ f_n\ f_c)\ nil = f_n$$
$$(fold^{list}\ f_n\ f_c)\ cons\ head\ tail = f_c\ head\ ((fold^{list}\ f_n\ f_c)\ tail)$$

$$map\_square\ x = (fold^{list}\ f_n\ f_c)\ x$$
$$f_n = \lambda().nil$$
$$f_c = \lambda(h,r).cons\ (square\ h)\ r$$

$$length\ x = fold^{list}\ (\lambda().0,\ \lambda(a,r).1 + r)\ x$$



Normalisation des *folds* sur les listes

Théorème de Promotion [Sheard&Fegaras 93] :

la composition d'une fonction avec un *fold* produit un *fold*.

$$\frac{\begin{array}{rcl} \phi_n & = & g\ f_n \\ \phi_c\ h\ (g\ r) & = & g\ (f_c\ h\ r) \end{array}}{g\ ((fold^{list}\ f_n\ f_c)\ x) = (fold^{list}\ \phi_n\ \phi_c)\ x}$$

$length\ (map\_square\ x) = (fold^{list}\ \phi_n()\ \phi_c)\ x$ où

$\phi_n = \lambda().length\ nil$

$\phi_c = \lambda(h, r).1 + r$

$\boxed{\text{Hylomorphismes sous forme de triplets}}$

Représenter à la fois l'opérateur *fold* et son dual.
Représenter tous les types de manière homogène.

Théorie des catégories : type $\equiv$ objet initial d'une L-algèbre

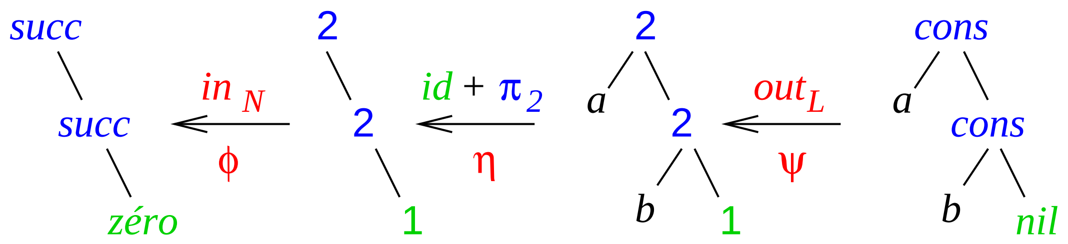
Foncteur des listes : $L = !1 + !A \times Id$

Idée : **Abstraire – Transformer – Construire**

$$[\![\phi, \eta, \psi]\!]_{G,F} \qquad f = \phi \circ (\eta \circ F f) \circ \psi$$

Fusion des hylomorphismes : *Acid Rain Theorem*

$$length = [\![in_N, id + \pi_2, out_L]\!]_{N,L}$$



Acid Rain Theorem [Takano&Meijer 95] : automatisation plus aisée.

$$\begin{array}{rl} & length \circ map\_square \\ = & [\![in_N, id + \pi_2, out_L]\!]_{N,L} \circ [\![in_L, id + (square \times id), out_L]\!]_{L,L} \\ = & [\![in_N \circ (id + \pi_2), id + (square \times id), out_L]\!]_{L,L} \\ = & [\![in_N, id + \pi_2, out_L]\!]_{N,L} \\ = & length \end{array}$$

Un problème pour les méthodes fonctionnelles

Déforestation des structures intermédiaires

construites dans des paramètres d'accumulation.

- Schéma récursion de la fonction $\neq$ schéma de la construction
  $\Rightarrow$ le foncteur ne suit plus fidèlement la construction

- Constructions "encapsulées" dans le paramètre
  $\Rightarrow$ difficiles à déforester par les méthodes fonctionnelles.

$$
\begin{array}{l}
\texttt{let } reverse\ x\ h = \texttt{case } x \texttt{ of} \\
\quad cons\ head\ tail \rightarrow \\
\qquad rev\ tail\ (cons\ head\ h) \\
\quad nil \rightarrow h
\end{array}
$$

$\boxed{\text{Plan de l'exposé}}$

– La déforestation, c'est quoi ?

– Principales méthodes en programmation fonctionnelle

– **Les grammaires attribuées comme alternative**

– La composition symbolique

– Résultats - conclusions - perspectives

Grammaires attribuées [Knuth 1968]

- **Spécification** : déclarative, dirigée par une grammaire
  (équations orientées sur chaque production)
  (attributs hérités / synthétisés)

- **Évaluation** : résolution d'un système d'équations
  (dépendances $\longrightarrow$ séquences de visites)

- **Transformation** : indépendante de l'évaluation
  Composition descriptionnelle [Ganzinger, Giegerich 84-88]

Des grammaires attribuées *fonctionnelles*

- Constructeur d'un type algébrique
  $\Rightarrow$ production de grammaire.

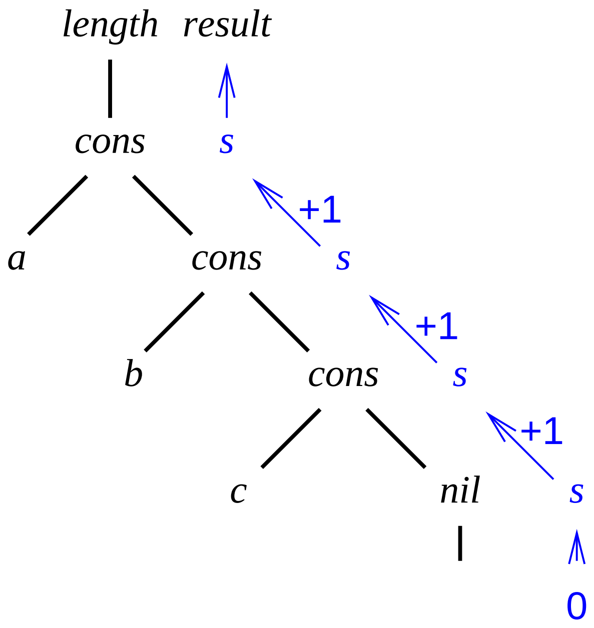
- Appel de fonction
  $\Rightarrow$ constructeur particulier (profil).

- Paramètres du profil
  $\Rightarrow$ attributs hérités sur la structure.

- Structures de données "virtuelles"
  $\Rightarrow$ grammaires attribuées dynamiques.

Longueur d'une liste : length

$$
\begin{array}{l}
\texttt{aglet}\ length = \\
\quad length\ x \longrightarrow \\
\quad\quad length.result = x.s \\
\quad cons\ head\ tail \longrightarrow \\
\quad\quad cons.s = 1 + tail.s \\
\quad nil \longrightarrow \\
\quad\quad nil.s = 0
\end{array}
$$



Inversion d'une liste : rev

$\texttt{aglet}\ rev =$

$\quad rev\ x\ h \rightarrow$

$\quad\quad reverse.result = x.rev$

$\quad\quad x.h = h$

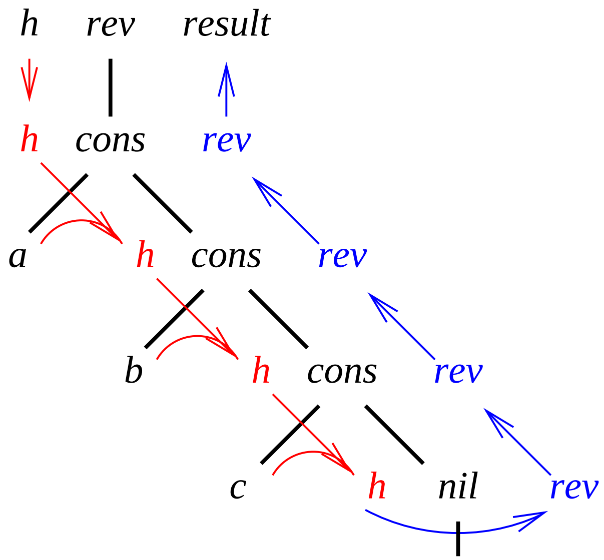
$\quad cons\ head\ tail \rightarrow$

$\quad\quad cons.rev = tail.rev$

$\quad\quad tail.h = cons\ head\ cons.h$

$\quad nil \rightarrow$

$\quad\quad nil.rev = nil.h$



Plan de l'exposé

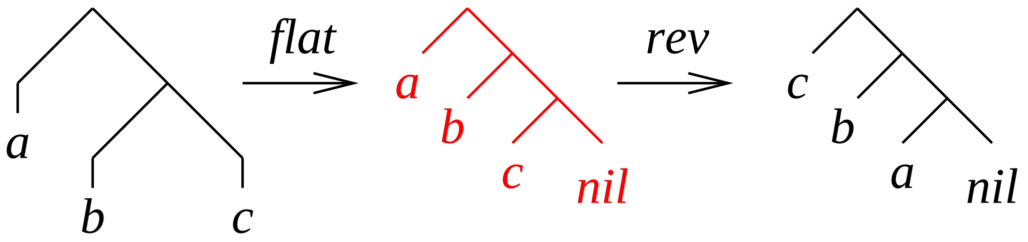
- La déforestation, c'est quoi ?
- Principales méthodes en programmation fonctionnelle
- Les grammaires attribuées comme alternative
- **La composition symbolique**
- Résultats - conclusions - perspectives

Une nouvelle méthode de déforestation

– Utiliser le formalisme des grammaires attribuées

– Améliorer le mécanisme de la composition descriptionnelle

FP-to-AG $\rightarrow$ composition symbolique $\rightarrow$ évaluateur fonctionnel



Transformation FP-to-AG

$\texttt{let}\ flat\ t\ l = \texttt{case}\ t\ \texttt{of}$
$\quad node\ left\ right \rightarrow$
$\quad\quad flat\ left\ (flat\ right\ l)$
$\quad leaf\ n \rightarrow$
$\quad\quad cons\ n\ l$

Transformation automatique :
paramètres d'appel
$\Rightarrow$ attributs hérités du profil
appels récursifs
$\Rightarrow$ occurrences d'attributs

$\texttt{aglet}\ flat =$
$\quad flat\ t\ l\ \rightarrow$
$\quad\quad flat.result = t.flat$
$\quad\quad t.l = l$
$\quad node\ left\ right \rightarrow$
$\quad\quad node.flat = (flat\ left\ (flat\ right\ l).result)).result$
$\quad leaf\ n \rightarrow$
$\quad\quad leaf.flat = cons\ n\ leaf.l$

$\boxed{\text{Évaluation symbolique}}$

Projection automatique des calculs d'un constructeur :
récursion explicite $\Rightarrow$ **système d'équations**.

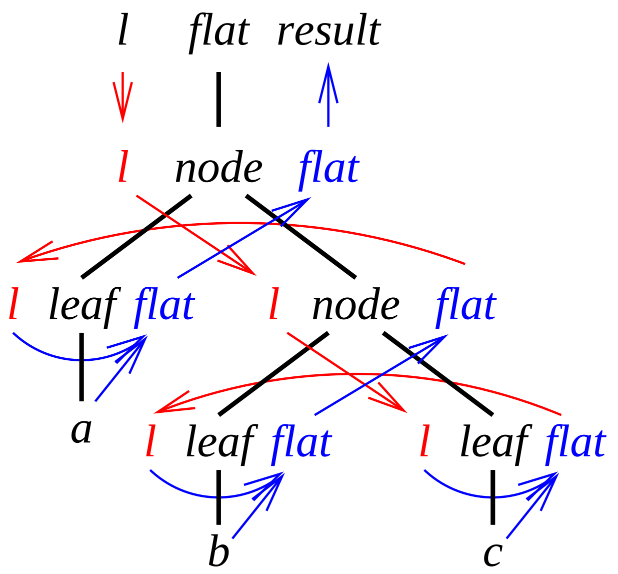
$$
\begin{array}{l}
flat\ t\ l\ \rightarrow \\
\quad flat.result = t.flat \\
\quad t.l = l
\end{array}
$$

$$
u.a = (flat\ right\ l).result \quad \Rightarrow \quad \begin{array}{l} u.a = right.flat \\ right.l = l \end{array}
$$

Transformation FP-to-AG sur *flat*

Finalement, FP-to-AG génère la **grammaire attribuée** :

$$
\begin{array}{l}
\texttt{aglet}\ flat =\\
\quad flat\ t\ l \rightarrow\\
\qquad flat.result = t.flat\\
\qquad t.l = l\\
\quad node\ left\ right \rightarrow\\
\qquad node.flat = left.flat\\
\qquad left.l = right.flat\\
\qquad right.l = node.l\\
\quad leaf\ n \rightarrow\\
\qquad leaf.flat = cons\ n\ leaf.l
\end{array}
$$



$\boxed{\text{Évaluation partielle}}$

L'évaluation symbolique effectue de l'évaluation partielle sur les termes clos finis.

Soit le terme fonctionnel $u = rev\ \ (cons\ a\ (cons\ b\ nil))\ \ nil$

FP-to-AG donne :

$$u.result = (cons\ \ a\ \ (cons\ \ b\ \ nil)).rev$$
$$(cons\ \ a\ \ (cons\ \ b\ \ nil)).h = nil$$

Un pas d'évaluation symbolique donne :

$$u.result = (cons\ \ b\ \ nil).rev$$
$$(cons\ \ b\ \ nil).h = cons\ \ a\ \ nil$$

Après deux pas de plus :

$$u.result = (cons\ \ b\ \ (cons\ \ a\ \ nil))$$

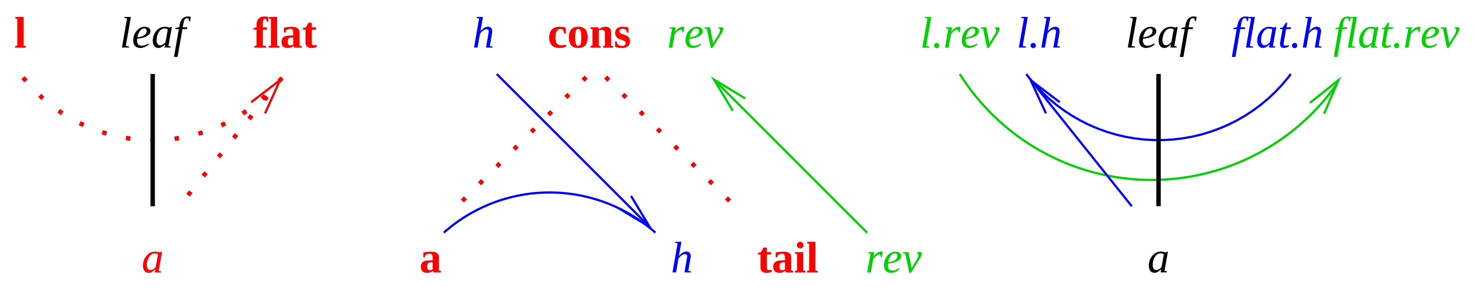
Composition symbolique : l'idée

Même principe que la composition descriptionnelle

Composition de rev avec $flat$:

$$revflat\ t\ l\ h = rev\ (flat\ t\ l)\ h$$

Principe : **projeter** les calculs de rev sur les constructions de $flat$



Composition symbolique : mise en œuvre

Pour la fonction $revflat\ t\ l\ h = rev\ (flat\ t\ l)\ h$

La transformation FP-to-AG génère

d'après le profil de $flat$:

$$\begin{aligned}&revflat\ t\ l\ h \rightarrow\\&\quad revflat.result = rev\ t.flat\ h\\&\quad t.l = l\end{aligned}$$

d'après le profil de rev :

$$\begin{aligned}&revflat\ t\ l\ h \rightarrow\\&\quad revflat.result = (t.flat).rev\\&\quad (t.flat).h = h\\&\quad t.l = l\end{aligned}$$

Composition symbolique : projection

Projection systématique des noms des attributs

consommateurs (rev et h) sur les équations

définissant les attributs producteurs ($flat$ et l).

Exemple pour $leaf$:

$$\begin{array}{l} leaf\ n \rightarrow \\ \quad leaf.flat = cons\ n\ leaf.l \end{array}$$

devient (**projection**)

$$\begin{array}{lll} leaf\ n \rightarrow & & \\ \quad (leaf.flat).rev = (cons\ n\ leaf.l).rev & & rev \text{ synthétisé} \\ \quad (cons\ n\ leaf.l).h = (leaf.flat).h & & h \text{ hérité} \end{array}$$

Composition symbolique : évaluation symbolique

$$
\begin{array}{l}
leaf\ n \rightarrow \\
\quad (leaf.flat).rev = (cons\ n\ leaf.l).rev \\
\quad (cons\ n\ leaf.l).h = (leaf.flat).h
\end{array}
$$

devient (**évaluation symbolique**)

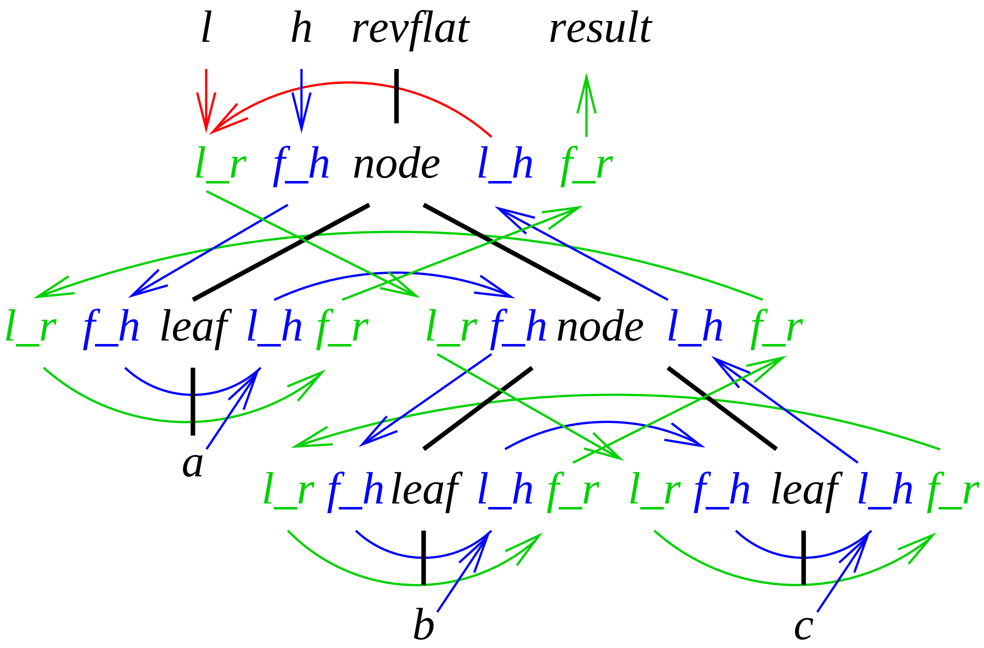
$$
\begin{array}{l}
leaf\ n \rightarrow \\
\quad (leaf.flat).rev = (leaf.l).rev \\
\quad (leaf.l).h = cons\ n\ (leaf.flat).h
\end{array}
$$

Un simple renommage [$(x.a).b$ en $x.a\_b$] produit :

$$
\begin{array}{l}
leaf\ n \rightarrow \\
\quad leaf.flat\_rev = leaf.l\_rev \\
\quad leaf.l\_h = cons\ n\ leaf.flat\_h
\end{array}
$$

Résultat : *revflat* déforesté

$$
\begin{array}{l}
\texttt{let } revflat = \\
\quad revflat\ t\ l\ h \rightarrow \\
\qquad revflat.result = t.flat\_rev \\
\qquad t.flat\_h = h \\
\qquad t.l\_rev = (rev\ l\ t.l\_h) \\
\quad node\ left\ right \rightarrow \\
\qquad \ldots
\end{array}
$$



$$\boxed{\text{Programme fonctionnel correspondant}}$$

$$\texttt{let}\ revflat\ t\ l\ h = f2\ t\ (rev\ l\ (f1\ t\ h))$$

$$
\begin{array}{l}
\texttt{let}\ f1\ t\ h = \texttt{case}\ t\ \texttt{of} \\
\quad node\ left\ right \rightarrow \\
\qquad f1\ right\ (f1\ left\ h) \\
\quad leaf\ n \rightarrow cons\ n\ h
\end{array}
$$

$$
\begin{array}{l}
\texttt{let}\ f2\ t\ l = \texttt{case}\ t\ \texttt{of} \\
\quad node\ left\ right \rightarrow \\
\qquad f2\ left\ (f2\ right\ l) \\
\quad leaf\ n \rightarrow l
\end{array}
$$

$\boxed{\text{Le coût de cette déforestation}}$

- FP-to-AG : transformation linéaire en fonction de la taille et de la profondeur du terme fonctionnel.

- Composition symbolique : transformation quadratique en nombre d'attributs ($m * n$).
  Attention : complexité de l'évaluation partielle ?

- Difficile de comparer avec les méthodes de déforestation fonctionnelles (pas abordé pour *folds* et *hylos*).

Composition symbolique : résultats

- **Résolution d'une classe de problèmes durs**
  $\Rightarrow$ Déforestation dans les paramètres d'accumulation
  $\longrightarrow$ plus de problème de fidélité foncteur/schéma de calcul
  $\longrightarrow$ formalisme homogène (hérités/synthétisés).
  $\longrightarrow$ cas simples traités de la même manière.

- **Extension de la composition descriptionnelle**
  $\longrightarrow$ plus seulement sur deux grammaires isolées
  $\longrightarrow$ gestion des paramètres, évaluation partielle

- **Réutilisation des analyses statiques et optimisations**
  $\longrightarrow$ élimination des règles de copies

Conclusion

- **Comparaison de styles de programmation (formalismes)**
  similitudes / différences de notations, d'expressivité
  $\Rightarrow$ améliorations : grammaires attribuées dynamiques.

- **Comparaison de différentes techniques de déforestation**
  efficacité / limites des méthodes, intrinsèques ?
  $\Rightarrow$ améliorations : composition symbolique.

- **Formalisme des grammaires attribuées**
  particulièrement adapté pour ce type de transformations
  $\Rightarrow$ représentation intérmédiaire

Perspectives

- Formalisation des classes de programmes acceptées

- Transfert de la technique en termes fonctionnels

- Extension de la comparaison
  $\Rightarrow$ évaluation partielle - *supercompiler* - ordre supérieur

- Utilisation de la composition symbolique (déforestation)
  $\Rightarrow$ généricité, spécialisation de programmes

“Moi qui ai tant parlé, avec le désir insatiable de convaincre”

Paul Valéry

Fold sur des listes (2)

Fold de second ordre pour $reverse$ avec accumulation

$$\begin{aligned} reverse\ x &= (fold^{list}\ f_n\ f_c)\ x\ nil \\ f_n &= \lambda().\lambda(w).w \\ f_c &= \lambda(a,r).\lambda(w).r\ (cons\ h\ w) \end{aligned}$$

Acid Rain Theorem

Cata-HyloFusion :

$$\frac{\tau : \forall A.(F\,A \to A) \to F'\,A \to A}{[\![\phi, \eta_1, out_F]\!]_{G,F} \circ [\![\tau\; in_F, \eta_2, \psi]\!]_{F',L} = [\![\tau(\phi \circ \eta_1), \eta_2, \psi]\!]_{F',L}}$$

Hylo-AnaFusion :

$$\frac{\sigma : \forall A.(A \to F\,A) \to A \to F'\,A}{[\![\phi, \eta_1, \sigma\; out_F]\!]_{G,F'} \circ [\![in_F, \eta_2, \psi]\!]_{F,L} = [\![\phi, \eta_1, \sigma(\eta_2 \circ \psi)]\!]_{G,F'}}$$