

Programmer avec des combinateurs

- ▶ Définitions de base : combinateurs, DSL, programmation sans points
- ▶ Redécouverte des combinateurs sur les listes
- ▶ Combinateurs pour le calcul parallèle

Quelques exemples de Backus : combinateurs

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

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

Cette définition ne fait que *combiner* des opérations plus élémentaires (qu'on appelle des *combinateurs*, ici *List.fold*, *List.map2*, *o* et *Transpose*).

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*.

N.B. : la valeur restriction en OCaml ne permet pas d'utiliser ce style en toute généralité ; pourquoi ?

1978 : Turing Award pour John Backus, créateur de Fortran

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

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.

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.

Revisitons les opérations sur les listes...

On utilise souvent les fonctions `List.map` et `List.fold_left`.

```
let rec map f = function
  [] -> []
  | a::l -> let r = f a in r :: map f l;;
|| val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

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

Ce sont des *combinateurs* pour les opérations sur les listes !

List homomorphism Lemma (Bird, 1986)

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

$$h[] = e$$
$$h(l_1 @ l_2) = (h l_1) \oplus (h l_2)$$

Exemples :

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

Theorem (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

$$h = (\text{reduce } f) \circ (\text{map } g)$$

N.B. : quand \oplus est associatif, et dispose d'un élément neutre, `fold_left` et `fold_right` coïncident et on les appelle *reduce*.

Des combinateurs pour les listes

Préons le temps de réfléchir à leur signification :

- ▶ on a besoin de la recursion 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

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 lequel il est possible de définir un grand nombre d'opérations sur les listes.

Par exemple, `concat = fold_left append []`.

Voyons voir ce qu'on peut faire avec juste *map* et *fold_left* !

Appication majeure : exécution parallèle

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

Quelques notions de base :

Definition (Speedup)

Le *speedup* d'un algorithme parallèle

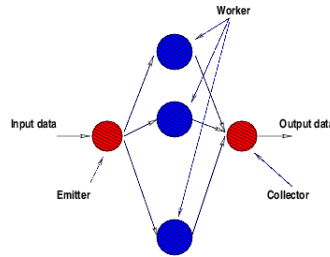
$$S_n = \frac{T_1}{T_n}$$

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 : une minute de réflexion

Le temps de calcul parallèle est de toute façon *au minimum* de l'ordre de $\log n$.

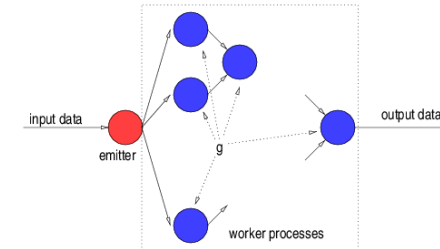
Donc, il convient d'envoyer aux feuilles de l'arbre déjà des segments de liste de taille $\log n$: le calcul sera fait plus vite en séquentiel (il n'y a pas d'overhead de communication).

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 $S_{n/\log n} \approx \frac{n}{\log n}$.

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.



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 .

On regarde 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 :

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

Difficulté

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 est entièrement dans l_1 ou dans l_2 , et pas s'il est au cheval des deux listes.

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 .

mis : le $mis (l_1 @ l_2)$ est égal à $(mis l_1) \uparrow (ts l_1 + mis l_2)$

mcs : le $mcs (l_1 @ l_2)$ est égal à $(mcs l_2) \uparrow (mcs l_1 + ts l_2)$

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 *extraît* mss seulement à la fin.

Calculer le maximum segment sum

Si on découpe $[2; -4; 2; -1; 6; -3]$ en $[2; -4; 2]$ et $[-1; 6; -3]$ on a $mss [2; -4; 2] = 4$ et $mss [-1; 6; -3] = 6$, mais $4 \uparrow 6 < 7$.

Pour couvrir le cas du mss à cheval des deux, on doit aussi prendre en compte une *maximum concluding sum* de l_1 et un *maximum initial sum* de l_2 .

Dans notre exemple :

$$mcs[2; -4; \underline{2}] = 2$$

$$mis[\underline{-1}; 6; -3] = 5$$

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

Maximum segment sum dans un homomorphisme

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

$$(mss, mis, mcs, ts) \oplus (mss', mis', mcs', ts') = \\ (mss \uparrow mss' \uparrow (mcs + mis'), mis \uparrow (ts + mis'), \\ mcs' \uparrow (mcs + ts'), ts + ts')$$

Et l'opération pour la phase *map*, qui calcule ces quadruplets 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?
- ▶ qui est l'élément neutre de \oplus ?

Une fois cela vérifié, on peut définir alors

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

Maximum segment sum en OCaml II

```
mss [2; -4; 2; -1; 6; -3];;  
|| - : int = 7
```

Maximum segment sum en OCaml I

Voici le code qu'on peut écrire en OCaml :

```
let op =  
  fun (mss, mis, mcs, ts) -> fun (mss', mis', mcs', ts') ->  
    max mss (max mss' (mcs+mis')),  
    max mis (ts+mis'),  
    max mcs' (mcs+ts'),  
    ts+ts';;  
|| val op :  
|| int * int * int * int -> int * int * int * int -> int * int * int *  
|| t =  
|| <fun>  
  
let f x = (max 0 x, max 0 x, max 0 x, x);;  
|| val f : int -> int * int * int * int = <fun>  
  
let mss l =  
  let (v, _, _, _) = List.fold_left op (0,0,0,0) (List.map f l)  
  in v;;  
|| val mss : int list -> int = <fun>
```

Et en parallèle?

On peut utiliser <http://functory.lri.fr>, et écrire :

```
open Functory.Cores  
let () = Functory.Control.set_debug true  
let () = set_number_of_cores 4  
(* sequential *)  
let mss_seq l =  
  let (v, _, _, _) = List.fold_left op (0,0,0,0) (List.map f l)  
  in v;;  
(* parallel *)  
let mss_par l =  
  let (v, _, _, _) = map_local_fold ~f:f ~fold:op (0,0,0,0) l  
  in v;;  
Printf.printf "Mssseq=%d\n%!" (mss_seq [2; -4; 2; -1; 6; -3]);;
```

Comparez avec le laborieux approche en utilisant Hadoop, par exemple ici : <http://pages.cs.brandeis.edu/~cs147a/lab/hadoop-example/>

Attention la phase *fold* n'est pas parallélisée dans Functory. Si besoin, utiliser Parmap ou Netcore, ou remettez à jour OCamlP3I :-)

No free lunch





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

Stratégies pour combiner et optimiser des combinateurs

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 OCamlP3I 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

Révenons 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 :

- manuel** : on peut écrire une première version du code et la **réécrire** en appliquant l'équation
- automatique** : on peut instrumenter le compilateur pour qu'il reconnaisse certaines équations et les applique dans certaines conditions (cet approche est utilisée dans Haskell)
- 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`

```
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;;
```

Des combinateurs aux structures de données

Pour commodité, on introduit un opérateur *infixe* `|>` de combinaison sur ces constructeurs en un opérateur `@` d'évaluation sur une liste :

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

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

Ainsi, on peut écrire nos transformations de listes en utilisant ces constructeurs comme si on écrivait une suite de combinateurs de listes.

```
let myop = Map (fun x -> x+1) |> Map (fun x -> x*2);;  
|| val myop : (int list, int list) lcomb = Comb (Map <fun>, Ma
```

et les appliquer facilement :

```
myop @ [1;2;3];;  
|| - : int list = [4; 6; 8]
```

Des combinateurs aux structures de données

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.

```
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 = 'a *)  
  | 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  
  ;;  
|| val exec : 'a -> ('a, 'b) lcomb -> 'b = <fun>
```

Transformation des (représentations de) combinateurs

Avec cette étape supplémentaire, nous gagnons la possibilité de transformer les combinateurs à notre guise :

```
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;;  
|| val optimize : ('a, 'b) lcomb -> ('a, 'b) lcomb = <fun>  
  
(optimize myop) @ [1;2;3];;  
|| - : int list = [4; 6; 8]
```

Changer la sémantique des combinateurs I

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.

Attention : si vous travaillez dans l'interpréteur, il faut charger la librairie graphique avec la directive :

```
#load "graphics.cma";;
open Graphics;;

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+xc) (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
```

Changer la sémantique des combinateurs III

```
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;;
|| val ( @ ) : ('a, 'b) lcomb -> 'a -> 'b = <fun>
```

Essayez par exemple :

```
myop @ [1;2;3;4];;
(optimize myop) @ [1;2;3;4];;
```

Changer la sémantique des combinateurs II

```
and draw_comb
: type a b c.(a,b) lcomb * (b,c) lcomb -> int -> int -> int -> int ->
= 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 ->
  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)
;;
```

Et maintenant nous pouvons redéfinir l'opération d'application d'une suite de transformations, pour qu'elle affiche le schéma graphique de la suite :

Une seule interface pour quatre implémentations I

On peut donner la même interface à toutes les implémentations que nous vons vu jusqu'ici :

```
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;;
```

On encapsule dans des modules ces différentes implémentations, et on restreigne leur interface pour s'assurer qu'on ne puisse les utiliser qu'avec la même interface COMB.

Une seule interface pour quatre implémentations II

```

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;;

```

Une seule interface pour quatre implémentations IV

```

module CombDraw : COMB =
struct
  (* definition des combinateurs avec @ = draw *)
  include CombExecInternal

  let ( @ ) c l =
    open_graph "└1024x800";
    draw_expr c 0 0 (size_x()) (size_y());
    print_string "Enter└└newline└to└stop└...\n";
    let _ = read_line() in
    close_graph(); exec l c
end;;

```

Une seule interface pour quatre implémentations III

```

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;;

```

Ajout d'une implémentation parallèle

Nous pouvons ajouter une implémentation pour l'exécution parallèle de nos combinateurs en utilisant la librairie Functory vue plus haut. Pour cela, il faut avoir installé cette librairie.

Avec `opam`, il suffit d'utiliser la commande

```
opam install functory
```

pour que la librairie soit compilée depuis les sources et installée.

Ajout d'une implémentation parallèle

Dans le toplevel, on peut utiliser `topfind` pour charger la librairie.

```
#use "topfind";;
|| - : unit = ()
|| Findlib has been successfully loaded. Additional directives:
|| #require "package";;      to load a package
|| #list;;                  to list the available packages
|| #camlp4o;;               to load camlp4 (standard syntax)
|| #camlp4r;;               to load camlp4 (revised syntax)
|| #predicates "p,q,...";;  to set these predicates
|| Topfind.reset();;        to force that packages will be reloaded
|| #thread;;                to enable threads
||
|| - : unit = ()

#require "functory";;
|| /home/dicosmo/.opam/4.00.1/lib/ocaml/unix.cma: loaded
|| /home/dicosmo/.opam/4.00.1/lib/functory: added to search path
|| /home/dicosmo/.opam/4.00.1/lib/functory/functory.cma: loaded
```

Exporter seulement les modules avec interface COMB

On a du garder ouverte l'interface de `CombExecInternal` pour pouvoir l'inclure dans les modules `CombDraw` et `CombPar` (pourquoi?), mais nous allons exporter seulement une version restreinte.

```
module CombExec = (CombExecInternal : COMB);;
|| module CombExec : COMB
```

Tout le code vu jusqu'ici doit être mis dans un module qui rend visible seulement le **module type** `COMB` et les modules `CombPlain`, `CombExec`, `CombDraw` et `CombPar`. Cela force les utilisateurs de ce modules à utiliser exclusivement les combinateurs `map`, `fold_left`, `fold_right`, `comb` exportés par la librairie.

Ajout d'une implémentation parallèle

```
module CombPar : COMB =
struct
  include CombExecInternal
  let rec exec : type a b. a -> (a,b) lcomb -> b =
  fun l ->
  function
  | Map f -> Functory.Cores.map ~f l
  | Fold_left (f,e) -> Functory.Cores.map_local_fold
    ~f:(fun x -> x) ~fold:f e l
  | Fold_right (f,e) -> Functory.Cores.map_local_fold
    ~f:(fun x -> x) ~fold:(fun x y -> f y x) e l
  | Comb (Map f, Fold_left(g,e)) ->
    Functory.Cores.map_local_fold ~f:f ~fold:g e l
  | Comb (Map f, Fold_right(g,e)) ->
    Functory.Cores.map_local_fold ~f ~fold:(fun x y -> g y x) e l
  | Comb (c1,c2) -> let l' = exec l c1 in exec l' c2

  let ( @ ) c l = exec l c
end;;
|| module CombPar : COMB
```

Le tout dans un seul exécutable I

Avec les modules de première classe, on peut construire *un seul* exécutable qui permet de choisir la sémantique à l'exécution, en gardant un code utilisateur absolument propre. Voici un exemple minimaliste : on commence par parser des options sur la ligne de commande :

```
let mode=ref ('Exec : [< 'Exec | 'Seq | 'Gra | 'Par of int]) ;;

let set m = mode := m;;
```

```
Arg.parse [
  ("draw", Arg.Unit (fun () -> set 'Gra), "Draw combinator sequences.");
  ("plain", Arg.Unit (fun () -> set 'Seq), "Use the List module.");
  ("par", Arg.Int (fun n -> set ('Par n);
    Functory.Control.set_debug true;
    Functory.Cores.set_number_of_cores n),
    "Use Functory to execute on n cores.");
] (fun _ -> ()) "Test combinators choosing their semantics at runtime"
```

Le tout dans un seul exécutable II

Ensuite, on crée un module Comb qui contient l'un d'entre CombPlain, CombExec, CombDraw et CombPar en fonction des paramètres passés sur la ligne de commande.

```

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

et on l'ouvre : le reste du programme utilisera les combinateurs définis dans ce module.

```
open Comb;;
```

Leçons des Combinateurs

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

- ▶ on a besoin de la recursion pour les définir, *pas pour les utiliser*
- ▶ ils permettent d'écrire des programmes sans points, par *simple composition*
- ▶ ils capturent un schema 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

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 difficile à prouver correctes.

Le tout dans un seul exécutable III

```

let myop = map (fun x -> x+1) |> map (fun x -> x*2);;
|| val myop : (int list, int list) Comb.t = <abstr>

let pr_list l = List.iter (fun n -> Printf.printf "%d_" n) l;
                          Printf.printf "\\n%!";;
|| val pr_list : int list -> unit = <fun>

pr_list (myop @ [1;2;3]);;
|| 4 6 8
|| - : unit = ()

pr_list((optimize myop) @ [1;2;3]);;
|| 4 6 8
|| - : unit = ()
    
```

Pour compiler et tester ce code vous même, il faudra utiliser une commande comme celle ci

```
ocamlfind ocamlpt -linkpkg -package functor,graphics \
-o comb.native comb.ml
```

Combinators Everywhere

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

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.