Compilation

Élimination des données structurées

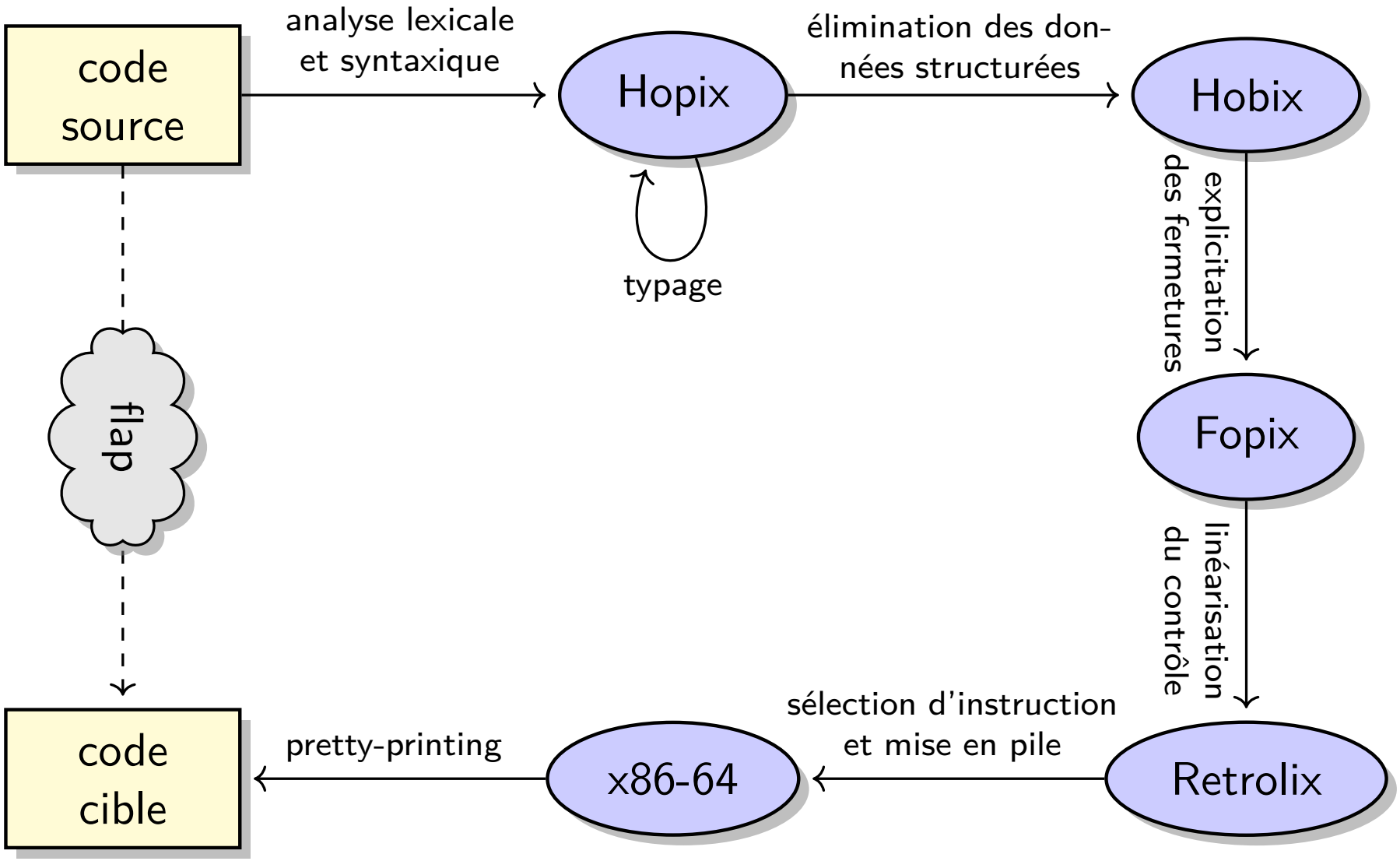
Adrien Guatto

Master 1 Informatique
2022–2023

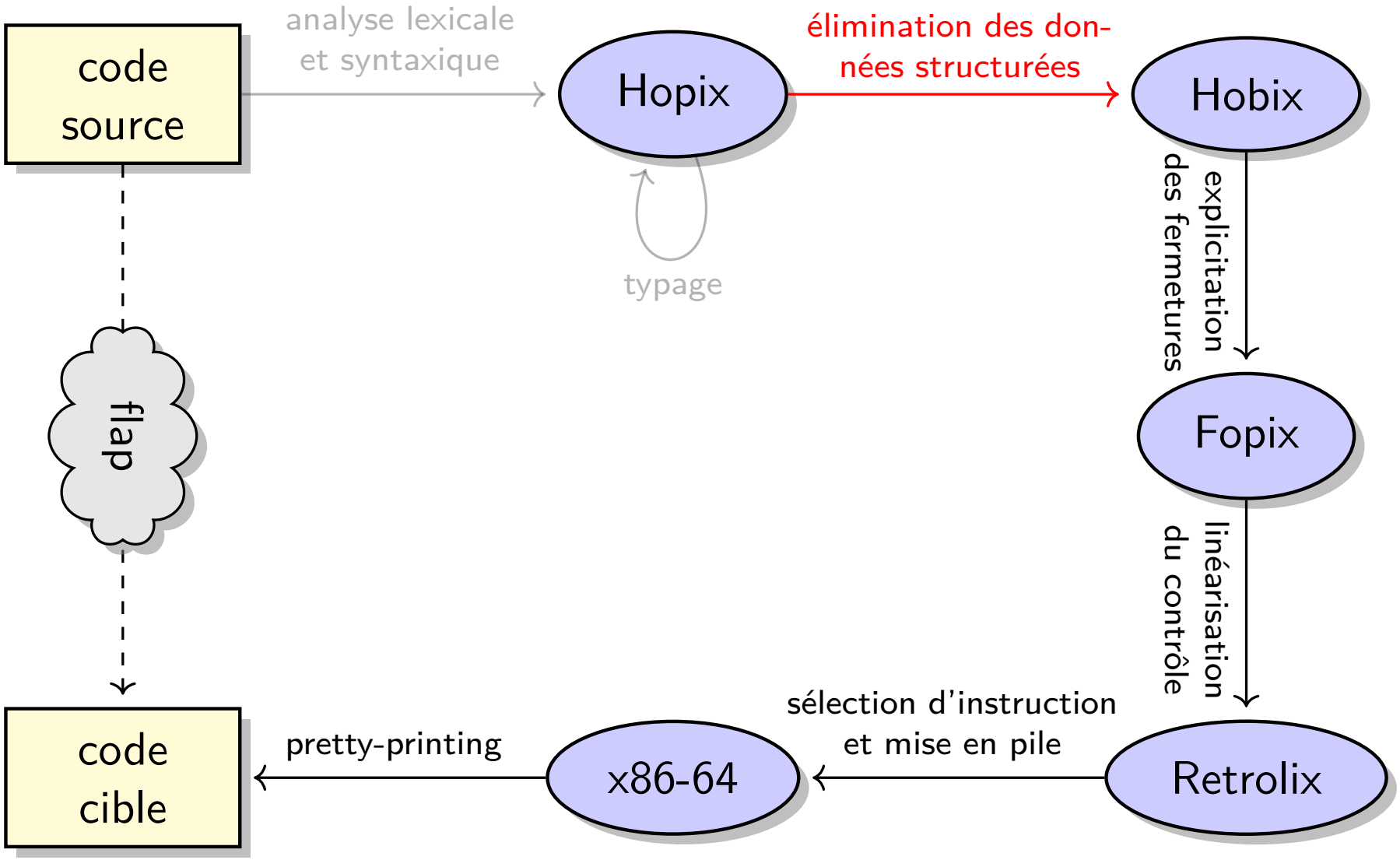
Plan

Plan

Le flot de compilation de flap



Le flot de compilation de flap



Le langage Hobix (1/2)

Le langage Hobix

Suit Hopix dans la chaîne de compilation, donc un peu plus concret.

Le langage Hobix (1/2)

Le langage Hobix

Suit Hopix dans la chaîne de compilation, donc un peu plus concret.

```
type program = definition list

and definition =
  | DeclareExtern of identifier * int
  | DefineValue of value_definition

and value_definition =
  | SimpleValue of identifier * expression
  | RecFunctions of (identifier * expression) list
```

Le langage Hobix (2/2)

```
type expression =
  | Literal of literal
  | Variable of identifier
  | Define of value_definition * expression
  | Apply of expression * expression list
  | IfThenElse of expression * expression * expression
  | Fun of identifier list * expression
  | AllocateBlock of expression
  | WriteBlock of expression * expression * expression
  | ReadBlock of expression * expression
  | Switch of expression * expression option array
              * expression option
  | While of expression * expression

and literal =
  | LInt of Int64.t
  | LString of string
  | LChar of char
```

De Hopix à Hobix

Question

Quelles sont les similarités et différences entre Hopix et Hobix ?

De Hopix à Hobix

Question

Quelles sont les similarités et différences entre Hopix et Hobix ?

Hobix est identique à Hopix, aux différences suivantes près :

- les données structurées et constructions attenantes ont disparu ;

De Hopix à Hobix

Question

Quelles sont les similarités et différences entre Hopix et Hobix ?

Hobix est identique à Hopix, aux différences suivantes près :

- les données structurées et constructions attenantes ont disparu ;
- de même que les boucles `for` et la séquence ;

De Hopix à Hobix

Question

Quelles sont les similarités et différences entre Hopix et Hobix ?

Hobix est identique à Hopix, aux différences suivantes près :

- les données structurées et constructions attenantes ont disparu ;
- de même que les boucles `for` et la séquence ;
- l'information de typage a été effacée ;

De Hopix à Hobix

Question

Quelles sont les similarités et différences entre Hopix et Hobix ?

Hobix est identique à Hopix, aux différences suivantes près :

- les données structurées et constructions attenantes ont disparu ;
- de même que les boucles `for` et la séquence ;
- l'information de typage a été effacée ;
- les fonctions sont n-aires plutôt que unaires ;

De Hopix à Hobix

Question

Quelles sont les similarités et différences entre Hopix et Hobix ?

Hobix est identique à Hopix, aux différences suivantes près :

- les données structurées et constructions attenantes ont disparu ;
- de même que les boucles `for` et la séquence ;
- l'information de typage a été effacée ;
- les fonctions sont n-aires plutôt que unaires ;
- on peut allouer, lire et modifier des blocs ;

De Hopix à Hobix

Question

Quelles sont les similarités et différences entre Hopix et Hobix ?

Hobix est identique à Hopix, aux différences suivantes près :

- les données structurées et constructions attenantes ont disparu ;
- de même que les boucles `for` et la séquence ;
- l'information de typage a été effacée ;
- les fonctions sont n-aires plutôt que unaires ;
- on peut allouer, lire et modifier des blocs ;
- on dispose d'une construction `switch`.

Les types de données structurées

Il s'agit des n-uplets, références, enregistrements et sommes.

Les types de données structurées

Il s'agit des n-uplets, références, enregistrements et sommes.

Question

Pouvez-vous rappeler comment créer et utiliser des valeurs de ces types ?

Type	**Introduction**	**Utilisation**
n-uplet		
Enregistrement		
Somme		
Référence		

Les types de données structurées

Il s'agit des n-uplets, références, enregistrements et sommes.

Question

Pouvez-vous rappeler comment créer et utiliser des valeurs de ces types ?

Type	**Introduction**	**Utilisation**
n-uplet	`(e1, ..., eN)`	`match e with ...`
Enregistrement		
Somme		
Référence		

Les types de données structurées

Il s'agit des n-uplets, références, enregistrements et sommes.

Question

Pouvez-vous rappeler comment créer et utiliser des valeurs de ces types ?

Type	**Introduction**	**Utilisation**
n-uplet	`(e1, ..., eN)`	`match e with ...`
Enregistrement	`{f1 = e1;...;fN = eN}`	`e.fI` / `match e with ...`
Somme		
Référence		

Les types de données structurées

Il s'agit des n-uplets, références, enregistrements et sommes.

Question

Pouvez-vous rappeler comment créer et utiliser des valeurs de ces types ?

Type	**Introduction**	**Utilisation**
n-uplet	`(e1, ..., eN)`	`match e with ...`
Enregistrement	`{f1 = e1;...;fN = eN}`	`e.fI` / `match e with ...`
Somme	`Ki (e1, ..., ek)`	`match e with ...`
Référence		

Les types de données structurées

Il s'agit des n-uplets, références, enregistrements et sommes.

Question

Pouvez-vous rappeler comment créer et utiliser des valeurs de ces types ?

Type	**Introduction**	**Utilisation**
n-uplet	`(e1, ..., eN)`	`match e with ...`
Enregistrement	`{f1 = e1;...;fN = eN}`	`e.fI` / `match e with ...`
Somme	`Ki (e1, ..., ek)`	`match e with ...`
Référence	`ref e`	`!e` / `e := e'`

Les types de données structurées

Il s'agit des n-uplets, références, enregistrements et sommes.

Question

Pouvez-vous rappeler comment créer et utiliser des valeurs de ces types ?

Type	**Introduction**	**Utilisation**
n-uplet	`(e1, ..., eN)`	`match e with ...`
Enregistrement	`{f1 = e1;...;fN = eN}`	`e.fI` / `match e with ...`
Somme	`Ki (e1, ..., ek)`	`match e with ...`
Référence	`ref e`	`!e` / `e := e'`

Par opposition, Hobix ne dispose que d'une notion très frustre de données.

Le modèle mémoire de Hobix

Le tas et les blocs

- Le tas est la partie de la mémoire où vivent les données dont la durée de vie est dynamique, par opposition à la pile (cf. cours ultérieur)
- Le tas contient des blocs, tableaux homogènes de mots machines.
- Chaque mot est soit un entier (ou un caractère), soit un pointeur.
- Chaque pointeur est l'adresse soit d'un bloc, soit d'une chaîne litérale.

Le modèle mémoire de Hobix

Le tas et les blocs

- Le tas est la partie de la mémoire où vivent les données dont la durée de vie est dynamique, par opposition à la pile (cf. cours ultérieur)
- Le tas contient des blocs, tableaux homogènes de mots machines.
- Chaque mot est soit un entier (ou un caractère), soit un pointeur.
- Chaque pointeur est l'adresse soit d'un bloc, soit d'une chaîne litérale.

```
type blockptr (* Un modèle en OCaml, pour fixer les idées. *)
type word = Int of Nativeint.t | Ptr of blockptr
val allocate_block : len:int -> blockptr
val read_block : blockptr -> idx:int -> word
val write_block : blockptr -> idx:int -> value:word -> unit
```

Le modèle mémoire de Hobix

Le tas et les blocs

- Le tas est la partie de la mémoire où vivent les données dont la durée de vie est dynamique, par opposition à la pile (cf. cours ultérieur)
- Le tas contient des blocs, tableaux homogènes de mots machines.
- Chaque mot est soit un entier (ou un caractère), soit un pointeur.
- Chaque pointeur est l'adresse soit d'un bloc, soit d'une chaîne litérale.

```
type blockptr (* Un modèle en OCaml, pour fixer les idées. *)
type word = Int of Nativeint.t | Ptr of blockptr
val allocate_block : len:int -> blockptr
val read_block : blockptr -> idx:int -> word
val write_block : blockptr -> idx:int -> value:word -> unit
```

Remarque importante : pas de sûreté mémoire par typage !

Compilateur faux = programme cible (Hobix) faux = *segmentation fault*.

Le module `HopixToHobix`

Résumé de la tâche de la passe

- Traduire les constructions attenantes aux données structurées en allocation, lecture et écriture de blocs.
- Éliminer les boucles `for` et les séquencements (`e1; ...; eN`).

Le module `HopixToHobix`

Résumé de la tâche de la passe

- Traduire les constructions attenantes aux données structurées en allocation, lecture et écriture de blocs.
- Éliminer les boucles `for` et les séquencements (`e1; ...; eN`).

En pratique :

- la passe prend globalement la forme d'une fonction récursive ;

```
val translate : HopixAST.t -> HobixAST.t
```

- elle procède par un parcours ascendant des arbres de syntaxe Hopix ;
- elle maintient des structures de données utiles à la traduction.

Ces caractéristiques sont partagées par presque toutes les passes de Flap.

Plan

Plan

Le séquencement

```
let rec expression env = function
  ...

  | HopixAST.Sequence es -> (* ??? *)

  ...
```

Pour avancer, il faut exploiter ses intuitions de programmation.

Question

Comment écrire `e1; e2` en OCaml si on ne dispose plus de la séquence ?

`e1; e2`

Le séquencement

```
let rec expression env = function
  ...

  | HopixAST.Sequence es -> (* ??? *)

  ...
```

Pour avancer, il faut exploiter ses intuitions de programmation.

Question

Comment écrire `e1; e2` en OCaml si on ne dispose plus de la séquence ?

`e1; e2` $\equiv$ `let _ = e1 in e2`

Le séquencement

```
let rec expression env = function
  ...

  | HopixAST.Sequence es -> (* ??? *)

  ...
```

Pour avancer, il faut exploiter ses intuitions de programmation.

Question

Comment écrire `e1; e2` en OCaml si on ne dispose plus de la séquence ?

```
e1; e2 ≡ let _ = e1 in e2
       ≡ let x = e1 in e2 (* x pas dans e2 *)
```

Le séquencement

```
let rec expression env = function
  ...

  | HopixAST.Sequence es -> (* ??? *)

  ...
```

Pour avancer, il faut exploiter ses intuitions de programmation.

Question

Comment écrire `e1; e2` en OCaml si on ne dispose plus de la séquence ?

```
e1; e2 ≡ let _ = e1 in e2
       ≡ let x = e1 in e2 (* x pas dans e2 *)
       ≡ match e1 with _ -> e2
```

Le séquencement

```
let rec expression env = function
  ...

  | HopixAST.Sequence es -> (* ??? *)

  ...
```

Pour avancer, il faut exploiter ses intuitions de programmation.

Question

Comment écrire `e1; e2` en OCaml si on ne dispose plus de la séquence ?

```
e1; e2 ≡ let _ = e1 in e2
       ≡ let x = e1 in e2 (* x pas dans e2 *)
       ≡ match e1 with _ -> e2
       ≡ (fun _ -> e2) e1
```

Le séquencement

```
let rec expression env = function
  ...

  | HopixAST.Sequence es -> (* ??? *)

  ...
```

Pour avancer, il faut exploiter ses intuitions de programmation.

Question

Comment écrire `e1; e2` en OCaml si on ne dispose plus de la séquence ?

```
e1; e2 ≡ let _ = e1 in e2
       ≡ let x = e1 in e2 (* x pas dans e2 *)
       ≡ match e1 with _ -> e2
       ≡ (fun _ -> e2) e1
       ≡ ...
```

Le séquencement

```
let rec expression env = function
  ...

  | HopixAST.Sequence es -> (* ??? *)

  ...
```

Pour avancer, il faut exploiter ses intuitions de programmation.

Question

Comment écrire `e1; e2` en OCaml si on ne dispose plus de la séquence ?

```
e1; e2 ≡ let _ = e1 in e2
       ≡ let x = e1 in e2 (* x pas dans e2 *)
       ≡ match e1 with _ -> e2
       ≡ (fun _ -> e2) e1
       ≡ ...
```

On choisira la deuxième expression, disponible et peu coûteuse.

Traduire le séquencement, premier essai

```
module Build = struct
  let fresh_identifier =
    let r = ref 0 in
    fun () -> incr r; HopixAST.Id ("_h2h_" ^ string_of_int !r)
  let (let*) e1 mk_e2 =
    let x = fresh_identifier () in
    HobixAST.(Define (SimpleValue (x, e1), mk_e2 x))
  let seq e1 e2 = let* _ = e1 in e2
end

let rec expression env = function
  ...
  | HopixAST.Sequence es ->
    let es = List.map (expression env) es in
    (* something something List.fold/es/Build.seq ?! *)
  ...
```

Traduire le séquencement, premier essai

```
module Build = struct
  let fresh_identifier =
    let r = ref 0 in
    fun () -> incr r; HopixAST.Id ("_h2h_" ^ string_of_int !r)
  let (let*) e1 mk_e2 =
    let x = fresh_identifier () in
    HobixAST.(Define (SimpleValue (x, e1), mk_e2 x))
  let seq e1 e2 = let* _ = e1 in e2
end

let rec expression env = function
  ...
  | HopixAST.Sequence es ->
    let es = List.map (expression env) es in
    (* something something List.fold/es/Build.seq ?! *)
  ...
```

Question

Pourquoi l'identifiant `Build.fresh_identifier ()` est-il toujours frais ?

Traduire le séquencement, pas à pas

```
let rec expression env = function
  ...
  | HopixAST.Sequence [e1; e2] ->
    let e1 = expression env e1 in
    let e2 = expression env e2 in
    Build.seq (e1, e2)
```

Traduire le séquencement, pas à pas

```
let rec expression env = function
  ...
  | HopixAST.Sequence [e1; e2] ->
    let e1 = expression env e1 in
    let e2 = expression env e2 in
    Build.seq (e1, e2)
  | HopixAST.Sequence [e1; e2; e3] ->
    let e1 = expression env e1 in
    let e2 = expression env e2 in
    let e3 = expression env e3 in
    Build.(seq (seq e1 e2) e3) (* ou bien [seq e1 (seq e2 e3)] ? *)
```

Traduire le séquencement, pas à pas

```
let rec expression env = function
  ...
  | HopixAST.Sequence [e1; e2] ->
    let e1 = expression env e1 in
    let e2 = expression env e2 in
    Build.seq (e1, e2)
  | HopixAST.Sequence [e1; e2; e3] ->
    let e1 = expression env e1 in
    let e2 = expression env e2 in
    let e3 = expression env e3 in
    Build.(seq (seq e1 e2) e3) (* ou bien [seq e1 (seq e2 e3)] ? *)
  | HopixAST.Sequence [e1; e2; e3; e4] ->
    let e1 = expression env e1 in
    let e2 = expression env e2 in
    let e3 = expression env e3 in
    let e4 = expression env e4 in
    Build.(seq (seq (seq e1 e2) e3) e4) (* ou bien... etc. *)
```

Traduire le séquencement, une option

```
let rec expression env = function
  ...
  | HopixAST.Sequence es ->
    (* T(e1; e2; ... eN)
     = val _ = (); val _ = T(e1); val _ = T(e2); ... ; T(eN) *)
    let es = List.map (expression env) es in
    List.fold_left Build.seq (Build.mk_unit ()) es
  ...
```

Traduire le séquencement, une option

```
let rec expression env = function
  ...
  | HopixAST.Sequence es ->
    (* T(e1; e2; ... eN)
     = val _ = (); val _ = T(e1); val _ = T(e2); ... ; T(eN) *)
    let es = List.map (expression env) es in
    List.fold_left Build.seq (Build.mk_unit ()) es
  ...
```

Ce code est beau, simple, général et correct.

Traduire le séquencement, une option

```
let rec expression env = function
  ...
  | HopixAST.Sequence es ->
    (* T(e1; e2; ... eN)
     = val _ = (); val _ = T(e1); val _ = T(e2); ... ; T(eN) *)
    let es = List.map (expression env) es in
    List.fold_left Build.seq (Build.mk_unit ()) es
  ...
```

Ce code est beau, simple, général et correct.

Que peut-on bien lui reprocher ?

Traduire le séquencement, une option

```
let rec expression env = function
  ...
  | HopixAST.Sequence es ->
    (* T(e1; e2; ... eN)
     = val _ = (); val _ = T(e1); val _ = T(e2); ... ; T(eN) *)
    let es = List.map (expression env) es in
    List.fold_left Build.seq (Build.mk_unit ()) es
  ...
```

Ce code est beau, simple, général et correct.

Que peut-on bien lui reprocher ?

- Il fabrique systématiquement une valeur de type `unit` (cf. *infra*).

Traduire le séquencement, une option

```
let rec expression env = function
  ...
  | HopixAST.Sequence es ->
    (* T(e1; e2; ... eN)
     = val _ = (); val _ = T(e1); val _ = T(e2); ... ; T(eN) *)
    let es = List.map (expression env) es in
    List.fold_left Build.seq (Build.mk_unit ()) es
  ...
```

Ce code est beau, simple, général et correct.

Que peut-on bien lui reprocher ?

- Il fabrique systématiquement une valeur de type `unit` (cf. *infra*).
- Le cas unaire n'est pas optimisé.

Traduire le séquencement, une option

```
let rec expression env = function
  ...
  | HopixAST.Sequence es ->
    (* T(e1; e2; ... eN)
     = val _ = (); val _ = T(e1); val _ = T(e2); ... ; T(eN) *)
    let es = List.map (expression env) es in
    List.fold_left Build.seq (Build.mk_unit ()) es
  ...
```

Ce code est beau, simple, général et correct.

Que peut-on bien lui reprocher ?

- Il fabrique systématiquement une valeur de type `unit` (cf. *infra*).
- Le cas unaire n'est pas optimisé.
- Le cas nullaire est inutile, du moins au sortir de l'analyseur syntaxique.

Traduire le séquencement, une autre option

```
(* À mettre dans un module utilitaire, e.g., utilities/list.ml. *)
let fold_left_1 f xs = match xs with
  | [] -> invalid_arg "fold_left_1"
  | x :: xs -> List.fold_left f x xs

...

let rec expression env = function
  ...
  | HopixAST.Sequence es ->
    (* INVARIANT [es] is never empty. See parser in particular. *)
    List.map (located (expression env)) es |> fold_left_1 Build.seq
  ...
```

Plan

Les boucles dénombrées

```
let rec expression env = function
  ...
  | HopixAST.For (x, lo, hi, body) -> (* ??? *)
  ...
```

Question

Comment écrire `for x = lo to hi do body done` sans boucles `for` ?

Se passer de boucles dénombrées en OCaml

Question

Comment écrire `for x = lo to hi do body done` sans boucles `for` ?

Se passer de boucles dénombrées en OCaml

Question

Comment écrire `for x = lo to hi do body done` sans boucles `for` ?

```
let _y = ref lo in   (* _y frais *)
while !_y <= hi do
  let x = !_y in
  body;
  incr _y
done
```

Se passer de boucles dénombrées en OCaml

Question

Comment écrire `for x = lo to hi do body done` sans boucles `for` ?

```
let _y = ref lo in   (* _y frais *)
while !_y <= hi do
  let x = !_y in
  body;
  incr _y
done
```

Cette traduction est fausse. Pourquoi ?

Se passer de boucles dénombrées en OCaml

Question

Comment écrire `for x = lo to hi do body done` sans boucles `for` ?

```
let _y = ref lo in   (* _y frais *)
while !_y <= hi do
  let x = !_y in
  body;
  incr _y
done
```

Cette traduction est fausse. Pourquoi ? Il répète l'évaluation de `hi`.

```
(* Qu'affiche ce code ? Et le code traduit ? *)
let r = ref 0 in
for x = 0 to (incr r; 10) do () done;
print_int !r
```

Se passer pour de bon des boucles dénombrées en OCaml

Question

Comment écrire `for x = lo to hi do body done` sans boucles `for` ?

Se passer pour de bon des boucles dénombrées en OCaml

Question

Comment écrire `for x = lo to hi do body done` sans boucles `for` ?

```
let i = ref lo in   (* i et j frais *)
let j = hi in
while !i <= j do
  let x = !i in
  body;
  incr i
done
```

Traduire les boucles dénombrées

```
let rec expression env = function
  ...

  | HopixAST.For (x, lo, hi, body) ->
    let open Build in
    let* i = ref_ (expression env lo) in
    let* j = expression env hi
    while_ (deref i <= j) begin
      let* _ = letval x (deref i) (expression env body) in
      assign i (deref i + int 1)
    end

  ...
```

Il faut maintenant implémenter les fonctions `Build.ref_`, `Build.deref`, etc.

Plan

Les types de données structurées

Type	**Introduction**	**Utilisation**
n-uplet	`(e1, ..., eN)`	`match e with ...`
Enregistrement	`{f1 = e1;...;fN = eN}`	`e.fI` / `match e with ...`
Somme	`Ki (e1, ..., ek)`	`match e with ...`
Référence	`ref e`	`!e` / `e := e'`

Quelle stratégie d'implémentation via `allocate`/`read`/`write_block` ?

- Chaque introduction alloue *et initialise* un nouveau bloc.
- Chaque utilisation accède au bloc et, si besoin est, le modifie.

On va procéder du simple aux complexe (`match` !).

Se passer de références en OCaml

Question

Imaginons un OCaml sans références mais muni de primitives de manipulation de blocs. Comment écrire `ref e`, `!e` et `e1 := e2` ?

Se passer de références en OCaml

Question

Imaginons un OCaml sans références mais muni de primitives de manipulation de blocs. Comment écrire `ref e`, `!e` et `e1 := e2` ?

```
(* ref e *)
let _x = allocate_block 1 in (write_block _x 0 e; _x)
```

Se passer de références en OCaml

Question

Imaginons un OCaml sans références mais muni de primitives de manipulation de blocs. Comment écrire `ref e`, `!e` et `e1 := e2` ?

```
(* ref e *)
let _x = allocate_block 1 in (write_block _x 0 e; _x)

(* !e *)
read_block e 0
```

Se passer de références en OCaml

Question

Imaginons un OCaml sans références mais muni de primitives de manipulation de blocs. Comment écrire `ref e`, `!e` et `e1 := e2` ?

```
(* ref e *)
let _x = allocate_block 1 in (write_block _x 0 e; _x)

(* !e *)
read_block e 0

(* e1 := e2 *)
write_block e1 0 e2
```

Traduire les références

```
module Build = struct
  ...
  let ref_ v = let* x = allocate_block (int 1) in
               let* _ = write_block x (int 0) v in
               x

  let deref x = read_block x (int 0)

  let assign x y = write_block x (int 0) y
end

let rec expression env = function
  ...
  | HopixAST.Ref e -> Build.ref_ (expression env e)
  | HopixAST.Read e -> Build.deref (expression env e)
  | HopixAST.Assign (e1, e2) ->
    Build.assign (expression env e1) (expression env e2)
  ...
```

Se passer de n-uplets

Question

Comment écrire `(e1, ..., eN)` en manipulant des blocs ?

Se passer de *n*-uplets

Question

Comment écrire `(e1, ..., eN)` en manipulant des blocs ?

```
(* (e1, ..., eN) *)
let _x = allocate_block N in
write_block _x 0 e1;
...;
write_block _x (N-1) eN;
_x
```

On en déduit facilement le code à ajouter dans `HopixToHobix.expression`.

Se passer de *n*-uplets

Question

Comment écrire `(e1, ..., eN)` en manipulant des blocs ?

```
(* (e1, ..., eN) *)
let _x = allocate_block N in
write_block _x 0 e1;
...;
write_block _x (N-1) eN;
_x
```

On en déduit facilement le code à ajouter dans `HopixToHobix.expression`.

Et le filtrage ?

On traitera le filtrage après avoir vu toutes les autres constructions.

Se passer des créations d'enregistrements

```
type foo = { a : int; b : int list; c : string; }
let x = { a = 12; b = [1; 2]; c = "toto"; }
let y = { a = 12; c = "toto"; b = []; }
let z = let r = ref 0 in
        { c = (r := 1; "toto"); a = !r; b = (r := 2; []); }
```

Question

Comment traduire systématiquement ces expressions en termes de blocs ?

Se passer des créations d'enregistrements

```
type foo = { a : int; b : int list; c : string; }
let x = { a = 12; b = [1; 2]; c = "toto"; }
let y = { a = 12; c = "toto"; b = []; }
let z = let r = ref 0 in
        { c = (r := 1; "toto"); a = !r; b = (r := 2; []); }
```

Question

Comment traduire systématiquement ces expressions en termes de blocs ?

```
let x = let _x0 = allocate_block 3 in
        write_block 0 12; write_block 1 [1; 2];
        write_block 2 "toto"; _x0
```

Se passer des créations d'enregistrements

```
type foo = { a : int; b : int list; c : string; }
let x = { a = 12; b = [1; 2]; c = "toto"; }
let y = { a = 12; c = "toto"; b = []; }
let z = let r = ref 0 in
        { c = (r := 1; "toto"); a = !r; b = (r := 2; []); }
```

Question

Comment traduire systématiquement ces expressions en termes de blocs ?

```
let x = let _x0 = allocate_block 3 in
        write_block 0 12; write_block 1 [1; 2];
        write_block 2 "toto"; _x0
let y = let _x1 = allocate_block 3 in
        write_block _x1 0 12; write_block _x1 2 "toto";
        write_block _x1 1 []; _x1
```

Se passer des créations d'enregistrements

```
type foo = { a : int; b : int list; c : string; }
let x = { a = 12; b = [1; 2]; c = "toto"; }
let y = { a = 12; c = "toto"; b = []; }
let z = let r = ref 0 in
        { c = (r := 1; "toto"); a = !r; b = (r := 2; []); }
```

Question

Comment traduire systématiquement ces expressions en termes de blocs ?

```
let x = let _x0 = allocate_block 3 in
        write_block 0 12; write_block 1 [1; 2];
        write_block 2 "toto"; _x0
let y = let _x1 = allocate_block 3 in
        write_block _x1 0 12; write_block _x1 2 "toto";
        write_block _x1 1 []; _x1
let z = let _x2 = allocate-block 3 in
        write_block _x2 2 (r := 1; "toto"); write_block _x2 0 !r;
        write_block _x2 1 (r := 2; []); _x2
```

Se passer des créations d'enregistrements

```
type foo = { a : int; b : int list; c : string; }
let x = { a = 12; b = [1; 2]; c = "toto"; }
let k = x.c
```

Question

Comment traduire une projection ?

Se passer des créations d'enregistrements

```
type foo = { a : int; b : int list; c : string; }
let x = { a = 12; b = [1; 2]; c = "toto"; }
let k = x.c
```

Question

Comment traduire une projection ?

```
let k = read_block x 2
```

Se passer des créations d'enregistrements

```
type foo = { a : int; b : int list; c : string; }
let x = { a = 12; b = [1; 2]; c = "toto"; }
let k = x.c
```

Question

Comment traduire une projection ?

```
let k = read_block x 2
```

Bilan

- La traduction doit pouvoir associer à chaque champ de chaque type enregistrement sa position dans les blocs qui l'implémentent.
- Le choix initial des positions est arbitraire mais doit être cohérent.

Traduire les références

```
let rec expression env = function
  ...
  | HopixAST.Field (e, l) ->
    let open Build in
    read_block (expression env e) (int (lookup_field_position env l))
  | HopixAST.Record fes ->
    let open Build in
    let* x = allocate_block (int (List.length fes)) in
    let field (l, e) e' =
      let* _ = write_block x (int (lookup_field_position env l))
                 (expression env e)
      in e'
    in
    List.fold_right field fes x
  ...

let type_definition env = function
  | HopixAST.DefineRecordType fs -> extend_with_field_positions env fs
  ...
```

Traduire les sommes

```
type ('a, 'b) t = Left of 'a | Right of 'b
```

Question

Comment représenter une valeur de type `('a, 'b) t` sous forme de blocs ?

Traduire les sommes

```
type ('a, 'b) t = Left of 'a | Right of 'b
```

Question

Comment représenter une valeur de type `('a, 'b) t` sous forme de blocs ?

On peut utiliser la solution standard des langages dépourvus de types sommes.

Les sommes en C

```
type ('a, 'b) t = Left of 'a | Right of 'b
```

Question

Comment implémentez-vous ce type en langage C ?

Les sommes en C

```
type ('a, 'b) t = Left of 'a | Right of 'b
```

Question

Comment implémentez-vous ce type en langage C ?

```
typedef enum { LEFT = 0, RIGHT } tag_t;
typedef struct t {
  tag_t tag;
  union { struct { void *p0; } left;
          struct { void *p0; } right; } params;
} t_t;
```

Les sommes en C

```
type ('a, 'b) t = Left of 'a | Right of 'b
```

Question

Comment implémentez-vous ce type en langage C ?

```
typedef enum { LEFT = 0, RIGHT } tag_t;
typedef struct t {
  tag_t tag;
  union { struct { void *p0; } left;
          struct { void *p0; } right; } params;
} t_t;
```

On trouve le même codage dans une toute autre littérature...

Aparté : le codage Bourbaki-Kernighan-Ritchie des sommes

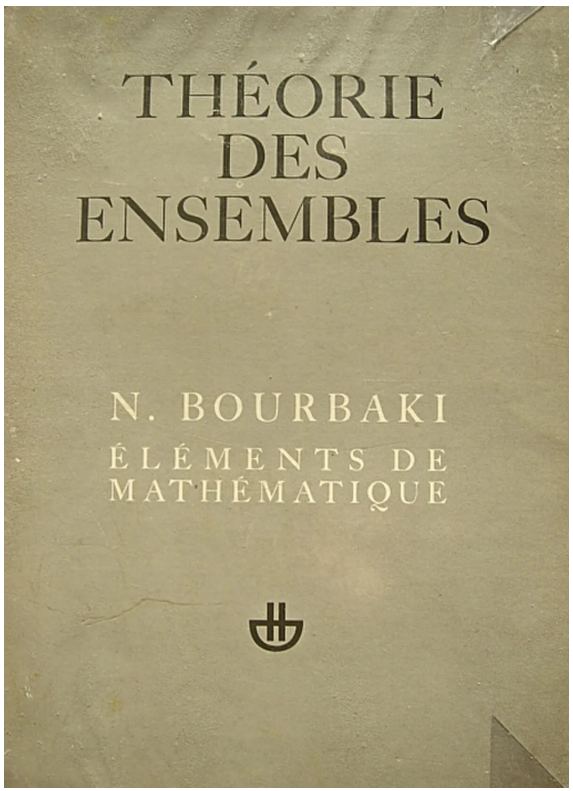
Définition (N. Bourbaki, Théorie des Ensembles (1970), II.30)

Soit $(X_i)_{i \in I}$ une famille d'ensembles. On appelle *somme* de cette famille d'ensembles la réunion de la famille des ensembles $\{i\} \times X_i$ $(i \in I)$.

Aparté : le codage Bourbaki-Kernighan-Ritchie des sommes

Définition (N. Bourbaki, Théorie des Ensembles (1970), II.30)

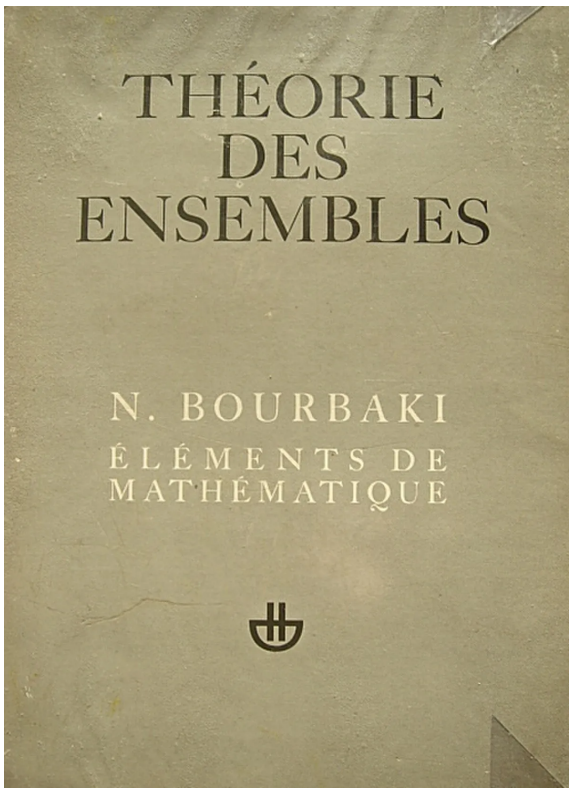
Soit $(X_i)_{i \in I}$ une famille d'ensembles. On appelle *somme* de cette famille d'ensembles la réunion de la famille des ensembles $\{i\} \times X_i$ $(i \in I)$.



Aparté : le codage Bourbaki-Kernighan-Ritchie des sommes

Définition (N. Bourbaki, Théorie des Ensembles (1970), II.30)

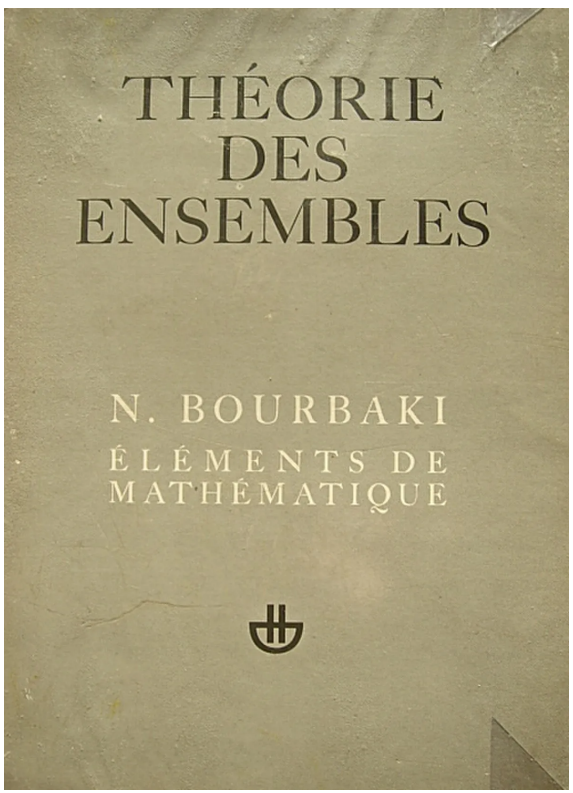
Soit $(X_i)_{i \in I}$ une famille d'ensembles. On appelle *somme* de cette famille d'ensembles la réunion de la famille des ensembles $\{i\} \times X_i$ $(i \in I)$.



Aparté : le codage Bourbaki-Kernighan-Ritchie des sommes

Définition (N. Bourbaki, Théorie des Ensembles (1970), II.30)

Soit $(X_i)_{i \in I}$ une famille d'ensembles. On appelle *somme* de cette famille d'ensembles la réunion de la famille des ensembles $\{i\} \times X_i$ $(i \in I)$.



Traduire les appels de constructeurs

```
type t = Var of string | App of t * t | Fun of string * t
let x = Fun ("x", Var "y")
```

Question

Comment traduire le code de x en manipulation de blocs, en utilisant le codage “Bourbaki-Kernighan-Ritchie” des transparents précédents ?

Traduire les appels de constructeurs

```
type t = Var of string | App of t * t | Fun of string * t
let x = Fun ("x", Var "y")
```

Question

Comment traduire le code de x en manipulation de blocs, en utilisant le codage “Bourbaki-Kernighan-Ritchie” des transparents précédents ?

```
let x =
  let _x0 = allocate_block 3 in
  write_block _x0 0 2;
  write_block _x0 1 "x";
  write_block _x0 2 (let _x1 = allocate_block 2 in
                     write_block _x1 0 0;
                     write_block _x1 1 "y";
                     _x1);
  _x0
```

Traduire les appels de constructeurs

```
let rec expression env = function
  ...
  | HopixAST.Tagged (k, _, es) ->
    let open Build in
    let* x = allocate_block (int (List.length es + 1)) in
    let* _ = write_block x (int 0) (lookup_constructor_index env k) in
    let write_arg e (i, e') =
      i + 1, let* _ = write_block x (int i) (expression env e) in e'
    in
    let _, e' = List.fold_right write_arg es (1, x) in
    e'
  ...
```

Le filtrage, balbutiements

```
let rec length xs =
  match xs with
  | Nil -> 0
  | Cons (_, xs) -> 1 + length xs
```

Question

Comment traduire ce filtrage en termes de blocs ?

Le filtrage, balbutiements

```
let rec length xs =
  match xs with
  | Nil -> 0
  | Cons (_, xs) -> 1 + length xs
```

Question

Comment traduire ce filtrage en termes de blocs ?

```
let rec length xs =
  if read_block xs 0 = 0 then 0
  else 1 + length (read_block xs 2)
```

Le filtrage, balbutiements

```
let rec length xs =
  match xs with
  | Nil -> 0
  | Cons (_, xs) -> 1 + length xs
```

Question

Comment traduire ce filtrage en termes de blocs ?

```
let rec length xs =
  if read_block xs 0 = 0 then 0
  else 1 + length (read_block xs 2)

(* C'est une bonne idée d'être plus systématique. *)
let rec length xs =
  if read_block xs 0 = 0 then 0
  else if read_block xs 0 = 1 then (let xs = read_block xs 2 in
                                    1 + length xs)
  else dummy (* valeur quelconque, jamais atteinte *)
```

Compilation naïve du filtrage

Quelle traduction simple pour l'expression OCaml

```
match e with p1 -> e1 | p2 -> e2 | ... | pN -> eN
```

dans le cas général ?

Compilation naïve du filtrage

Quelle traduction simple pour l'expression OCaml

```
match e with p1 -> e1 | p2 -> e2 | ... | pN -> eN
```

dans le cas général ?

```
let _x = e in
if (* _x filtré par p1 *) then let (* liaisons de p1 *) in e1
else if (* _x filtré par p2 *) then let (* liaisons de p2 *) in e2
else ...
else if (* _x filtré par pN *) then let (* liaisons de pN *) in eN
else dummy (* valeur quelconque, jamais atteinte *)
```

Compilation naïve du filtrage

Quelle traduction simple pour l'expression OCaml

```
match e with p1 -> e1 | p2 -> e2 | ... | pN -> eN
```

dans le cas général ?

```
let _x = e in
if (* _x filtré par p1 *) then let (* liaisons de p1 *) in e1
else if (* _x filtré par p2 *) then let (* liaisons de p2 *) in e2
else ...
else if (* _x filtré par pN *) then let (* liaisons de pN *) in eN
else dummy (* valeur quelconque, jamais atteinte *)
```

Question

Quel est le type de la fonction de traduction des motifs ?

Traduction des motifs pour la compilation naïve du filtrage

```
val pattern : env -> HopixAST.pattern -> HobixAST.expression ->
              HobixAST.(expression * (id * expression) list)
```

L'évaluation de `pattern env p d = (e, defs)` produit :

- l'expression booléenne Hobix `e` qui indique si `p` filtre `d`,
- les liaisons produites par `p` le cas échéant.

Traduction des motifs pour la compilation naïve du filtrage

```
val pattern : env -> HopixAST.pattern -> HobixAST.expression ->
              HobixAST.(expression * (id * expression) list)
```

L'évaluation de `pattern env p d = (e, defs)` produit :

- l'expression booléenne Hobix `e` qui indique si `p` filtre `d`,
- les liaisons produites par `p` le cas échéant.

```
let rec pattern env p d =
  let open Build in
  match p with
  | PWildcard -> (true_, [])
  | PVar x -> (true_, [x, d])
  | PLiteral l -> (d = lit l, []) (* (=) redéfini dans Build *)
  | PTaggedValue (k, _, ps) -> (* À votre avis ? *)
  ...
```

Plan

Y a-t-il des choses à optimiser ?

Y a-t-il des choses à optimiser ?

Folklore des implémenteurs de langages

L'efficacité du filtrage est critique pour certains programmes fonctionnels.

Y a-t-il des choses à optimiser ?

Folklore des implémenteurs de langages

L'efficacité du filtrage est critique pour certains programmes fonctionnels.

- Ce sujet a été étudié en détail depuis longtemps.
- Le Fessant et Maranget (2001) et surtout Maranget (2008) donnent une introduction accessible.
- On ne peut donner qu'une mise en bouche ici.

Naïveté de notre traduction du filtrage, un exemple

```
let f x y z = match x, y, z with
  | _, F, T -> 1
  | F, T, _ -> 2
  | _, _, F -> 3
  | _, _, T -> 4
```

Notre traduction produit essentiellement le code suivant.

```
if                y = F && z = T then 1
else if x = F && y = T           then 2
else if                  z = F then 3
else if                  z = T then 4
else error
```

- Notre traduction traite chaque branche indépendamment.
- En conséquence, on effectue beaucoup de tests redondants.

Ne tester chaque variable qu'une fois

```
let f x y z = match x, y, z with
  | _, F, T -> 1 | F, T, _ -> 2
  | _, _, F -> 3 | _, _, T -> 4
```

Ne tester chaque variable qu'une fois

```
let f x y z = match x, y, z with
  | _, F, T -> 1 | F, T, _ -> 2
  | _, _, F -> 3 | _, _, T -> 4
```

```
if x = T then
  if y = T then
     if z = T then 4 else 3
  else (* y = F *) then
     if z = T then 1 else 3
else (* x = F *)
  if y = T then
     2
  else (* y = F *)
     if z = T then
       1
     else (* z = F *)
       3
```

Ne tester chaque variable qu'une fois

```
let f x y z = match x, y, z with
  | _, F, T -> 1 | F, T, _ -> 2
  | _, _, F -> 3 | _, _, T -> 4
```

Ne tester chaque variable qu'une fois

```
let f x y z = match x, y, z with
  | _, F, T -> 1 | F, T, _ -> 2
  | _, _, F -> 3 | _, _, T -> 4


if y = T then
  if x = T then
    if z = T then 4 else 3
  else 2
else
  if z = T then 1 else 3
```

Plan

Conclusion

- On réduit les données structurées à des primitives de bas niveau.
- Celles-ci sont dangereuses, d'où leur absence du langage source.
- La traduction doit être correcte, et peut être optimisée (`match`).

Conclusion

- On réduit les données structurées à des primitives de bas niveau.
- Celles-ci sont dangereuses, d'où leur absence du langage source.
- La traduction doit être correcte, et peut être optimisée (`match`).

La suite du flot de compilation

- Les données structurées ont été éliminées et ne sont plus une question.
- La prochaine étape consiste à éliminer les fonctions anonymes.

Bibliographie


Le Fessant, Fabrice et Luc Maranget (2001). “Optimizing Pattern Matching”. In : *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*. Sous la dir. de Benjamin C. Pierce. ACM, p. 26-37. DOI : 10.1145/507635.507641. URL : https://pauillac.inria.fr/~maranget/papers/opat/.


Maranget, Luc (2008). “Compiling Pattern Matching to Good Decision Trees”. In : *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*. ML '08. Victoria, BC, Canada : Association for Computing Machinery, p. 35-46. ISBN : 9781605580623. DOI : 10.1145/1411304.1411311. URL : http://moscova.inria.fr/~maranget/papers/ml05e-maranget.pdf.