Symbolic composition
Deforestation/ program transformations
How to deforest in accumulative parameters ?

Didier Parigot

work with Etienne Duris, Loic Correnson and Gilles Roussel

http://www-rocq.inria.fr/oscar/www/fnc2/
Didier.Parigot@inria.fr

INRIA

$\boxed{\text{General context of this work}}$

Compare different programming styles

and their approaches to solve different problems

Goal : To find new solutions (a cross-fertilization)

- declarative programming, directed by data structure

$\Rightarrow$ the expressive power of specifications

$\Rightarrow$ program transformations (deforestation)

Research Works carried out in team (1)

Attribute Grammars / Functional Programming

the expressive power

- **Attribute grammars**
  - $\longrightarrow$ specifications on concrete data structures.
  - $\longrightarrow$ static semantics.
- **Dynamic attribute grammars**
  - $\longrightarrow$ example : Factorial functional.
  - $\longrightarrow$ specifications on **abstract** data structures.
  - $\longrightarrow$ First order functional language.

Research Works carried out in team (2)

Attribute Grammars / Functional Programming

Program transformations / Deforestation

- **The functional deforestation methods.**

  $\longrightarrow$ presentation

  $\longrightarrow$ comparison

  $\longrightarrow$ limits.

- **Symbolic composition**

  $\longrightarrow$ extends the descriptional composition of attribute grammars.

  $\longrightarrow$ usable for functional languages.

  $\longrightarrow$ solves a class of hard problems.

$\boxed{\text{Outline of the talk}}$

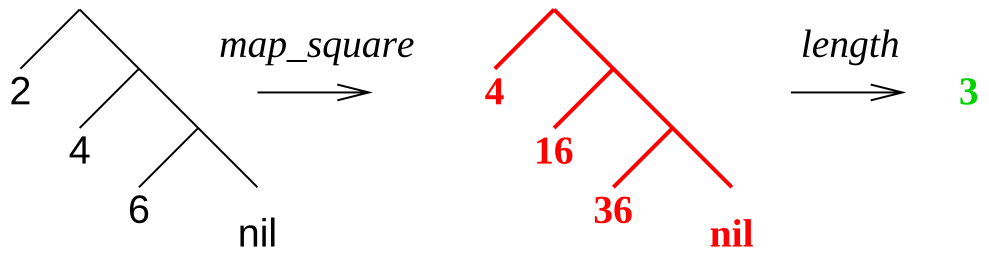
- The **deforestation** method, What is it ?
- The basic concepts of **deforestation** methods
- **Attribute grammars** as an alternative
- The **symbolic composition**
- Results and conclusions

First example: The intermediate data structures

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

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

If we apply the composition $length \circ map\_square$ on an list...



$\boxed{\text{Deforestation}}$

In some compositions

$$f \circ g$$

the function g produces an intermediate data structure which is consumed by the function f.

- Fundamental for modular programs
- harmful cost from efficiency point of view

Deforestation : Transform $f \circ g$ into new function h, where h does not build anymore the intermediate data structure.

Outline of the talk

- The **deforestation**, What it is ?
- The basic concepts of **deforestation** methods
- The **attribute grammars** like alternative
- The **symbolic composition**
- Results and conclusions

Deforestation on functional programming

Transformations by unfold/fold step[Burstall&Darlington 77]

$\longrightarrow$ Deforestation Algorithm [Wadler 88]

Algebraic operator [Meijer&Fokkinga&Paterson 91]

$\longrightarrow$ **`foldr/build`** elimination rules [Gill&Launchbury&Peyton-Jones 93]

$\longrightarrow$ Algorithm of normalization *folds* [Sheard&Fegaras 93]

$\longrightarrow$ Hylomorphisms in triplet form [Takano&Meijer 95]

$\longrightarrow$ Def. rec. $\longrightarrow$ hylomorphisms [Hu&Iwasaki&Takeishi 96]

Deforestation by unfolding/folding

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

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

$$\begin{aligned}&\texttt{let}\ lms\ x = \texttt{case}\ x\ \texttt{of} \quad \text{application to constructors}\\ &\quad cons\ head\ tail \rightarrow\\ &\qquad 1 + length\ (map\_square\ tail) \qquad [\ \equiv 1 + (lms\ tail)\ ]\\ &\quad nil \rightarrow 0 \qquad\qquad \text{folding}\end{aligned}$$

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

Algebraic methods

- Functor notion $\equiv$ abstraction of recursion schemes
  both that of the data structure and that of the function

- Type $\Rightarrow$ Functor $\Rightarrow$ generic control operator
  fold operator

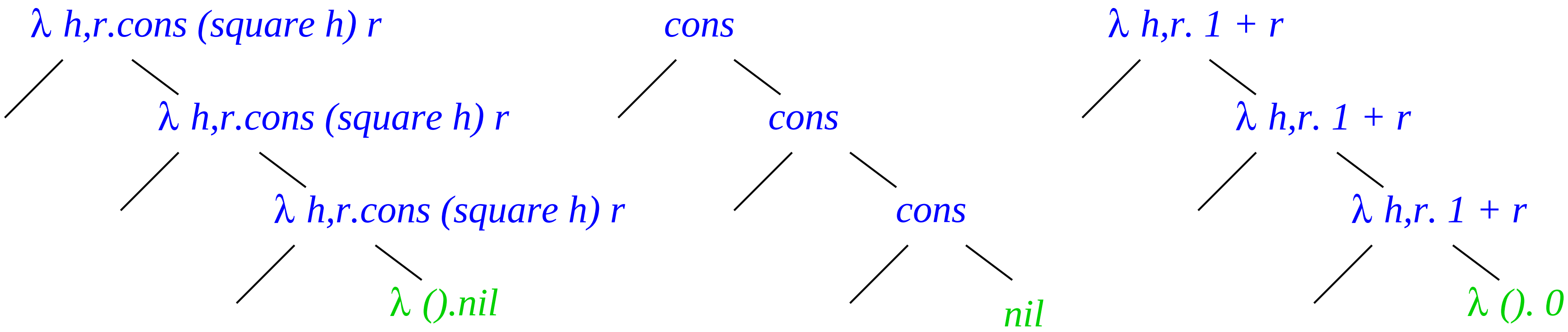
- Transformation is driven by the data structure
  $\longrightarrow$ to replace the constructor by the effect of the function

The *fold* operator on the list

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

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

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



folds normalization algorithm for the type List

Promotion Theorem [Sheard&Fegaras 93] :

The composition of a function with a *fold* (operator) produces a *fold*.

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

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

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

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

$\boxed{\text{The problems of the functional methods (1)}}$

- Folding transformation and Termination

- Generic control operator ($fold^{list}$)
  polynomial type

- More abstract method : hylomorphisms in triplet form

The problems of the functional methods (2)

- Constructions "encapsulated" in the parameters
  $\Rightarrow$ difficult to deforest using the functional methods.

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

Outline of the talk

- The **deforestation**, What it is ?
- The basic concepts of **deforestation** methods
- The **attribute grammars** a good alternative
- The **symbolic composition**
- Results and conclusions

$\boxed{\text{Attribute Grammars [Knuth 1968]}}$

- **Specification** : declarative, structured by the grammar (oriented equations on each production) (inherited/synthesized attributes)

- **Evaluation methods** : resolution of equation system (dependences $\longrightarrow$ visits sequences)

- **Transformation** : independent of the evaluation method **Descriptional composition** [Ganzinger, Giegerich 84-88]

Functional attribute grammars

- Constructor of algebraic type
  $\Rightarrow$ production of context-free grammar.

- Function Call
  $\Rightarrow$ particular constructor (profile).

- Parameters of the profile (function)
  $\Rightarrow$ inherited attributes on the data structure.

- Abstract data structures
  $\Rightarrow$ Dynamic attribute grammars.

$\boxed{\text{Length of a list: length}}$

aglet $length$ =

$\quad length\ x \longrightarrow$

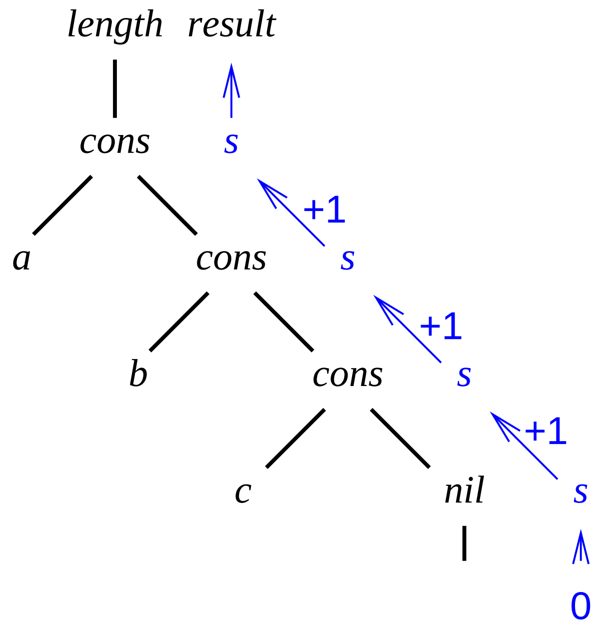
$\quad\quad length.result = x.s$

$\quad cons\ head\ tail \longrightarrow$

$\quad\quad cons.s = 1 + tail.s$

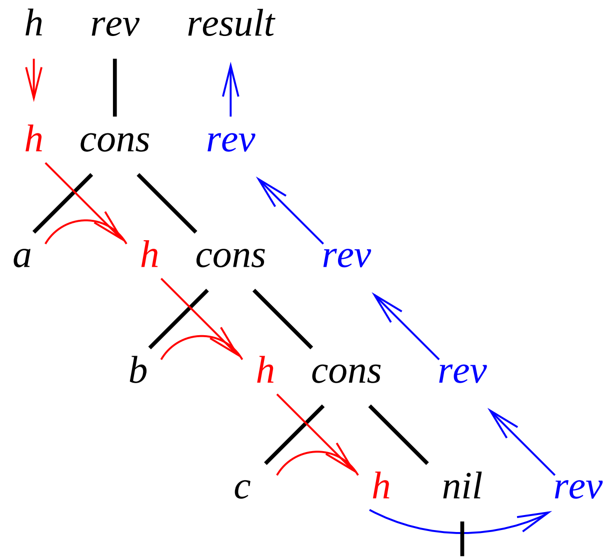
$\quad nil \longrightarrow$

$\quad\quad nil.s = 0$



Reverse of a list : rev

$$
\begin{array}{l}
\texttt{aglet}\ rev = \\
\quad rev\ x\ h \rightarrow \\
\qquad rev.result = x.\color{blue}{rev} \\
\qquad x.\color{red}{h} = h \\
\quad cons\ head\ tail \rightarrow \\
\qquad cons.\color{blue}{rev} = tail.\color{blue}{rev} \\
\qquad tail.\color{red}{h} = cons\ head\ cons.\color{red}{h} \\
\quad nil \rightarrow \\
\qquad nil.\color{blue}{rev} = nil.\color{red}{h}
\end{array}
$$



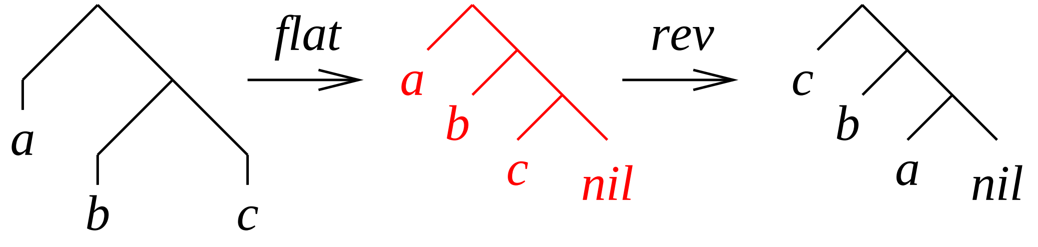
$\boxed{\text{Outline of the talk}}$

- The **deforestation**, What it is ?
- The basic concepts of **deforestation** methods
- The **attribute grammars** as good alternative
- The **symbolic composition**
- Results and conclusions

A new deforestation method : Symbolic composition

- to use the formalism of attribute grammar
- to improve the descriptional composition
- to transpose into the functional framework

FP-to-AG → Symbolic composition → functional evaluator



Transformation FP-to-AG : preliminary transformation

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

Automatic transformation :
Parameters of the function call
$\Rightarrow$ inherited attributes of the profile
Recursive function calls
$\Rightarrow$ attribute occurrences $t.flat$

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

Symbolic evaluation

Automatic projection of calculations of a constructor :
breaking explicit recursion ⇒ equation system.

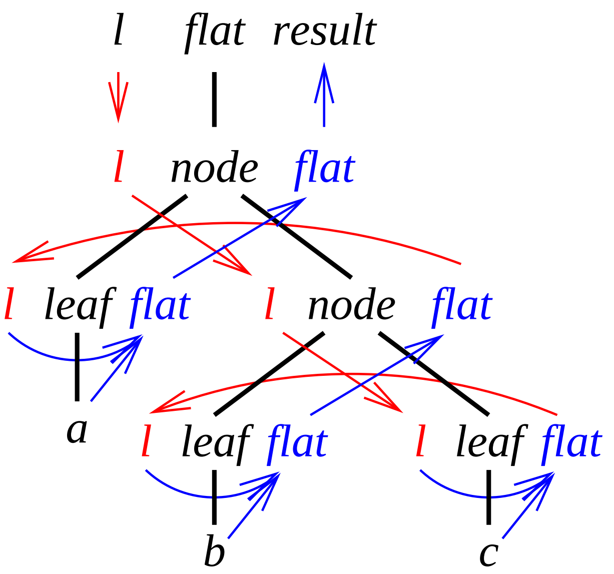
$$
\begin{array}{l}
flat\ t\ l \ \rightarrow \\
\quad flat.result = t.flat \\
\quad t.l = l
\end{array}
$$

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

Transformation FP-to-AG for the function *flat*

FP-to-AG produces the following **attribute grammar**:

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



Partial evaluation

Symbolic evaluation performs partial evaluation on finite ground terms.

let $u = rev\ ((cons\ a\ (cons\ b\ nil))\ \ nil)$ be a functional term

FP-to-AG gives:

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

one step of symbolic evaluation gives:

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

after two more steps:

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

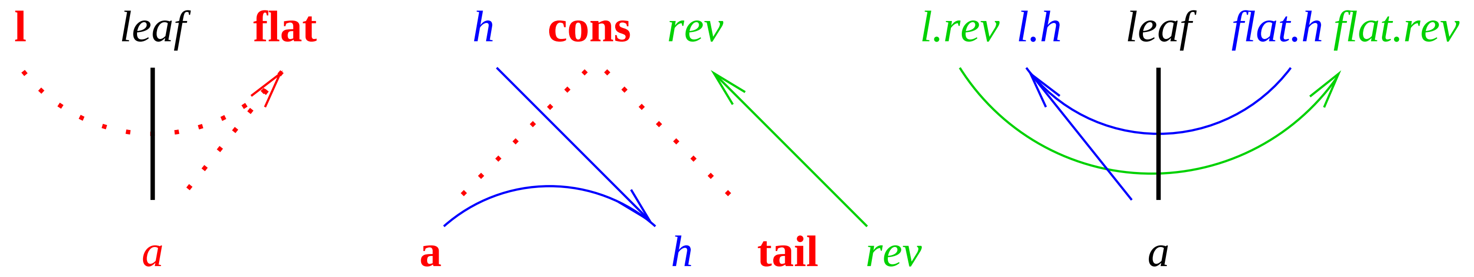
Symbolic composition: the basic idea

Same basic principle as the descriptional composition

Composition of function rev with function $flat$:

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

Principle : **projection** of the calculations of rev on the constructions of $flat$



Symbolic composition

For the function $revflat\ t\ l\ h = rev\ (flat\ t\ l)\ h$

The transformation FP-to-AG gives

according to the profile of $flat$:

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

according to the profile of rev :

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

$\boxed{\text{Symbolic composition : projection step}}$

Systematic projection of attribute names :

consumer attributes (rev and h) on the equations

defining the producer attributes ($flat$ and l).

Example for constructor $leaf$:

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

becomes (by **projection step**)

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

$\boxed{\text{Symbolic composition : symbolic evaluation}}$

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

becomes (by **symbolic evaluation**)

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

According to definition of function rev with some substitutions

$$
\begin{array}{ll}
cons\ head\ tail \rightarrow & \\
\quad cons.rev = tail.rev & [tail = leaf.l] \\
\quad tail.h = cons\ head\ cons.h & [cons = leaf.flat]
\end{array}
$$

$\boxed{\text{Symbolic composition : a simple renaming}}$

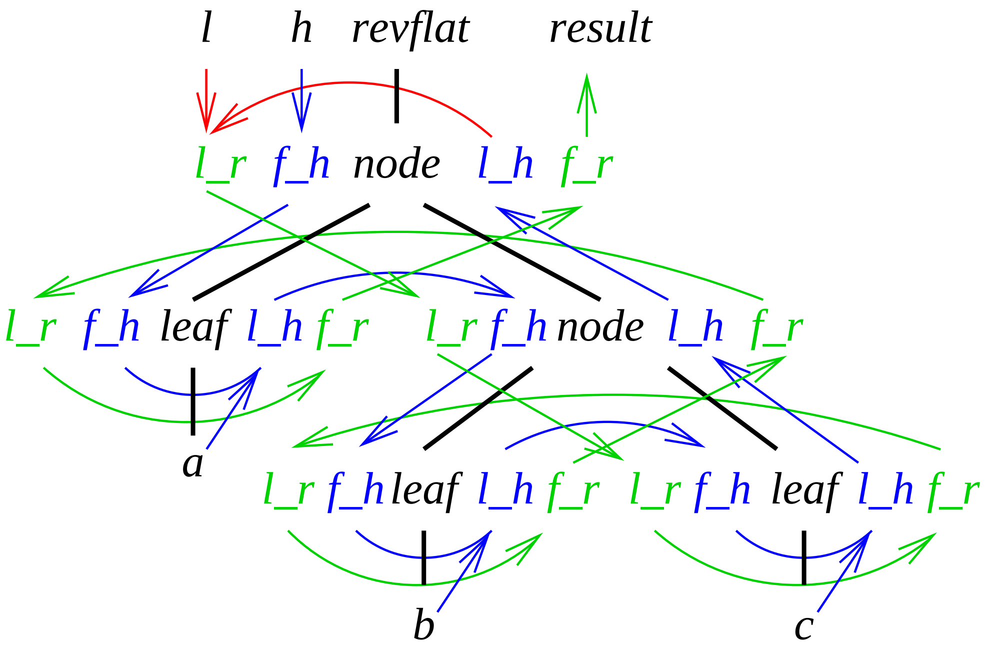
$$
\begin{array}{l}
leaf\ \ n \rightarrow \\
\quad (leaf.flat).rev = (leaf.l).rev \\
\quad (leaf.l).h = cons\ \ n\ \ (leaf.flat).h
\end{array}
$$

A simple renaming [$(x.a).b$ to $x.a\_b$] produces :

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

Result : *revflat* deforested

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



Corresponding functional program of *revflat*

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

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

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

Symbolic composition: results

- **Resolution of a class of hard problems**
  $\Rightarrow$ deforestation in accumulative parameters
  $\longrightarrow$ no more problem of fidelity functor/scheme of construction
  $\longrightarrow$ homogeneous formalism (inherited/synthesized).
  $\longrightarrow$ the simple case are handled in a same manner.

- **Extension of descriptional composition**
  $\longrightarrow$ not only on two distinct attribute grammars
  $\longrightarrow$ with parameters, partial evaluation

- **Re-use of the static analyses and optimizations of AG**
  $\longrightarrow$ elimination of copy rules

Conclusion

- **Formalism of attribute grammars**
  particularly suitable for program transformations
  $\Rightarrow$ a "good" abstract representation of programs

- There exists a first implementation by Loic Correnson
  $\longrightarrow$ **Equational Semantics**
  $\longrightarrow$ Dynamic attribute grammar
  $\longrightarrow$ Partial evaluation
  $\longrightarrow$ Higher order functions

- **Generic programming**
  $\longrightarrow$ polytypic programming [Jansson97]
  $\longrightarrow$ adaptive programming [Lieberherr]

$\boxed{\text{Informations}}$

- The Fnc-2 attribut grammar system
  http://www-rocq.inria.fr/oscar/www/fnc2/

- Publications of team
  http://www-rocq.inria.fr/oscar/www/fnc2/recentpaper.html

- Bibliography on Attribute Grammars
  http://www-rocq.inria.fr/oscar/www/fnc2/AGabstract.html

- Second International Workshop on Attribute Grammars and their Applications WAGA'99
  http://www-rocq.inria.fr/oscar/waga99.html