3.11. Stratégies d'évaluation

3.11.1. Principes de base

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
(Fonction intermédiaire.)
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
(Fonction intermédiaire.)
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
(Point d'entrée de l'exécution.)
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la *stratégie d'évaluation* du langage, c'est-à-dire, de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)` :

1. si l'évaluation de cet appel à `g` a pour prérequis de connaître les valeurs de ses arguments, alors `f` est appliquée à `-1`, ce qui provoque une **non-terminaison** ;
2. sinon, l'expression `f(-1)` n'est pas évaluée car le second argument, `0`, de l'appel à `g` est pair. L'exécution se **termine** dans ce cas.

3.11.2. Appel par valeur

La stratégie d'évaluation en *appel par valeur* consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

Exemple

Si l'on a une fonction

```
let f x y z =
    x + z
```

l'expression

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)
```

s'évalue au moyen des étapes suivantes :

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)  →  f 1 (f 2 1 4) 12  →  f 1 6 12  →  13
```

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent) ;
2. cette stratégie peut influencer négativement la **terminaison** de certains programmes (voir l'exemple introductif).

Beaucoup de langages utilisent cette stratégie, dont l'OCaml.

3.11.3. Appel par nom

La stratégie d'évaluation en *appel par nom* consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

Exemple

Si l'on a une fonction

```
let f x y z =
    x + z
```

l'expression

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)
```

s'évalue au moyen des étapes suivantes :

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)  →  (1 * 1) + (3 * 4)  →  13
```

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

Exemple

En effet, si l'on a une fonction

```
let f x y =
    x * x + y
```

l'expression

```
f (4 * 3) (2 * 1)
```

s'évalue en appel par nom au moyen des étapes suivantes :

`f (4 * 3) (2 * 1)` → `(4 * 3) * (4 * 3) + (2 * 1)` → `146`

L'argument `4 * 3` est **évalué** ainsi **deux fois** (au lieu d'une seule que ferait un appel par valeur).

3.11.4. Appel par nécessité

La stratégie en *appel par nécessité* est une **version mémoïsée de l'appel par nom**.

Une fonction est *mémoïsée* si, à chaque premier appel pour un jeu d'arguments donnés, la **valeur qu'elle renvoie** est **enregistrée** dans une table associant les arguments au résultat. Ainsi, tout second appel à la fonction avec le même jeu d'arguments ne provoque pas de réévaluation.

Lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, chaque `ai` n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

De plus, l'évaluation d'un `ai`, si elle a lieu, est enregistrée. Ainsi, toute occurrence d'un paramètre de `f` correspondant à un `ai` ne redemande pas d'être réévaluée.

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le **principe de transparence référentielle** est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Rappel : le principe de transparence référentielle est respecté si dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat construit par l'exécution du programme.

Le langage HASKELL utilise cette stratégie.

3.12. Non mutabilité

3.12.1. Principes généraux

Principalement en programmation fonctionnelle (mais pas uniquement), nous manipulons des données *non mutables* : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné une entité `x`, pour obtenir une entité `x'` calculée à partir de `x`, il faut **reconstruire** `x'`.

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée ;
2. démonstration de correction de programmes facilitée ;
3. gain de place mémoire.

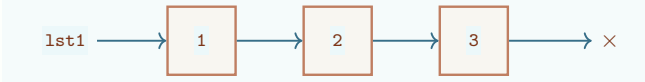
Ces trois avantages s'appuient sur le fait que plusieurs grosses données peuvent **partager** des sous-données en commun, **sans aucune interférence**.

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par la **construction d'une liste résultat**.

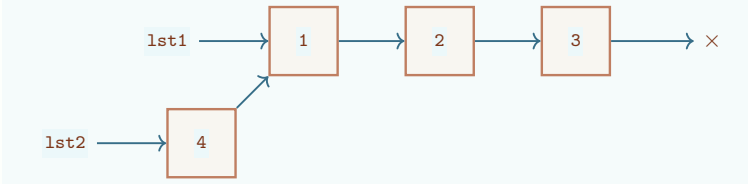
Considérons les phrases

```
# let lst1 = [1; 2; 3];;
# let lst2 = 4 :: lst1;;
```

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé qu'une seule cellule et **partage** les trois précédentes :



Considérons la fonction

```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2)
```

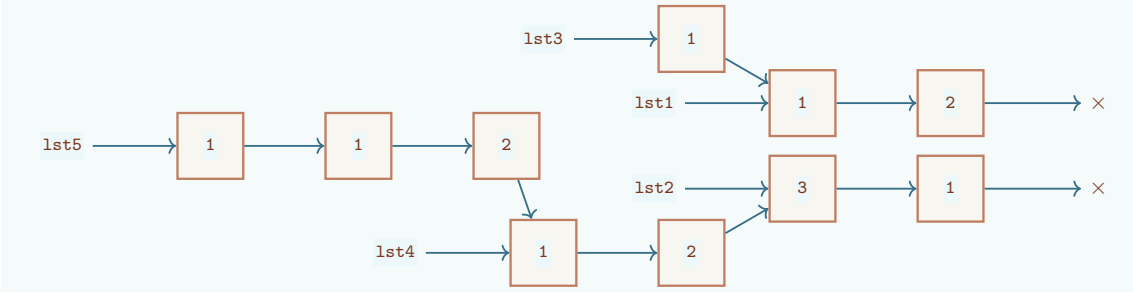
Elle permet de concaténer deux listes (il s'agit d'une implantation de la fonction `List.append`).

Considérons les phrases

```
# let lst1 = [1; 2];;
# let lst2 = [3; 1];;
```

```
# let lst3 = 1 :: lst1;;
# let lst4 = concatener lst1 lst2;;
# let lst5 = concatener lst3 lst4;;
```

Leur exécution produit en mémoire la configuration de partage suivante :



3.12.2. Implantation des files

Nous souhaitons implanter les *files* (piles First In, First Out) dont les éléments sont d'un type quelconque.

On doit pour cela

1. définir un type à un paramètre `file` pour **représenter les files polymorphes** ;
2. définir une constante `vide : 'a file` égale à la **file vide** ;
3. définir une fonction `ajouter : 'a -> 'a file -> 'a file` qui renvoie une nouvelle file prenant en compte de l'**ajout d'un élément** ;
4. définir une fonction `ancien : 'a file -> 'a option` qui renvoie une valeur optionnelle sur **le plus ancien élément** de la file ;
5. définir une fonction `supprimer : 'a file -> 'a file` qui renvoie la file obtenue en **supprimant son plus ancien élément**.

La fonction `ancien` appliquée à la file vide renvoie `None`.

La fonction `supprimer` appliquée à la file vide renvoie la file vide.

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list

let vide = []

let ajouter x f =
    x :: f
```

```
let rec ancien f =
    match f with
        |[] -> None
        |[e] -> Some e
        |_ :: reste -> ancien reste
```

```
let rec supprimer f =
    match f with
        |[] |[_] -> []
        |e :: reste -> e :: supprimer reste
```

En notant par n le nombre d'éléments de la file, nous obtenons les complexités

Fonction	Complexité en temps
`ajouter`	$\Theta(1)$
`ancien`	$\Theta(n)$
`supprimer`	$\Theta(n)$

Une variante possible consiste à ranger les éléments du plus ancien au plus récent. Les complexités en temps obtenues sont complémentaires à celles présentées ici.

Il existe une implantation qui respecte aussi **le principe de non-mutabilité** mais qui est beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes in et out.

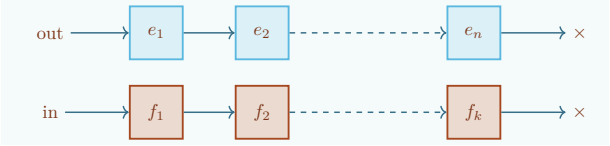
- Les éléments **prêts à sortir** se situent dans out. Ils sont rangés **du plus ancien au plus récent**.
- Les éléments **ajoutés** sont « mis en mémoire tampon » dans in. Ils sont rangés **du plus récent au plus ancien**.

Les opérations sur cette structure de données se décrivent ainsi :

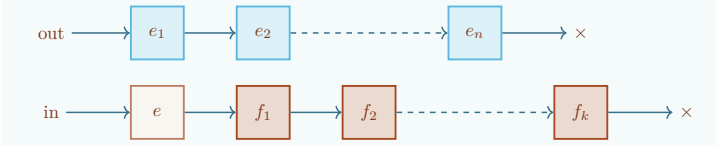
- l'ajout d'un élément e à la file consiste à positionner e en tête de in ;
- la suppression/renvoi du plus ancien élément consiste à supprimer/renvoyer la tête de out si elle est non vide.

  Si out est vide, on remplace out par le **miroir** de in, on vide in et on supprime/renvoie la tête de out.

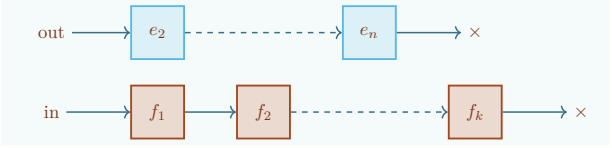
Voici une file dans une situation générique :



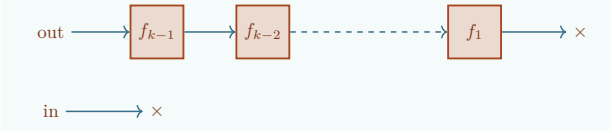
Après l'**ajout** d'un élément e depuis la situation générique, elle devient



Après une **suppression** depuis la situation générique, elle devient



et renvoie e_1 si out n'est pas vide,



et renvoie f_k si out est vide.

Exemple

Opération	in	out	Éléments (plus ancien en 1^{er})
`vide`	`[]`	`[]`	ϵ
`ajout(1)`	`[1]`	`[]`	1
`ajout(1)`	`[1; 1]`	`[]`	$1, 1$
`ajout(2)`	`[2; 1; 1]`	`[]`	$1, 1, 2$
`ajout(3)`	`[3; 2; 1; 1]`	`[]`	$1, 1, 2, 3$
`supprimer()`	`[]`	`[1; 2; 3]`	$1, 2, 3$
`ajout(4)`	`[4]`	`[1; 2; 3]`	$1, 2, 3, 4$
`ajout(5)`	`[5; 4]`	`[1; 2; 3]`	$1, 2, 3, 4, 5$
`supprimer()`	`[5; 4]`	`[2; 3]`	$2, 3, 4, 5$
`supprimer()`	`[5; 4]`	`[3]`	$3, 4, 5$
`ajouter(6)`	`[6; 5; 4]`	`[3]`	$3, 4, 5, 6$
`supprimer()`	`[6; 5; 4]`	`[]`	$4, 5, 6$
`supprimer()`	`[]`	`[5; 6]`	$5, 6$

```
type 'a file =
    {entree : 'a list ; sortie : 'a list}
```

```
let vide =
    {entree = [] ; sortie = []}
```

```
let ajouter x f =
    {f with entree = x :: f.entree}
```

```
let ancien f =
    match f.sortie with
        |[] -> begin
            match List.rev f.entree with
                |[] -> None
                |e :: _ -> Some e
        end
        |e :: _ -> Some e
```

```
let supprimer f =
    match f.sortie with
        |[] -> begin
            match List.rev f.entree with
                |[] -> f
                |_ :: reste -> {entree = [] ; sortie = reste}
        end
        |_ :: reste -> {f with sortie = reste}
```

Cette implantation est plus efficace que la précédente.

Elle est même strictement plus efficace : les trois opérations ont une complexité en temps en $\Theta(1)$ sauf lorsque out est vide, ce qui demande pour `ancien` et `supprimer` une mise à jour en $\Theta(n)$.

Les fonctions précédentes de manipulation de files s'utilisent aisément en pratique à l'aide de

- l'opérateur d'application d'inversée ;
- la fonction `Option.map : ('a -> 'b) -> 'a option -> 'b option` ;
- la fonction `Option.bind : 'a option -> ('a -> 'b option) -> 'b option`.

Exemple

```
# let f = vide |> ajouter 1 |> ajouter 1 |> ajouter 2 |> ajouter 3;;
val f : int file = {entree = [3; 2; 1; 1]; sortie = []}

# let f = f |> supprimer;;
val f : int file = {entree = []; sortie = [1; 2; 3]}

# let f = f |> ajouter 4 |> ajouter 5;;
val f : int file = {entree = [5; 4]; sortie = [1; 2; 3]}

# f |> ancien |> Option.map (fun x -> Printf.printf "Valeur : %d\n" x);;
Valeur : 1
- : unit option = Some ()
```

3.13. Preuves

3.13.1. Validité d'un programme

Un **programme** p est *valide* si ce qu'il calcule est conforme à ce qui est attendu.

Ce qui est attendu est décrit par une *spécification*.

Exemples

Voici quelques spécifications de programmes :

- le programme p_1 affiche tous les diviseurs de 60 ;
- le programme p_2, sur l'argument entier n, affiche « oui » si n est un nombre premier et « non » sinon.

Cette notion s'étend aux **fonctions** : une fonction f est *valide* si elle est conforme à une *spécification*. Elle décrit ce que f renvoie lorsque appliquée à des arguments a_1, ..., a_n.

Exemples

Voici quelques spécifications de fonctions :

- la fonction f_1 renvoie la somme des éléments de la liste d'entiers sur laquelle elle est appliquée ;
- la fonction f_2 teste si l'arbre binaire sur lequel elle est appliqué est équilibré.

Un *test* (*unitaire*) d'une fonction f est un couple (e, r) où e est une expression dans laquelle f intervient, pour laquelle le résultat r conforme à la spécification de f est connu.

Il consiste à comparer e avec la valeur attendue r. Ce test est *positif* lorsque la valeur de e est bien r et est *négatif sinon*.

Exemple

Soit la fonction

```
let est_premier n =
    List.init (n - 2) ((+) 2)
    |> List.exists (fun k -> n mod k = 0)
    |> not
```

spécifiée par « `est_premier` renvoie `true` ssi elle est appliquée à un entier $n \geqslant 2$ premier ».

Un test de `est_premier` est composé de l'expression

```
List.init 20 ((+) 2) |> List.filter est_premier
```

et de la valeur attendue

```
[2; 3; 5; 7; 11; 13; 17; 19]
```

Ce test est positif.

Dans le cas où f est une fonction de **domaine fini**, un ensemble de tests parcourant toutes les entrées possibles peut montrer que la fonction est valide. Cet ensemble de tests est alors qualifié de *couvrant*.

Exemple

Soient le type et la fonction

```
type joueurs = Blancs | Noirs

let adversaire j =
    match j with
        |Blancs -> Noirs
        |Noirs -> Blancs
```

spécifiée par « `adversaire` renvoie l'adversaire du joueur sur lequel elle est appliquée ».

Son domaine est fini. Les deux tests

`(adversaire Blancs, Noirs)` et `(adversaire Noirs, Blancs)`

forment un ensemble de tests couvrant.

Dans le cas contraire, un ensemble de tests **ne démontre pas la validité** d'une fonction. Il peut seulement **démontrer son invalidité**.

Malgré cette limitation pour les fonctions à domaines infinis (qui sont les plus courantes), les tests sont utiles pour

- détecter des **erreurs** lors de la **phase initiale** de programmation ;
- détecter des **erreurs émergentes** lors des **mises à jour** du programme ;
- participer à la **documentation** d'une fonction en illustrant une façon de l'utiliser.

Écrire de bons tests demande

- une **compréhension parfaite de la spécification** de la fonction ;
- un moyen de **connaître à l'avance** ce qu'elle doit calculer ;
- une **bonne imagination** pour concevoir des tests significatifs et non redondants.

Il est important que chaque test mette au défi la fonction f testée sur le moins d'aspects possibles à la fois. Ceci est utile car, lorsque le test échoue, il devient facile de délimiter le problème contenu dans l'implantation de f.

Plusieurs méthodes sont possibles pour mettre des tests en place :

- une **approche artisanale** consiste à utiliser une fonction outil de la forme

```
let test nom e r =
    Printf.printf "Test %s : " nom;
    if e = r then
        print_endline "[SUCCES]"
    else
        print_endline "[ECHEC]"
```

  et à disposer dans (par exemple) un fichier `Tests.ml` une fonction principale de test.

Exemple

```
let () =
    test "est_premier 2" (est_premier 2) true;
    test "est_premier 15" (est_premier 15) false;
    test "est_premier 23" (est_premier 23) true
```

- Une approche plus robuste consiste à utiliser des **bibliothèques dédiées** comme `Alcotest`, `MDX` `OUnit` ou `QCheck`. Ceci s'articule bien avec `dune` qui peut accueillir des fichiers de test.

3.13.2. Équivalence de fonctions

Deux fonctions `f1` et `f2` sont *égales*, noté `f1 = f2`, si leur code est **syntaxiquement identique**.

Ce n'est pas une notion très intéressante en pratique.

Deux fonctions `f1` et `f2` sont *équivalentes*, noté `f1 ≡ f2`, si elles sont **extensionnellement équivalentes**. C'est le cas lorsque

- `f1 = f2` si `f1` et `f2` sont des valeurs ;
- pour tout argument `a`, `f1 a ≡ f2 a`, si `f1` et `f2` admettent au moins un argument.

Les fonctions `f1` et `f2` se comportent alors de la même façon : en notant `n` le nombre d'arguments de `f1` et `f2`, pour tous arguments `a1`, ..., `an`, `f1 a1 ...an` et `f2 a1 ...an` renvoient la même valeur.

Exemple

Les fonctions

```
let doubler_1 lst =
    lst |> List.map (fun x -> x * 2)
```

et

```
let rec doubler_2 lst =
    match lst with
        |[] -> []
        |x :: lst' -> x * 2 :: doubler_2 lst'
```

sont équivalentes. Elles sont soumises à **la même spécification** qui est de renvoyer la liste d'entiers en doublant chacun de ses éléments.

Le système de types du langage assure à lui seul une bonne partie de la spécification d'une fonction `f`. Il permet souvent de déterminer des propriétés plus fines que seulement la nature des paramètres et du renvoi de `f`.

Dans certains cas très favorables, le type d'une fonction impose même de manière unique son comportement.

Exemples

- Une fonction `f` de type `'a -> 'a` est forcément équivalente à l'identité.
- Une fonction `f` de type `'a * 'b -> 'a` est forcément équivalente à `fst`.
- Une fonction `f` de type `'a -> ('a -> 'b) -> 'b` est forcément équivalente à la fonction d'application inversée `(|>)`.
- Une fonction `f` de type `'a list -> 'a` renvoie nécessairement l'un des éléments de la liste sur laquelle elle est appliquée.
- Une fonction `f` de type `'a list -> 'a list` renvoie nécessairement une liste constituée d'éléments de la liste sur laquelle elle est appliquée.

En pratique cependant, les types ne suffisent pas pour démontrer qu'une fonction suit une spécification donnée.

Le système de types **polymorphe paramétrique** d'OCaml, bien que puissant et bien meilleur que beaucoup de ses concurrents, est finalement assez limité.

Exemples

En utilisant seulement le système de types, il est impossible d'induite des contraintes sur

- les longueurs des listes (le type des listes de longueurs paires, de longueurs inférieures à n ou de longueur non nulle n'existe pas) ;
- les intervalles auxquels des nombres doivent appartenir ;
- les contraintes structurelles d'une valeur (listes ayant des contraintes locales, comme le fait d'être triées, arbres qui évitent certains motifs).

Il existe des systèmes de types beaucoup plus performants qui le permettent. C'est le cas des **types dépendants**. Dans un tel système de types, il est par exemple possible de spécifier au niveau des types qu'une fonction accepte et renvoie un arbre binaire équilibré.

3.13.3. Démonstrations d'équivalence

Pour **démontrer** qu'une fonction `f` vérifie une spécification, nous démontrons que `f` est équivalente à une fonction g qui est une **fonction hypothétique** vérifiant la spécification.

Commençons par un exemple quasi immédiat.

Exemple

Soit la spécification « la fonction renvoie le dernier élément de la liste non vide ».

Soit la fonction

```
let dernier lst =
    lst |> List.rev |> List.hd
```

Appelons g une fonction hypothétique qui répond à la spécification.

Ainsi, pour toute liste $[e_1, \dots, e_{n-1}, e_n]$ non vide (donc $n \geqslant 1$), nous avons $g\,[e_1, \dots, e_{n-1}, e_n] = e_n$.

D'après la définition de `dernier`, nous avons

$$\texttt{dernier}\,[e_1, \dots, e_{n-1}, e_n] = [e_1, \dots, e_{n-1}, e_n]\,\texttt{|> List.rev |> List.hd}.$$

D'après les spécifications (considérées comme vérifiées) de `List.rev` et `List.hd`, nous avons

$$[e_1, \dots, e_{n-1}, e_n]\,\texttt{|> List.rev |> List.hd} = [e_n, e_{n-1}, \dots, e_1]\,\texttt{|> List.hd} = e_n.$$

Ainsi, les fonctions `dernier` et g sont équivalentes. La fonction `dernier` vérifie donc la spécification annoncée.

Il existe un outil très puissant pour démontrer l'équivalence de fonctions opérant sur des valeurs d'un **type somme** : l'*induction structurelle*. Voici quelques définitions pour l'introduire.

Soit T un **type somme** défini par l'intermédiaire de **constructeurs** C.

Si C est un constructeur du type T,

- l'*arité* de C est le nombre d'arguments de type T qu'il reçoit ;
- si C est un constructeur d'arité n de T et x_1, ..., x_n sont des éléments de type T, alors $C(x_1, \ldots, x_n)$ désigne tout élément de type T obtenu en utilisant le constructeur C avec les arguments x_1, ..., x_n et éventuellement d'autres arguments de types différents de T.

Exemple

Soit le type somme des arbres unaires binaires (sans contrainte supplémentaire ici) :

```
type 'n arbres_unaires_binaires =
    |Feuille of int
    |NoeudUnaire of 'n * ('n arbres_unaires_binaires)
    |NoeudBinaire of 'n * ('n arbres_unaires_binaires) * ('n arbres_unaires_binaires)
```

Les constructeurs `Feuille`, `NoeudUnaire` et `NoeudBinaire` sont d'arités respectives 0, 1 et 2.

Si `t1` et `t2` sont de type `'n arbres_unaires_binaires`, alors `NoeudBinaire(t1, t2)` désigne tout élément de type `'n arbres_unaires_binaires` de la forme `NoeudBinaire (x, t1, t2)` où `x` est de type `'n`.

L'élément `NoeudBinaire (Feuille 1, NoeudUnaire ('x', Feuille 2))` est un exemple d'une telle valeur.

Soit T un type somme et P une propriété sur T.

Un constructeur C d'arité n de T vérifie la *propriété d'induction structurelle* si pour tous éléments x_1, ..., x_n de type T, $P(x_1) \wedge \cdots \wedge P(x_n)$ implique $P(C(x_1, \ldots, x_n))$.

Terminologie :

- lorsque C est d'arité 0, cette propriété est appelée **cas de base** ;
- lorsque C est d'arité $n \geqslant 1$, cette propriété est appellée **cas d'hérédité**.

Le *principe d'induction structurelle* stipule que si tout constructeur de T vérifie la propriété d'induction structurelle, alors $P(x)$ est vrai pour tout élément x de type T.

En définissant adéquatement P relativement à la spécification f d'une fonction, ce principe est un **outil très puissant** pour démontrer que f vérifie cette spécification.

Exemple

Soit le type

```
type arbres =
    |Feuille
    |Noeud of arbres * arbres
```

En notant $\mathrm{n}(\texttt{t})$ (resp. $\mathrm{f}(\texttt{t})$) le nombre de nœuds (resp. feuilles) de `t`, soit la propriété P telle que $P(\texttt{t})$ est vrai si $\mathrm{f}(\texttt{t}) = \mathrm{n}(\texttt{t}) + 1$.

Démontrons $P(\texttt{t})$ pour tout `t` de type `arbres` en utilisant le principe d'induction structurelle.

- (Constructeur `Feuille`.) Nous avons bien $\mathrm{f}(\texttt{Feuille}) = \mathrm{n}(\texttt{Feuille}) + 1$ car $\mathrm{f}(\texttt{Feuille}) = 1$ et $\mathrm{n}(\texttt{Feuille}) = 0$. Ainsi, nous avons bien $P(\texttt{Feuille})$.
- (Constructeur `Noeud`.) Soient `t1` et `t2` deux éléments de type `arbres` tels que $P(\texttt{t1})$ et $P(\texttt{t2})$. Soit $\texttt{t} = \texttt{Noeud (t1, t2)}$. Comme les feuilles d'un arbre binaire sont exclusivement situées dans ses deux sous-arbres, $\mathrm{f}(\texttt{t}) = \mathrm{f}(\texttt{t1}) + \mathrm{f}(\texttt{t2})$. De plus, comme $P(\texttt{t1})$ et $P(\texttt{t2})$,

$$\mathrm{f}(\texttt{t1}) + \mathrm{f}(\texttt{t2}) = \big(\mathrm{n}(\texttt{t1}) + 1\big) + \big(\mathrm{n}(\texttt{t2}) + 1\big) = \mathrm{n}(\texttt{t1}) + \mathrm{n}(\texttt{t2}) + 2$$

  Comme les nœuds d'un arbre binaire sont exclusivement situés dans ses deux sous-arbres ou sa racine, $\mathrm{n}(\texttt{t}) = \mathrm{n}(\texttt{t1}) + \mathrm{n}(\texttt{t2}) + 1$. Ceci montre $\mathrm{f}(\texttt{t}) = \mathrm{n}(\texttt{t}) + 1$ et donc $P(\texttt{t})$.

Une `liste` est un type somme où `[]` est un constructeur d'arité 0 et `::` est un constructeur d'arité 1.

Exemple

Soit la fonction

```
let rec dernier lst =
    match lst with
        |[] -> None
        |[x] -> Some x
        |_ :: lst' -> dernier lst'
```

Soit g la fonction telle que g `lst` renvoie `None` si `lst` est vide et `Some x` sinon, où `x` est le dernier élément de `lst`.

Soit la propriété P telle que $P($`lst`$)$ si `dernier lst` renvoie la même résultat que g `lst`.

Démontrons $P($`lst`$)$ pour toute liste `lst` en utilisant le principe d'induction structurelle.

- (Constructeur `[]`.) L'appel `dernier []` s'évalue en `None`. Ceci montre $P($`[]`$)$.
- (Constructeur `::`.) Soit `lst'` une liste telle que $P\big($`lst'`$\big)$.
  Si `lst'` $=$ `[]`, l'appel `dernier lst` s'évalue en `Some x` où `x` est la tête de `lst`. Comme dans ce cas, `x` est aussi le dernier élément de `lst`, nous avons $P($`lst`$)$.
  Sinon, l'appel `dernier lst` s'évalue en `dernier lst'`. Comme $P\big($`lst'`$\big)$, l'expression `dernier lst'` s'évalue en `Some x'` où `x'` est le dernier élément de `lst'`. Comme `lst'` est la queue de `lst`, ces deux listes ont le même dernier élément. Ainsi, nous avons $P($`lst`$)$.

Considérons le type des *entiers de Peano* défini par

```
type naturels_peano =
    |Zero
    |Succ of naturels_peano
```

Il représente l'ensemble $\mathbb{N}$, où chaque $n \in \mathbb{N}$ est codé par l'élément `Succ (...(Succ (Succ Zero)) ...)` de type `naturels_peano` avec n occurrences de `Succ`. Par exemple, l'entier 2 est codé par `Succ (Succ Zero)`.

Soit P une propriété sur `naturels_peano`. Ainsi, P est une **propriété sur les entiers naturels**.

D'après le principe d'induction structurelle, pour démontrer que $P($`n`$)$ est vrai pour tout élément `n` de type `entiers_peano`, il faut démontrer

- que $P($`Zero`$)$ est vrai (**cas de base**) ;
- que $P($`n`$)$ implique $P($`Succ n`$)$ pour tout élément `n` de type `entiers_peano` (**cas d'hérédité**).

Nous reconnaissons alors le **principe d'induction** habituel.

Il est finalement un **cas particulier** du principe d'induction structurelle.