Programmation Fonctionnelle Avancée
7 : Combinators

Ralf Treinen

Université Paris Diderot
UFR Informatique
Institut de Recherche en Informatique Fondamentale

PARIS DIDEROT

treinen@irif.fr

31 octobre 2018

© Roberto Di Cosmo et Ralf Treinen

1978 : Turing Award pour John Backus, créateur de Fortran

- En 1978, John Backus, le créateur de Fortran, et de la BNF, reçoit le Turing Award.
- La leçon associée à ce prix s'intitule *Can programming be liberated from the von Neumann style ? : a functional style and its algebra of programs*. On y trouve des idées fondatrices.
- "Functional programs deal with *structured data*, ... *do not name their arguments*, and do not require the complex machinery of procedure declarations ..."
- "*do not name their arguments*" : calculer des fonctions par combinaisons de fonctions données, un style de programmation très défendu par Backus.

Backus, continué

- *Associated with the functional style of programming is an algebra of programs ... [that] can be used to transform programs ... .*
- *These transformations are given by algebraic laws and are carried out in the same language in which programs are written.*
- *Combining forms are chosen not only for their programming power but also for the power of their associated algebraic laws.*

Un exemple de Backus : combinateurs de listes

- Backus définit le produit scalaire (*inner product* en anglais) comme suit :

$$\mathit{Def}\ \mathit{IP} \equiv\ (\mathit{List.fold}\ +) \circ (\mathit{List.map2}\ \times) \circ \mathit{Transpose}$$

- Cette définition ne fait que *combiner* des opérations plus élémentaires (qu'on appelle des *combinateurs*, ici *List.fold*, *List.map2*, ∘ et *Transpose*).
- Dans son formalisme, l'argument est une liste de deux vecteurs. La fonction Transpose la transforme en une liste de pairs.
- Sa version de List.fold (appelée *Insert*) agit sur des listes non vides, donc pas besoin de valeur initiale.
- Il propose aussi un formalisme ultra concis :

$$\mathrm{IP} = (/+) \circ (\alpha\times) \circ \mathrm{Trans}$$

Pointless Programming

- De plus, ces combinateurs sont assemblés par simple composition fonctionnelles, sans nommer explicitement leur paramètres : ce style d'assemblage de fonctions va sous le nom de *programmation sans points*, ou *pointless programming*.
- La même idée est derrière les *tubes* (*pipes*) en UNIX :

> *Pipes facilitated function composition on the command line. You could take an input, perform some transformation on it, and then pipe the output into another program. This provided a very powerful way of quickly creating new functionality with simple composition of programs. People started thinking how to solve problems along these lines.*

Alfred Aho, *Masterminds of Programming*

Pointless Programming in OCaml

- La value restriction en OCaml ne permet pas d'utiliser ce style en toute généralité
- Elle est nécessaire car OCaml n'est pas purement fonctionnel, comme nous avons vu.
- Voir le cours 8 pour une explication

Exemples (pointless.ml)

```
(* value restriction gets in the way of the
   pointless programming style *)

let compose f g = fun x -> f (g x);;

let myid = compose (fun x -> x) (fun x -> x);;

myid 42;;

myid "coocoo";;

(* explication au cours8 **
```


Quelques exemples de Backus : transformations

- John Backus est visionnaire aussi dans sa présentation de transformations de programmes qui sont valides dans un formalisme purement fonctionnel : son article en mentionne une grande quantité.
- En particulier, on retrouve l'équation III.4, qui se lit, en notation moderne, comme :

$$List.map(f \circ g) = (List.map\ f) \circ (List.map\ g)$$

- Cette équation, utilisée de droite à gauche, décrit une transformation qui est l'analogue de la *fusion de boucle* en programmation impérative : elle remplace deux visites d'une liste par une seule visite, et fournit donc une *optimisation* du code.

Re-visitons les opérations sur les listes...

- On utilise souvent les fonctions *List .map* et *List . fold_left* .
- Ce sont des *combinateurs* pour les opérations sur les listes!

Exemples (lists1.ml)

```
let rec map f = function
  | [] -> []
  | a::l -> let r = f a in r :: map f l;;

let rec fold_left f accu l =
  match l with
  | [] -> accu
  | a::l -> fold_left f (f accu a) l;;
```


Des combinateurs pour les listes

Prenons le temps de réfléchir à leur signification :

- on a besoin de la récursion pour les définir, *pas pour les utiliser*
- permettent d'écrire des programmes sans points, par *simple composition*
- capturent un schéma général réutilisable
- permettent de définir des transformations génériques intéressantes comme celle indiquée par Backus

Combinateurs sur les listes

- Bird et Meertens ont défini un formalisme puissant pour manipuler des listes, basé sur les combinateurs *length*, *append*, *map*, *filter* , *fold_left* et *fold_right*.
- Avec ces combinateurs, il est possible de définir un grand nombre d'opérations sur les listes.
- Par exemple, pour concatener tous les éléments d'une liste : concat = fold_left append []
- Voyons ce qu'on peut faire avec juste *map* et *fold_left* !

Homomorphismes de Listes

- *Définition :* Une fonction h est un *homomorphisme de liste* si il existe un opérateur *associatif* $\oplus$ avec élément neutre e tel que :

$$\begin{array}{rcl} h\ [] & = & e \\ h\ (l_1 @ l_2) & = & (h\ l_1) \oplus (h\ l_2) \end{array}$$

- Exemples :
  - la recherche de l'élément le plus grand ($\oplus = max$ et $e = -\infty$)
  - le tri ($\oplus$ le merge du mergesort, $e = []$)

List Homomorphism Lemma (Bird, 1986)

- *List homomorphism lemma* : Une fonction h est un homomorphisme de liste ssi il existe un couple de fonctions f et g telle que h peut s'écrire comme

$$h = (reduce\ f) \circ (map\ g)$$

- Quand $\oplus$ est associatif, et dispose d'un élément neutre, fold_left et fold_right coïncident et on les appelle *reduce*.

Application importante : exécution parallèle

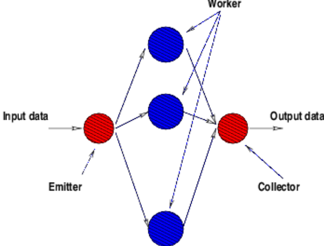
- Le résultat qu'on vient de voir trouve des nombreuses applications aujourd'hui parce que map et reduce peuvent être parallélisées *très facilement*. Les systèmes MapReduce de Google ou Hadoop de Apache sont basés sur ce fait.
- Le *speedup* d'un algorithme parallèle

$$S_n = \frac{T_1}{T_n}$$

  où T_1 est le temps de calcul du meilleur algorithme séquentiel sur une seule machine, et T_n est le temps de calcul sur n machines.
- Dans le cas idéal, on aimerait avoir $S_n = n$, et on parle de speedup *linéaire*.

Exécution parallèle d'un map

- Dans les *conditions idéales*[1], on peut calculer $map\ f\ l$ sur une liste l de longueur n en effectuant le calcul pour chaque élément en parallèle sur chacun des n processeurs.

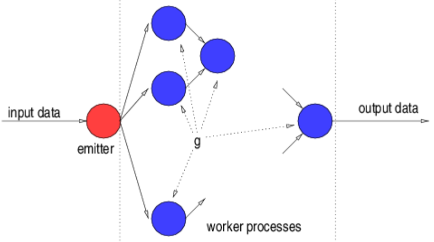


- On a alors $S_n = \frac{T_1}{(T_1/n)} = n$, le meilleur résultat possible.

1. Le temps de calcul pour chaque élément est uniforme ; le temps de communication est négligeable par rapport au temps de calcul, ...

Exécution parallèle d'un reduce

- Toujours dans les *conditions idéales*, on peut calculer *reduce* $\oplus$ *l* sur une liste de longueur n, avec un arbre binaire équilibré complet, en seulement $log\ n$ étapes.



Exécution parallèle d'un reduce : une minute de réflection

- Le temps de calcul parallèle est de toute façon *au minimum* de l'ordre de $log\ n$.
- Donc, il convient d'envoyer aux feuilles de l'arbre déjà des segments de liste de taille $log\ n$: le calcul sera fait plus vite en séquentiel (il n'y a pas d'overhead de communication).
- On peut donc utiliser seulement $\frac{n}{log\ n}$ processeurs, et s'attendre à un temps de calcul de l'ordre de $log\ n$.
- On a donc $S_{n/log\ n} \approx \frac{n}{log\ n}$.

Google MapReduce, Apache Hadoop

- Ces observations sont à la base des implémentations massivement distribuées du paradigme *map/reduce* qui ont été popularisées par Google.
- Si on veut faire un calcul efficace en utilisant ces bibliothèques, il ne nous reste qu'à trouver le $\oplus$ et le e qui vont bien, en nous assurant de l'associativité de $\oplus$.
- Regardons un exemple intéressant où cette définition n'est pas si évidente.

Maximum segment sum (Cole, 1993)

- *Problème :* étant donnée une liste d'entiers, trouver le segment de cette liste ayant la plus grande somme.
- *Exemple :*

$$mss\ [2; -4; \underline{2; -1; 6}; -3] = 7$$

- Malheureusement, *mss* *n'est pas* un homomorphisme : si nous notons $a \uparrow b$ le maximum entre a et b, alors, en général :

$$mss(l_1 @ l_2) \neq (mss\ l_1) \uparrow (mss\ l_2)$$

- L'égalité vaut seulement si le mss de $(l_1 @ l_2)$ est entièrement dans l_1 ou dans l_2, et pas s'il est à cheval sur les deux listes.

Exemple

- Si on découpe $[2;-4;2;-1;6;-3]$ en $[2;-4;2]$ et $[-1;6;-3]$ on a *mss* $[2;-4;2]=2$ et *mss* $[-1;6;-3]=6$, mais $2\uparrow 6<7$.
- Pour couvrir le cas du *mss* à cheval sur les deux, on doit aussi prendre en compte une *maximum concluding sum* de l_1 et une *maximum initial sum* de l_2.
- Dans notre exemple :

$$\begin{array}{rcl} mcs[2;-4;\underline{2}] & = & 2 \\ mis[\underline{-1;6};-3] & = & 5 \end{array}$$

On a alors $mss(l_1 @ l_2) = mss\ l_1 \uparrow mss\ l_2 \uparrow (mcs\ l_1 + mis\ l_2)$

Calculer le maximum segment sum

- Mais alors, on a besoin de calculer aussi *mis* et *mcs*, et pour cela on aura besoin de garder aussi la somme totale d'une liste, qu'on notera *ts*.
- $$\begin{array}{rcl} mis\ (l_1 @ l_2) & = & (mis\ l_1) \uparrow (ts\ l_1 + mis\ l_2) \\ mcs\ (l_1 @ l_2) & = & (mcs\ l_2) \uparrow (mcs\ l_1 + ts\ l_2) \end{array}$$
- On utilise ces observations pour construire un homomorphisme *emms* (*extended mss*) qui *contient* notre fonction *mss*.
- La fonction *emms* va maintenir les valeurs de *mss*, *mis*, *mcs* et *ts* tout au long du calcul, et on *extrait* *mss* seulement à la fin.

Maximum segment sum par un homomorphisme

- On définit alors l'opération pour la phase *reduce* comme suit :

$$\begin{array}{rcl} (mss, mis, mcs, ts) & \oplus & (mss', mis', mcs', ts') = \\ & (& mss \uparrow mss' \uparrow (mcs + mis'), \\ & & mis \uparrow (ts + mis'), \\ & & mcs' \uparrow (mcs + ts'), \\ & & ts + ts' \quad) \end{array}$$

- Et l'opération pour la phase *map*, qui calcule ces quadruple sur chaque élément de la liste :

$$f\ x = (x \uparrow 0, x \uparrow 0, x \uparrow 0, x)$$

Maximum segment sum dans un homomorphisme

- Questions :
  - est-ce que $\oplus$ est bien associatif ?
  - quel est l'élément neutre de $\oplus$?
- Une fois cela vérifié, on peut définir alors

$$emms = reduce(\oplus) \circ map(f)$$

Exemples (mss.ml)

```
let op (mss,mis,mcs,ts) (mss',mis',mcs',ts') =
  max mss (max mss' (mcs+mis')),
  max mis (ts+mis'),
  max mcs' (mcs+ts'),
  ts+ts';;

let f x = (max 0 x, max 0 x, max 0 x, x);;

let mss l =
  let (v,_,_,_) =
    List.fold_left op (0,0,0,0) (List.map f l)
in v;;
mss [2;-4;2;-1;6;-3];;
```


Exemples (mssparmap.ml)

```
#use "topfind";;
#require "parmap";;

(* open Mss *)
open Parmap
let () = set_default_ncores 4;;
let () = debugging true;;

let mss l =
  let (v,_,_,_) = parmapfold f (L l) op (0,0,0,0) op
in v;;

mss [2;-4;2;-1;6;-3];;
```


La fonction `parmapfold` de la bibliothèque `parmap`

```
type 'a sequence = L of 'a list | A of 'a array;;
val parmapfold : ('a -> 'b) -> 'a sequence
  -> ('b-> 'c -> 'c) -> 'c -> ('c->'c->'c) -> 'c
```

(voir la doc pour les arguments optionnels)

```
parmapfold ~ncores:n f (L l) op b concat
```

computes List.fold_right op (List.map f l) b by forking *n* processes on a multicore machine. You need to provide the extra *concat* operator to combine the partial results of the fold computed on each core.

Le module `Graphics` et le calcul parallèle

- Attention, on ne peut pas utiliser les primitives du module *Graphics* dans des processus créés par un parmapfold.
- Il faut faire un calcul pur dans les processus, et on fait du graphisme seulement au moment de la collection.
- On se sert du fait que parmapfold fait une séparation entre l'argument *op*, qui est calculé par chacun des processus, et l'argument *concat* qui est calculé par le collecteur.
- voir `graphicspar.ml`

Limites du Speedup

- Il est important de se rappeler toujours de la loi de Amdhal (1967), qui dit que si notre programme a une partie du code parallélisable correspondante a une fraction p du temps d'exécution séquentiel, alors le meilleur speedup qu'on peut atteindre en utilisant n processeurs ne peut dépasser

$$S_{max} = \frac{1}{(1-p)+\frac{p}{n}}$$

- C'est une limitation forte : si 10% du temps de calcul est séquentiel, alors le speedup de dépassera jamais 10, peu importe le nombre de processeurs utilisés

Pour en savoir plus

Richard S. Bird.
An introduction to the theory of lists.
Logic of Programming and Calculi of Discrete Design, pages 3–42. Springer-Verlag, 1987.
Also http://www.cs.ox.ac.uk/files/3378/PRG56.pdf

Murray Cole.
Parallel programming with list homomorphisms.
Parallel Processing Letters, 5 :191–203, 1995.

M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti.
Parallel functional programming with skeletons : the OCamlP3l experiment. The ML Workshop, 1998.

Marco Danelutto and Roberto Di Cosmo.
A "minimal disruption" skeleton experiment : Seamless map & reduce embedding in OCaml.
Procedia CS, 9 :1837–1846, 2012

Manipuler (réécrire) des séquences de combinateurs

Revenons à l'équation III.4 de Backus :

$$List.map(f \circ g) = (List.map\ f) \circ (List.map\ g)$$

Il y a plusieurs façons de l'utiliser :

1. *manuel* : on peut écrire une première version du code et la *réécrire* en appliquant l'équation
2. *automatique* : on peut instrumenter le compilateur pour qu'il reconnaisse certaines équations et les applique dans certaines conditions (cette approche est utilisée dans Haskell)
3. *par programme* : on peut écrire des combinateurs qui construisent une structure de donnée intermédiaire sur laquelle une fonction de optimisation ou réécriture est appelée
   C'est cette dernière approche que nous allons explorer dans la suite.

Des combinateurs aux structures de données

- Grâce aux GADT, il est possible de décrire précisément une structure de données qui peut représenter une composition de combinateurs.
  Nous explorons cela sur l'exemple des listes, avec map, fold_left et fold_right
- En nous inspirant du code de compute vu dans le cours sur le GADT, il est facile d'écrire une fonction qui exécute les combinateurs correspondant à ces constructeurs.

Exemples (comb2.ml)

```
type (_,_) lcomb =
| Map :
  ('a -> 'b) -> ('a list,'b list) lcomb
| Fold_left :
  ('a -> 'b -> 'a) * 'a -> ('b list,'a) lcomb
| Fold_right :
  ('a -> 'b -> 'b) * 'b -> ('a list,'b) lcomb
| Comb :
  ('a,'b) lcomb * ('b,'c) lcomb -> ('a,'c) lcomb
;;
```


Exemples (comb3.ml)

```
 let rec exec : type a b. a -> (a,b) lcomb -> b =
fun l ->
 function
  | Map f -> List.map f l (* 'a = 'c list ; 'b = 'd list
  | Fold_left (f,e) -> List.fold_left f e l
  | Fold_right (f,e) -> List.fold_right f l e
  | Comb (c1,c2) -> let l' = exec l c1 in exec l' c2
;;
```


Des combinateurs aux structures de données

- Pour commodité, on introduit un opérateur *infixe* |> de combinaison sur ces constructeurs, et un opérateur @ d'évaluation sur une liste.
- Ainsi, on peut écrire nos transformations de listes en utilisant ces constructeurs comme si on écrivait une suite de combinateurs de listes.
- Avec cette étape supplémentaire, nous gagnons la possibilité de transformer les combinateurs à notre guise.

Exemples (comb4.ml)

```
let ( |> ) = fun c c' -> Comb (c,c');;
let ( @ ) c l = exec l c;;

let myop = Map (fun x -> x+1) |> Map (fun x -> x*2);;
myop @ [1;2;3];;

let rec optimize : type a b. (a,b) lcomb -> (a,b) lcomb =
function
| Comb (c1,c2) ->
   (match optimize c1, optimize c2 with
    | Map f, Map g -> Map (fun x -> g (f x))
    | c,c' -> Comb (c,c'))
| x -> x;;
(optimize myop) @ [1;2;3];;
```

Changer la sémantique des combinateurs

- Grâce au fait que nous avons construit une structure de données, il devient possible de *changer* leur *sémantique*. Par exemple *dessinons* la suite de transformations.
- Et maintenant nous pouvons redéfinir l'opération d'application d'une suite de transformations, pour qu'elle affiche le schéma graphique de la suite :

Le Code complet I

```
(*  Author(s):   Roberto Di Cosmo     *)

(** The GADT defining List combinators *)

type ('a,'b) lcomb =
| Map : ('a -> 'b) -> ('a list,'b list) lcomb
| Fold_left : ('a -> 'b -> 'a) * 'a -> ('b list,'a) lcomb
| Fold_right : ('a -> 'b -> 'b) * 'b -> ('a list,'b) lcomb
| Comb : ('a,'b) lcomb * ('b,'c) lcomb -> ('a,'c) lcomb

(* Executing combinators *)

let rec exec : type a b. a -> (a,b) lcomb -> b =
fun l ->
 function
 | Map f -> List.map f l (* here 'a = 'c list and 'b = 'd list*)
 | Fold_left (f,e) -> List.fold_left f e l
```


Le Code complet II

```
  | Fold_right (f,e) -> List.fold_right f l e
  | Comb (c1,c2) -> let l' = exec l c1 in exec l' c2

(* Example of an optimization function *)

let rec optimize : type a b. (a,b) lcomb -> (a,b) lcomb =
function
| Comb (c1,c2) ->
   (match optimize c1, optimize c2 with
    | Map f, Map g -> Map (fun x -> g (f x))
    | c,c' -> Comb (c,c'))
| x -> x

(* Drawing combinator sequences *)

open Graphics;;
(* #require "graphics";; *)
```


Le Code complet III

```
let rec draw_step name x0 y0 x1 y1 =
  let xc = (x1-x0)/2 and yc = (y1-y0)/2 in
  let r  = truncate((float (min xc yc))*.0.80) in
  let dr = truncate((float r) *. 0.20) in
  let lname = (fst (text_size name)) in
    draw_circle (xc+x0) (y0+yc) r ;
    moveto x0 (y0+(y1-y0)/2); lineto (x0+dr) (y0+(y1-y0)/2);
    moveto (x1-dr) (y0+(y1-y0)/2); lineto (x1)  (y0+(y1-y0)/2);
    moveto (xc+x0-(lname/2)) (y0+yc); draw_string name
and draw_comb
  : type a b c.(a,b) lcomb * (b,c) lcomb -> int -> int -> int -> i
  = fun (c1,c2) x0 y0 x1 y1 ->
  let dx = (x1-x0)/2 in
  draw_expr c1 x0 y0 (x0+dx) y1;
  draw_expr c2 (x0+dx) y0 (x0+2*dx) y1;
and draw_expr
  : type a b.(a,b) lcomb -> int -> int -> int -> int -> unit
  = fun e x0 y0 x1 y1 ->
```

Le Code complet IV

```
    match e with
      Map s          -> (draw_step "map" x0 y0 x1 y1)
    | Fold_left (_,_)  -> (draw_step "fold_left" x0 y0 x1 y1)
    | Fold_right (_,_) -> (draw_step "fold_right" x0 y0 x1 y1)
    | Comb (c1,c2) -> (draw_comb (c1,c2) x0 y0 x1 y1)

(* The interface of a list combinator implementation *)

module type COMB =
sig
  type ('a,'b) t
  val map : ('a -> 'b) -> ('a list, 'b list) t
  val fold_left : ('a -> 'b -> 'a) -> 'a -> ('b list, 'a) t
  val fold_right : ('a -> 'b -> 'b) -> 'b -> ('a list, 'b) t
  val ( |> ) : ('a, 'b) t -> ('b, 'c) t -> ('a, 'c) t
  val optimize : ('a, 'b) t -> ('a, 'b) t
  val ( @ ) : ('a, 'b) t -> 'a -> 'b
end
```


Le Code complet V

```
(** List combinator implementation using the List from OCaml *)

module CombPlain : COMB =
struct
 type ('a,'b) t = 'a -> 'b

 let map f l = List.map f l
 let fold_left f e l = List.fold_left f e l
 let fold_right f e l = List.fold_right f l e

 let ( |> ) = fun f g l -> g (f l)

 let rec optimize c = c

 let ( @ ) c l = c l
end;;
```


Le Code complet VI

```
(** List combinator implementation using exec *)

module CombExecInternal =
struct
 (* definition des combinateurs avec @ = exec *)
 type ('a,'b) t = ('a,'b) lcomb
 let map f = Map f
 let fold_left f e = Fold_left (f,e)
 let fold_right f e = Fold_right (f,e)

 let ( |> ) = fun c c' -> Comb (c,c')

 let optimize = optimize

 let ( @ ) c l = exec l c
end;;

(** List combinator implementation using draw *)
```


Le Code complet VII

```
module CombDraw : COMB =
struct
 include CombExecInternal
 let ( @ ) c l =
    open_graph "␣1024x800";
    draw_expr c 0 0 (size_x()) (size_y());
    print_string "Enter␣a␣newline␣to␣stop␣...\n";
    let _ = read_line() in
    close_graph(); exec l c
end;;

(* restrict CombExec after defining CombDraw *)
(* to share the abtract type definition *)

module CombExec = (CombExecInternal : COMB);;

(* parse the command line options *)
```

Le Code complet VIII

```
let mode=ref (`Exec : [< `Exec | `Seq | `Gra ]) ;;
let set m = mode := m;;

Arg.parse [
  ("-draw", Arg.Unit (fun () -> set `Gra), "Draw␣combinator␣sequences."
  ("-plain", Arg.Unit (fun () -> set `Seq), "Use␣directly␣the␣list␣com
 ] (fun _ -> ()) "Test␣combinators:␣try␣me␣with␣and␣without␣-draw";;

(* choose the right implementation, using first class modules *)

module Comb =
  (val
      (match !mode with
        `Gra -> (module CombDraw : COMB)
      | `Exec -> (module CombExec : COMB)
      | `Seq -> (module CombPlain : COMB)
      ) : COMB);;
```


Le Code complet IX

```
(* and use it in the rest of the program *)

open Comb;;

let myop = map (fun x -> x+1) |> map (fun x -> x*2);;
let pr_list l = List.iter (fun n -> Printf.printf "%d␣" n) l; Prin

pr_list (myop @ [1;2;3]);;

pr_list((optimize myop) @ [1;2;3]);;
```


Bilan des Combinateurs

On a manipulé des combinateurs sous différentes formes, et on a bien vérifié que :

- on a besoin de la récursion pour les définir, *pas pour les utiliser*
- ils permettent d'écrire des programmes sans points, par *simple composition*
- ils capturent un schéma général réutilisable
- ils permettent de définir des transformations génériques intéressantes
- ils permettent d'écrire du code dont la sémantique peut changer sans changer le code

Les DSL

- Une famille cohérente de combinateurs peut définir un *Domain Specific Language*, ou DSL.
- Il s'agit le plus souvent d'un langage très spécialisé pour un domaine applicatif spécifique, et qui n'est pas Turing-complet : cela permet de prouver des propriétés et effectuer des transformations qui seraient sinon difficiles à prouver correctes.

Combinators Everywhere

- listes...
- parallélisme...
- contrats financiers...
- workflow/business logic...
- requêtes sur les données...
- sondages en ligne...

Programmation Fonctionnelle Avancée 7 : Combinators
└Combinateurs et Optimisation

Pour en savoir plus

Simon L. Peyton Jones.
Composing contracts : An adventure in financial engineering.
In FME, page 435, 2001.

Christian Stefansen.
Smawl : A small workflow language based on CCS.
In CAiSE Short Paper Proceedings, 2005.