Compilation

Interprétation

Adrien Guatto

Master 1 Informatique
2022–2023

Plan

1. Introduction
2. Équivalence syntaxique et substitution
3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs
4. Conclusion

Plan

1. Introduction
2. Équivalence syntaxique et substitution
3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs
4. Conclusion

La sémantique des langages de programmation

La sémantique des langages de programmation

Quoi ?

La description mathématique du comportement des programmes.

La sémantique des langages de programmation

Quoi ?

La description mathématique du comportement des programmes.

Pourquoi ?

- Mieux programmer en comprenant très précisément ses programmes.
- Vérifier la correction des programmes et des outils qui les manipulent.
- Guider la conception de nouveaux langages et outils.

La sémantique des langages de programmation

Quoi ?

La description mathématique du comportement des programmes.

Pourquoi ?

- Mieux programmer en comprenant très précisément ses programmes.
- Vérifier la correction des programmes et des outils qui les manipulent.
- Guider la conception de nouveaux langages et outils.

Comment ?

Ce cours : une approche pratique, illustrée par la pratique.

Sémantique et équivalence

La question clef

Sous quelle condition peut-on dire que deux programmes sont équivalents ?

Sémantique et équivalence

La question clef

Sous quelle condition peut-on dire que deux programmes sont équivalents ?

Deux notions d'égalité

- Syntaxique : les programmes représentent le même arbre de syntaxe ;
  - Déjà pas si trivial que ça à cause des variables liées.
- Sémantique : les programmes "calculent la même chose".
  - Plusieurs façons de définir ce qu'un programme calcule.

Sémantique et équivalence

La question clef

Sous quelle condition peut-on dire que deux programmes sont équivalents ?

Deux notions d'égalité

- Syntaxique : les programmes représentent le même arbre de syntaxe ;
  - Déjà pas si trivial que ça à cause des variables liées.
- Sémantique : les programmes "calculent la même chose".
  - Plusieurs façons de définir ce qu'un programme calcule.

Le reste de la séance : illustrer sur des langages miniatures avec du code.

Plan

1. Introduction
2. Équivalence syntaxique et substitution
3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs
4. Conclusion

Plan

1. Introduction

2. Équivalence syntaxique et substitution

3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs

4. Conclusion

Qu'est-ce qu'un programme ?

Qu'est-ce qu'un programme ?

Une première approximation

- L'espacement et les commentaires n'entrent pas en ligne de compte.
- Un programme est donc un arbre de syntaxe abstraite.

Qu'est-ce qu'un programme ?

Une première approximation

- L'espacement et les commentaires n'entrent pas en ligne de compte.
- Un programme est donc un arbre de syntaxe abstraite.

Par exemple, les arbres de syntaxe abstraite du mini-langage $\mathrm{L0}$:

```
type t = Var of Id.t                (* occurrence "x", "y", etc. *)
       | Int of int                 (* entier litéral "42", etc. *)
       | Add of t * t               (* somme "e1 + e2" *)
       | Let of t * bound1          (* déf. locale "let e1 be x in e2" *)
and bound1 = { bound : Id.t; body : t; } (* liaison "x.e" *)
```

Qu'est-ce qu'un programme ?

Une première approximation

- L'espacement et les commentaires n'entrent pas en ligne de compte.
- Un programme est donc un arbre de syntaxe abstraite.

Par exemple, les arbres de syntaxe abstraite du mini-langage $\mathrm{L0}$:

```
type t = Var of Id.t                (* occurrence "x", "y", etc. *)
       | Int of int                 (* entier litéral "42", etc. *)
       | Add of t * t               (* somme "e1 + e2" *)
       | Let of t * bound1          (* déf. locale "let e1 be x in e2" *)
and bound1 = { bound : Id.t; body : t; } (* liaison "x.e" *)
```

Un problème

L'égalité brute des arbres de syntaxe abstraite est trop fine. Par exemple,

- `Let (Int 1, { bound = "x", body = Add (Var "x", Int 2); })`
- `Let (Int 1, { bound = "y", body = Add (Var "y", Int 2); })`

devraient être considérés équivalents. C'est la notion d'α-*équivalence*.

Occurrences libres et liées de variables

Définition (Occurrence d'une variable)

Soient x une variable et e une expression. Une *occurrence de* x *dans* e est la position dans e d'une sous-expression de la forme `Var` x.

Occurrences libres et liées de variables

Définition (Occurrence d'une variable)

Soient x une variable et e une expression. Une *occurrence de* x *dans* e est la position dans e d'une sous-expression de la forme `Var x`.

L'unique occurrence de x dans l'expression

```
Let (Int 1, { bound = "y"; body = Add (Var "x", Int 1); })
```

est de nature syntaxiquement différente de celle dans

```
Let (Int 1, { bound = "x"; body = Add (Var "x", Int 1); })
```

dans la mesure où, intuitivement :

- la première se réfère au contexte ambiant et est dite libre, tandis que
- la seconde nomme une sous-expression (`Int 1`) et est donc dite liée.

Seule la première peut être substituée (remplacée) si besoin est.

Calcul des variables et variables libres occurrentes

Calcul des variables et variables libres occurrentes

```
let rec occurring : L0.t -> Id.Set.t =
  let open Id.Set in function
  | Var x -> singeton x
  | Int _ -> empty
  | Add (e1, e2) -> union (occurring e1) (occurring e2)
  | Let (e, b) -> union (occurring e) (occurring b.body)
```

Calcul des variables et variables libres occurrentes

```
let rec occurring : L0.t -> Id.Set.t =
  let open Id.Set in function
  | Var x -> singeton x
  | Int _ -> empty
  | Add (e1, e2) -> union (occurring e1) (occurring e2)
  | Let (e, b) -> union (occurring e) (occurring b.body)

let rec occurring_free : L0.t -> Id.Set.t =
  let open Id.Set in function
  | Var x -> singeton x
  | Int _ -> empty
  | Add (e1, e2) -> union (occurring_free e1) (occurring_free e2)
  | Let (e, b) -> union (occurring_free e) (occurring_free_b1 b)
and occurring_free_b1 : L0.bound1 -> Id.Set.t =
  fun { bound; body; } -> Id.Set.remove bound (occurring_free body)
```

Calcul des variables et variables libres occurrentes

```
let rec occurring : L0.t -> Id.Set.t =
  let open Id.Set in function
  | Var x -> singeton x
  | Int _ -> empty
  | Add (e1, e2) -> union (occurring e1) (occurring e2)
  | Let (e, b) -> union (occurring e) (occurring b.body)

let rec occurring_free : L0.t -> Id.Set.t =
  let open Id.Set in function
  | Var x -> singeton x
  | Int _ -> empty
  | Add (e1, e2) -> union (occurring_free e1) (occurring_free e2)
  | Let (e, b) -> union (occurring_free e) (occurring_free_b1 b)
and occurring_free_b1 : L0.bound1 -> Id.Set.t =
  fun { bound; body; } -> Id.Set.remove bound (occurring_free body)
```

Remarque

`Id.Set.subset (occurring_free e) (occurring e)` est toujours `true`.

L'égalité à renommage des occurrences liées près

Une occurence liée n'a pas d'identité : elle ne fait que référence à un lieur.

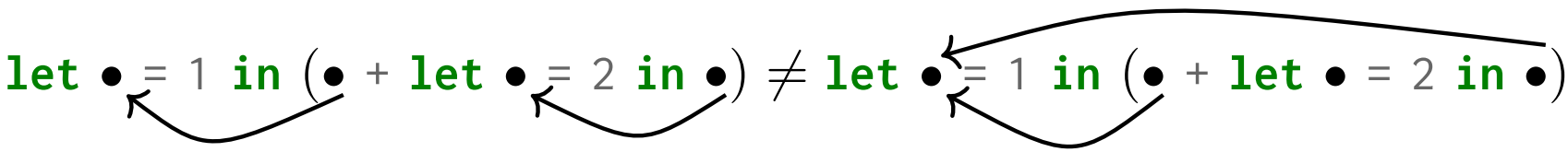
```
let x = 1 in (x + let x = 2 in x) ≡ let x = 1 in (x + let y = 2 in y)
                                  ≢ let x = 1 in (x + let y = 2 in x)
```

L'égalité à renommage des occurrences liées près

Une occurence liée n'a pas d'identité : elle ne fait que référence à un lieur.

```
let x = 1 in (x + let x = 2 in x) ≡ let x = 1 in (x + let y = 2 in y)
                                  ≢ let x = 1 in (x + let y = 2 in x)
```

On doit donc voir les expressions comme des graphes de syntaxe abstraite.

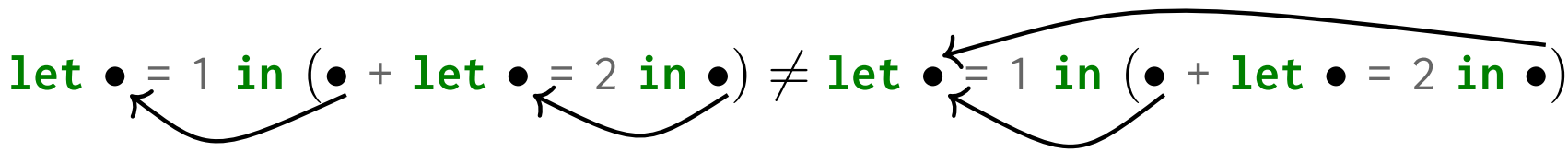


L'égalité à renommage des occurrences liées près

Une occurence liée n'a pas d'identité : elle ne fait que référence à un lieur.

$$\texttt{let x = 1 in (x + let x = 2 in x)} \equiv \texttt{let x = 1 in (x + let y = 2 in y)}$$
$$\not\equiv \texttt{let x = 1 in (x + let y = 2 in x)}$$

On doit donc voir les expressions comme des graphes de syntaxe abstraite.

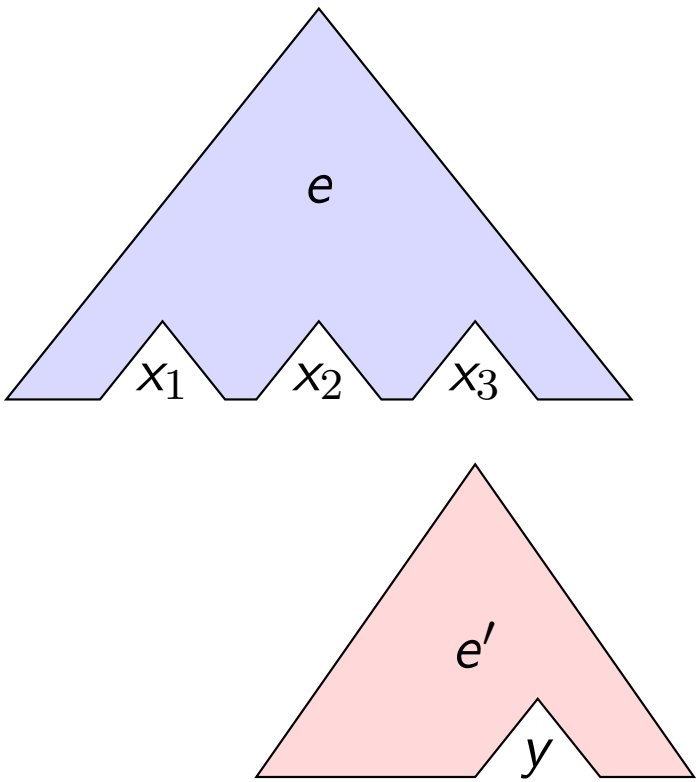


En pratique

- Les implémentations qui doivent traiter sérieusement les occurrences libres utilisent des variantes de graphes de syntaxe abstraite.
- On peut se contenter des arbres de syntaxe abstraite, mais il faut alors s'assurer de n'écrire que des opérations compatibles avec les graphes.

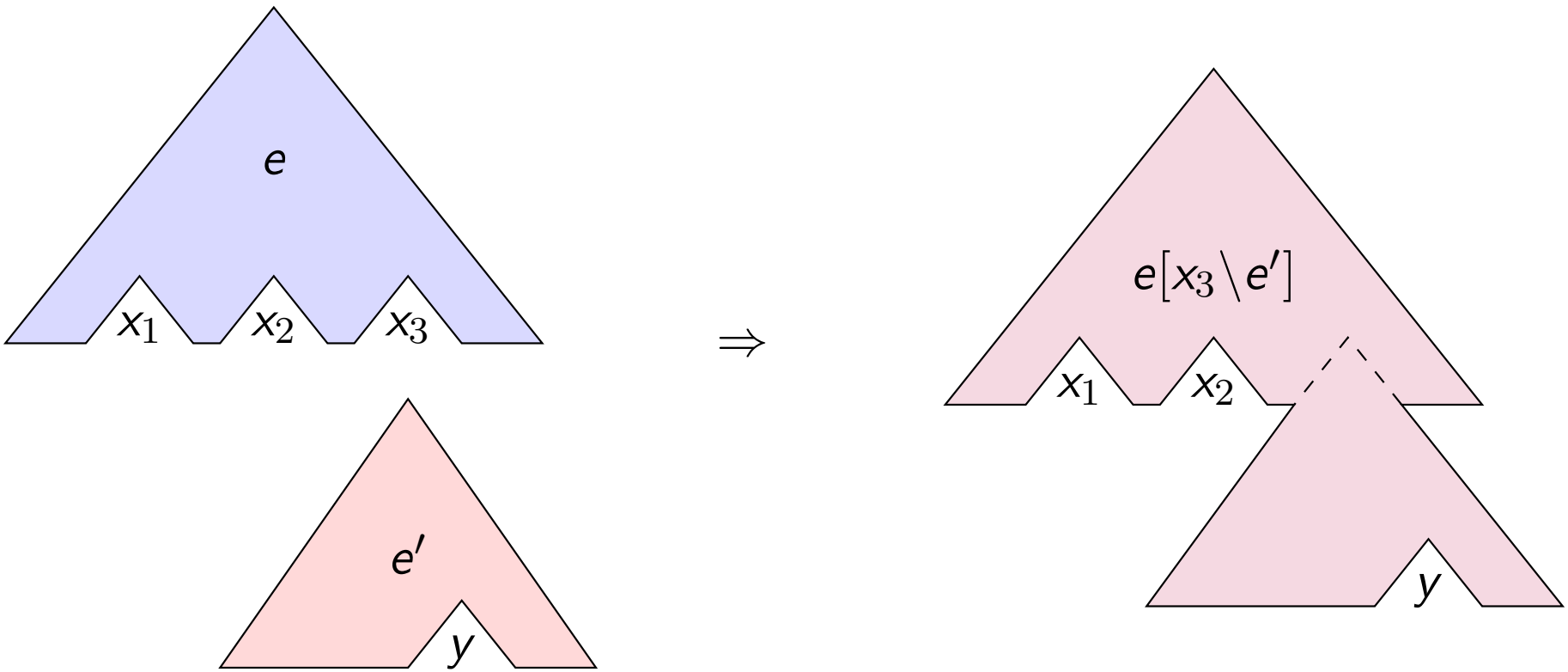
La substitution, visuellement

On note $e[x \backslash e']$ la substitution des occurrences libres de x par e' dans e.



La substitution, visuellement

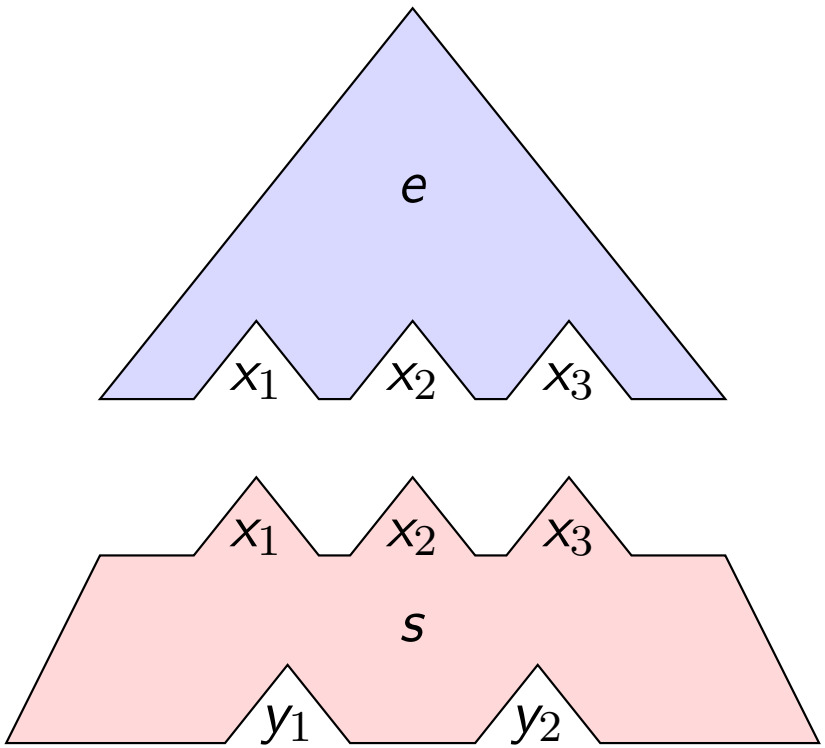
On note $e[x \backslash e']$ la substitution des occurrences libres de x par e' dans e.



La substitution parallèle, visuellement

On généralise pour substituer simultanément plusieurs variables.

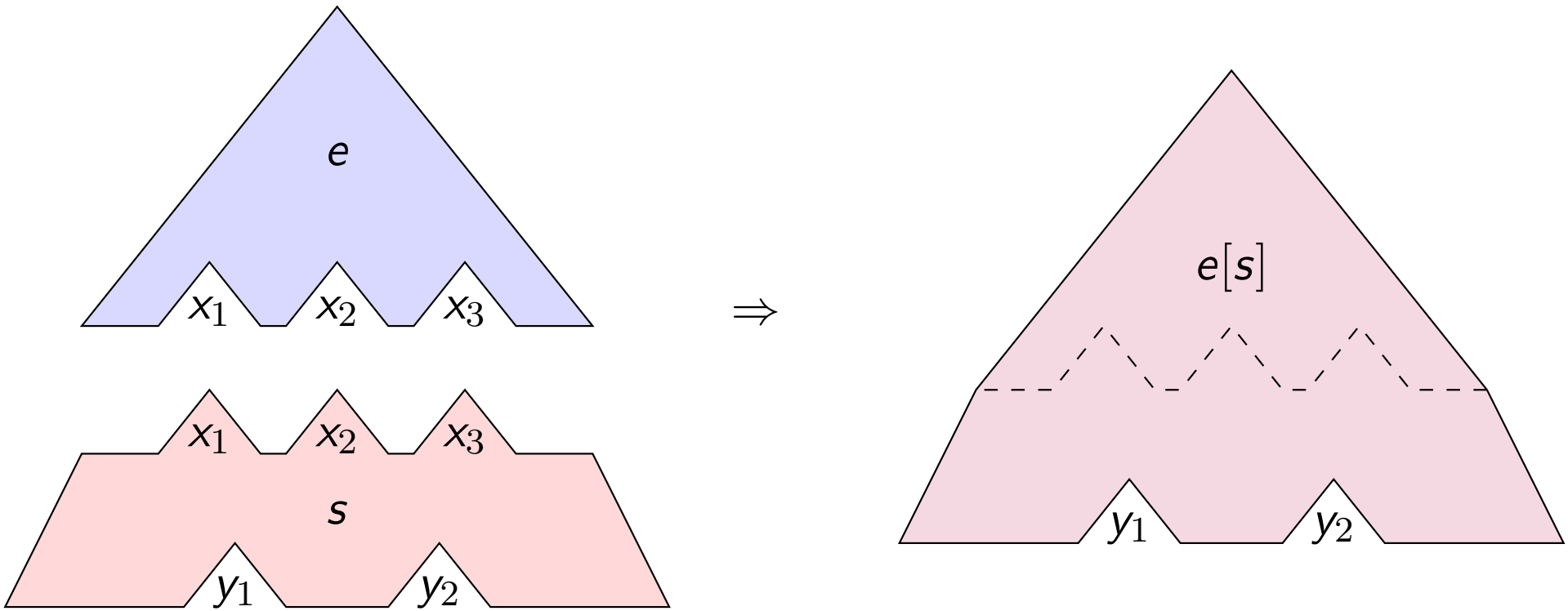
```
type subst = (Id.t * t) list
let s = [(x1, e1); (x2, e2); (x3, e3)]
```



La substitution parallèle, visuellement

On généralise pour substituer simultanément plusieurs variables.

```
type subst = (Id.t * t) list
let s = [(x1, e1); (x2, e2); (x3, e3)]
```



La substitution parallèle pour $\mathrm{L0}$

```
let rec subst : t -> subst -> t =
  fun e s ->
  match e with
  | Var x -> (try List.assoc x s with Not_found -> e)
  | Int _ -> e
  | Add (e1, e2) -> Add (subst e1 s, subst e2 s)
  | Let (e, b) -> Let (subst e s, subst_b1 b s)
and subst_b1 : bound1 -> subst -> bound1 =
  fun { bound = x; body; } s ->
    let x' = Var (Id.fresh ()) in
    { bound = x'; body = subst body ((x, x') :: s); }
```

La substitution parallèle pour L0

```
let rec subst : t -> subst -> t =
  fun e s ->
  match e with
  | Var x -> (try List.assoc x s with Not_found -> e)
  | Int _ -> e
  | Add (e1, e2) -> Add (subst e1 s, subst e2 s)
  | Let (e, b) -> Let (subst e s, subst_b1 b s)
and subst_b1 : bound1 -> subst -> bound1 =
  fun { bound = x; body; } s ->
    let x' = Var (Id.fresh ()) in
    { bound = x'; body = subst body ((x, x') :: s); }
```

Deux écueils évités par le renommage systématique des variables liées

- La substitution erronnée d'une occurrence liée dans e.
  - (`List.assoc` assure un fonctionnement *last in, first out*.)
- La capture les variables libres de s en passant sous un lieur de e.

L'α-équivalence pour L0

```
let close_with : bound1 -> t -> t =
  fun { bound; body; } e -> subst body [(bound, e)]

let rec equal e e' =
  match e, e' with
  | Var x, Var x' -> x = x'
  | Int i, Int i' -> i = i'
  | Add (e1, e2), Add (e1', e2') -> equal e1 e1' && equal e2 e2'
  | Let (e, b), Let (e', b') -> equal e e' && equal_b1 b b'
  | _ -> false

and equal_b1 b b' =
  let y = Var (Id.fresh ()) in
  equal (close_with b y) (close_with b' y)
```

Plan

1 Introduction

2 Équivalence syntaxique et substitution

3 Évaluation
- Évaluation naïve
- Environnements et fermetures
- Métacircularité
- Évaluation des références
- Évaluation du filtrage de motifs

4 Conclusion

Plan

1. Introduction
2. Équivalence syntaxique et substitution
3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs
4. Conclusion

Vers des notions d'équivalence sémantiques

Une première intuition

- Deux programmes sont équivalents quand "ils ont le même résultat".
- Il faut donc définir le résultat d'un programme, i.e., de son évaluation.

Vers des notions d'équivalence sémantiques

Une première intuition

- Deux programmes sont équivalents quand "ils ont le même résultat".
- Il faut donc définir le résultat d'un programme, i.e., de son évaluation.

Il y a de nombreuses façons de définir ce résultat :

- dans un style *à grands pas*, par une fonction partielle d'évaluation ;
- dans un style *à petit pas*, par une relation de réduction ;
- ...

Chacune de ces approches admet de nombreuses variations.

Vers des notions d'équivalence sémantiques

Une première intuition

- Deux programmes sont équivalents quand "ils ont le même résultat".
- Il faut donc définir le résultat d'un programme, i.e., de son évaluation.

Il y a de nombreuses façons de définir ce résultat :

- dans un style *à grands pas*, par une fonction partielle d'évaluation ;
- dans un style *à petit pas*, par une relation de réduction ;
- ...

Chacune de ces approches admet de nombreuses variations.

On va voir les deux approches en OCaml, pour aboutir à un interprète.

Un évaluateur pour $\mathrm{L0}$

Il est facile d'évaluer les expressions closes (sans variables libres) de $\mathrm{L0}$.

```
type value = int

let rec eval : t -> value = function
| Var _ -> invalid_arg "open expression"
| Int n -> n
| Add (e1, e2) -> eval e1 + eval e2
| Let (e, b) -> eval (close_with b e)
```

Un évaluateur pour L0

Il est facile d'évaluer les expressions closes (sans variables libres) de L0.

```
type value = int

let rec eval : t -> value = function
| Var _ -> invalid_arg "open expression"
| Int n -> n
| Add (e1, e2) -> eval e1 + eval e2
| Let (e, b) -> eval (close_with b e)
```

Voyez-vous une alternative pour l'évaluation des définitions locales ?

Un évaluateur pour L0

Il est facile d'évaluer les expressions closes (sans variables libres) de L0.

```
type value = int

let rec eval : t -> value = function
| Var _ -> invalid_arg "open expression"
| Int n -> n
| Add (e1, e2) -> eval e1 + eval e2
| Let (e, b) -> eval (close_with b e)
```

Voyez-vous une alternative pour l'évaluation des définitions locales ?

```
let rec eval : t -> value = function
| Var _ -> invalid_arg "open expression"
| Int n -> n
| Add (e1, e2) -> eval e1 + eval e2
| Let (e, b) -> eval (close_with b (Int (eval e)))
```

Le second choix est majoritaire dans les langages de programmation.

Le langage $\mathrm{L1}$

On étend la syntaxe de $\mathrm{L0}$ avec des fonctions de première classe.

```
type t = Var of Id.t              (* occurrence "x", "y", etc. *)
       | Int of int               (* entier litéral "42", etc. *)
       | Add of t * t             (* somme "e1 + e2" *)
       | Let of t * bound1        (* déf. locale "let e1 be x in e2" *)
       | Fun of bound1            (* fonction anonyme "fun x -> e" *)
       | App of t * t             (* application "e1 e2" *)
and bound1 = { bound : Id.t; body : bound1; }
```

La substitution et l'α-conversion s'étendent de façon routinière.

Un évaluateur pour L1

```
type value = VInt of int | VFun of bound1

let rec eval : t -> value = function
| Var _ -> invalid_arg "open expression"
| Int n -> VInt n
| Add (e1, e2) -> add_values (eval e1) (eval e2)
| Let (e, b) -> eval (close_with b (reify (eval e)))
| Fun b -> VFun b
| App (e1, e2) -> app_values (eval e1) (eval e2)

and add_values v1 v2 = match v1, v2 with
| VInt n1, VInt n2 -> VInt (n1 + n2)
| _ -> failwith "ill-typed"

and app_values v1 v2 = match v1 with
| VFun b -> eval (close_with b (reify v2))
| _ -> failwith "ill-typed"

and reify : value -> expr = function VInt n -> Int n | VFun b -> Fun b
```

Exercice : ajouter les booléens à $\mathrm{L1}$

```
type t = Var of Id.t              (* occurrence "x", "y", etc. *)
       | Int of int               (* entier litéral "42", etc. *)
       | Add of t * t             (* somme "e1 + e2" *)
         ...
       | Bool of bool             (* booléen litéral "true", "false" *)
       | If of t * t * t          (* conditionnelle *)
```

Exercice : ajouter les booléens à L1

```
type t = Var of Id.t              (* occurrence "x", "y", etc. *)
       | Int of int               (* entier litéral "42", etc. *)
       | Add of t * t             (* somme "e1 + e2" *)
         ...
       | Bool of bool             (* booléen litéral "true", "false" *)
       | If of t * t * t          (* conditionnelle *)

type value = VInt of int | VFun of bound1 | VBool of bool
```

Exercice : ajouter les booléens à L1

```
type t = Var of Id.t              (* occurrence "x", "y", etc. *)
       | Int of int               (* entier litéral "42", etc. *)
       | Add of t * t             (* somme "e1 + e2" *)
         ...
       | Bool of bool             (* booléen litéral "true", "false" *)
       | If of t * t * t          (* conditionnelle *)

type value = VInt of int | VFun of bound1 | VBool of bool

let rec eval : t -> value = function
| Var _ -> invalid_arg "open expression"
| Int n -> VInt n
| Add (e1, e2) -> add_values (eval e1) (eval e2)
  ...
| Bool b -> VBool b
| If (e1, e2, e3) -> if_values (eval e1) e2 e3
and if_values v e2 e3 = match v with
| VBool true -> eval e1 | VBool false -> eval e2
| _ -> failwith "ill-typed"
```

Interlude

Remarque de fond : OCaml $\neq$ Mathématiques

- Écrire l'évaluateur en OCaml est agréable, le code est très concis.
- En contrepartie, comprendre la sémantique de $\mathrm{L0}$ telle que définie par l'évaluateur exige de comprendre la sémantique d'OCaml.
- Un cours de sémantique utilisera plutôt le langage mathématique.
  - Différent d'OCaml, p. ex., pas de fonctions récursives arbitraires !

Interlude

Remarque de fond : OCaml $\neq$ Mathématiques

- Écrire l'évaluateur en OCaml est agréable, le code est très concis.
- En contrepartie, comprendre la sémantique de $\mathrm{L0}$ telle que définie par l'évaluateur exige de comprendre la sémantique d'OCaml.
- Un cours de sémantique utilisera plutôt le langage mathématique.
  - Différent d'OCaml, p. ex., pas de fonctions récursives arbitraires !

```
let b : bool = let rec f () = not (f ()) in f ()
```

Est-ce que `b` est le booléen `true` ou le booléen `false` ? Est-ce que les expressions OCaml de type `bool` sont en bijection avec $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$?

Interlude

Remarque de fond : OCaml $\neq$ Mathématiques

- Écrire l'évaluateur en OCaml est agréable, le code est très concis.
- En contrepartie, comprendre la sémantique de $\mathrm{L0}$ telle que définie par l'évaluateur exige de comprendre la sémantique d'OCaml.
- Un cours de sémantique utilisera plutôt le langage mathématique.
  - Différent d'OCaml, p. ex., pas de fonctions récursives arbitraires !

```
let b : bool = let rec f () = not (f ()) in f ()
```

Est-ce que `b` est le booléen `true` ou le booléen `false` ? Est-ce que les expressions OCaml de type `bool` sont en bijection avec $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$?

Pour les curieux et les curieuses

Sémantique des langages de programmation au S2 traite de ces questions.

Plan

1. Introduction
2. Équivalence syntaxique et substitution
3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs
4. Conclusion

Des substitutions aux environnements

```
let close_with : bound1 -> t -> t =
  fun { bound; body; } e -> subst body [(bound, e)]
...
and app_values v1 v2 = match v1 with
| VFun b -> eval (close_with b (reify v2))
| _ -> failwith "ill-typed"
```

L'appel à `subst` dans `close_with` est coûteux : on parcourt tout `b.body`, même les sous-expressions dont l'évaluation ne sera pas nécessaire.

Des substitutions aux environnements

```
let close_with : bound1 -> t -> t =
  fun { bound; body; } e -> subst body [(bound, e)]
...
and app_values v1 v2 = match v1 with
| VFun b -> eval (close_with b (reify v2))
| _ -> failwith "ill-typed"
```

L'appel à `subst` dans `close_with` est coûteux : on parcourt tout `b.body`, même les sous-expressions dont l'évaluation ne sera pas nécessaire.

Une alternative à la substitution : les environnements

- L'évaluateur reçoit, en plus de l'expression, un *environnement*.
- C'est une "substitution de valeurs", i.e., `(Id.t * value) list`.
- Il détermine la valeur des variables libres de l'expression évaluée.
- On l'étend lors du passage sous un lieur (`let`, `fun`).

Un évaluateur avec environnement pour L1

```
type env = (Id.t * value) list
type value = VInt of int | VFun of bound1 | VBool of bool
let rec eval' : env -> t -> value = fun env e -> match e with
| Var x -> List.assoc x env
| Int n -> VInt n
| Add (e1, e2) -> add_values' (eval' env e1) (eval' env e2)
| Let (e, b) -> eval'_bound1 env b (eval' env e)
| Fun b -> VFun b
| App (e1, e2) -> app_values' env (eval' env e1) (eval' env e2)
| Bool b -> VBool b
| If (e1, e2, e3) -> if_values' env (eval' env e1) e2 e3

...

and app_values' env v1 v2 = match v1 with
| VFun b -> eval'_bound1 env b v2
| _ -> failwith "ill-typed"

and eval'_bound1 env { bound; body; } v =
  eval' ((bound, v) :: env) body
```

Un évaluateur avec environnement pour L1 (**FAUX**)

```
type env = (Id.t * value) list
type value = VInt of int | VFun of bound1 | VBool of bool
let rec eval' : env -> t -> value = fun env e -> match e with
| Var x -> List.assoc x env
| Int n -> VInt n
| Add (e1, e2) -> add_values' (eval' env e1) (eval' env e2)
| Let (e, b) -> eval'_bound1 env b (eval' env e)
| Fun b -> VFun b
| App (e1, e2) -> app_values' env (eval' env e1) (eval' env e2)
| Bool b -> VBool b
| If (e1, e2, e3) -> if_values' env (eval' env e1) e2 e3

...

and app_values' env v1 v2 = match v1 with
| VFun b -> eval'_bound1 env b v2
| _ -> failwith "ill-typed"

and eval'_bound1 env { bound; body; } v =
  eval' ((bound, v) :: env) body
```

Un évaluateur avec environnement

Pourquoi la fonction `eval'` est-elle fausse ? Elle doit passer ce test :

```
let test env e =
  let subst_of_env = List.map (fun (x, v) -> (x, reify v)) in
  eval' env e = eval (subst (subst_of_env env) e)
```

Que se passe-t-il si on évalue les expressions ci-dessous ?

```
let f = let x = 1 in fun y -> x + y in f 1;;
let g = let x = 1 in fun y -> x + y in let x = 2 in g 1;;
```

Un évaluateur avec environnement

Pourquoi la fonction `eval'` est-elle fausse ? Elle doit passer ce test :

```
let test env e =
  let subst_of_env = List.map (fun (x, v) -> (x, reify v)) in
  eval' env e = eval (subst (subst_of_env env) e)
```

Que se passe-t-il si on évalue les expressions ci-dessous ?

```
let f = let x = 1 in fun y -> x + y in f 1;;
let g = let x = 1 in fun y -> x + y in let x = 2 in g 1;;
```

Environnements et portée lexicale

- La gestion de l'environnement est fausse pour les fonctions.
- L'environnement pertinent est celui du point de création.
- Il faut transmettre cet environnement jusqu'au point d'application.

Un évaluateur avec environnement et fermetures

```
type env = (Id.t * value) list
and value = VInt of int | VFun of bound1 * env | VBool of bool

let rec eval' : env -> t -> value = fun env e -> match e with
  | Var x -> List.assoc x env
  | Int n -> VInt n
  | Add (e1, e2) -> add_values' (eval' env e1) (eval' env e2)
  | Let (e, b) -> eval'_bound1 env b (eval' env e)
  | Fun b -> VFun (b, env)
  | App (e1, e2) -> app_values' (eval' env e1) (eval' env e2)
  | Bool b -> VBool b
  | If (e1, e2, e3) -> if_values' env (eval' env e1) e2 e3
  ...
and app_values' v1 v2 = match v1 with
| VFun (b, env) -> eval'_bound1 env b v2
| _ -> failwith "ill-typed"

and eval'_bound1 env { bound; body; } v =
  eval' ((bound, v) :: env) body
```

Plan

1. Introduction
2. Équivalence syntaxique et substitution
3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs
4. Conclusion

Langage objet et langage sujet (“métalangage”)

La semaine dernière

Un évaluateur avec environnements gérant correctement la portée lexicale via les *fermetures*, qui groupent les expressions et leurs environnements.

Y avait-il plus simple ?

Langage objet et langage sujet (“métalangage”)

La semaine dernière

Un évaluateur avec environnements gérant correctement la portée lexicale via les *fermetures*, qui groupent les expressions et leurs environnements.

Y avait-il plus simple ?

Une solution fainéante

- On utilise les booléens d'OCaml pour interpréter les booléens de $\mathrm{L1}$.
- On utilise les entiers d'OCaml pour interpréter les entiers de $\mathrm{L1}$.
- Pourquoi ne pas faire de même pour les fonctions ?

Un évaluateur avec env., partiellement métacirculaire

```
type value = VInt of int | VFun of (value -> value) | VBool of bool
type env = (Id.t * value) list
```

Un évaluateur avec env., partiellement métacirculaire

```
type value = VInt of int | VFun of (value -> value) | VBool of bool
type env = (Id.t * value) list

let rec eval' : env -> t -> value = fun env e -> match e with
  | Var x -> List.assoc x env
  | Int n -> VInt n
  | Add (e1, e2) -> add_values' (eval' env e1) (eval' env e2)
  | Let (e, b) -> eval'_bound1 env b (eval' env e)
  | Fun b -> VFun (fun v_arg -> eval_bound1 env b v_arg)
  | App (e1, e2) -> app_values' (eval' env e1) (eval' env e2)
  | Bool b -> VBool b
  | If (e1, e2, e3) -> if_values' env (eval' env e1) e2 e3
  ...
and app_values' v1 v2 = match v1 with
| VFun f -> f v2
| _ -> failwith "ill-typed"

and eval'_bound1 env { bound; body; } v =
  eval' ((bound, v) :: env) body
```

Les évaluateurs métacirculaires

Soient L_s et L_o deux langages de programmation.

Définition (Évaluateur métacirculaire)

- Un *évaluateur métacirculaire* de L_o en L_s interprète chaque construction de L_o par la même construction dans L_s.
- Dans le cas extrême, $L_s = L_o$ et on parle d'*autoévaluateur*.

Les évaluateurs métacirculaires

Soient L_s et L_o deux langages de programmation.

Définition (Évaluateur métacirculaire)

- Un *évaluateur métacirculaire* de L_o en L_s interprète chaque construction de L_o par la même construction dans L_s.
- Dans le cas extrême, $L_s = L_o$ et on parle d'*autoévaluateur*.

- De façon générale, il existe différents degrés de métacircularité.
  - Un interprète peut reposer plus ou moins fortement sur le métalangage.
- Par exemple, nous aurions pu éviter complètement la substitution.
  - Solution sans environnements, ni fermetures, ni... variables !

Syntaxe abstraite d'ordre supérieur et métacircularité

```
type t = ...                    (* plus de constructeur Var ! *)
       | Fun of bound1          (* fonction anonyme "fun x -> e" *)
       | App of t * t           (* application "e1 e2" *)
and bound1 = t -> t             (* plus de substitution ! *)

type value = VInt of int | VFun of (value -> value) | VBool of bool
```

Syntaxe abstraite d'ordre supérieur et métacircularité

```
type t = ...                    (* plus de constructeur Var ! *)
       | Fun of bound1          (* fonction anonyme "fun x -> e" *)
       | App of t * t           (* application "e1 e2" *)
and bound1 = t -> t             (* plus de substitution ! *)

type value = VInt of int | VFun of (value -> value) | VBool of bool

let rec eval : t -> value = function
  | Int n -> VInt n
  | Add (e1, e2) -> add_values (eval e1) (eval e2)
  | Let (e, b) -> eval_bound1 b (eval e)
  | Fun b -> VFun (eval_bound1 b)
  | App (e1, e2) -> app_values (eval e1) (eval e2)
...
```

Syntaxe abstraite d'ordre supérieur et métacircularité

```
type t = ...                    (* plus de constructeur Var ! *)
       | Fun of bound1          (* fonction anonyme "fun x -> e" *)
       | App of t * t           (* application "e1 e2" *)
and bound1 = t -> t             (* plus de substitution ! *)

type value = VInt of int | VFun of (value -> value) | VBool of bool

let rec eval : t -> value = function
  | Int n -> VInt n
  | Add (e1, e2) -> add_values (eval e1) (eval e2)
  | Let (e, b) -> eval_bound1 b (eval e)
  | Fun b -> VFun (eval_bound1 b)
  | App (e1, e2) -> app_values (eval e1) (eval e2)
...

and eval_bound1 bound1 v = eval (bound1 (reify v))

and reify = function VInt n -> Int n | VBool -> Bool b
                   | VFun f -> Fun (fun x -> reify (f (eval x)))
```

Bilan au sujet des évaluateur métacirculaires

Un évaluateur métacirculaire :

- permet de réaliser des extensions de langage à faible coût (cf. Lisp) ;
  - Interprétation métacirculaire de $L_o = L_s \cup \{\text{nouvelles constructions}\}$.
- délègue une bonne partie de la sémantique de L_o à celle de L_s.
  - Y compris les aspects subtils et implicites, e.g., l'ordre d'évaluation.

Bilan au sujet des évaluateur métacirculaires

Un évaluateur métacirculaire :

- permet de réaliser des extensions de langage à faible coût (cf. Lisp) ;
  - Interprétation métacirculaire de $L_o = L_s \cup \{\text{nouvelles constructions}\}$.
- délègue une bonne partie de la sémantique de L_o à celle de L_s.
  - Y compris les aspects subtils et implicites, e.g., l'ordre d'évaluation.

Observation générale

Plus L_s et L_o sont éloignés, plus l'interprète est informatif.

Bilan au sujet des évaluateur métacirculaires

Un évaluateur métacirculaire :

- permet de réaliser des extensions de langage à faible coût (cf. Lisp) ;
  - Interprétation métacirculaire de $L_o = L_s \cup \{\text{nouvelles constructions}\}$.
- délègue une bonne partie de la sémantique de L_o à celle de L_s.
  - Y compris les aspects subtils et implicites, e.g., l'ordre d'évaluation.

Observation générale

Plus L_s et L_o sont éloignés, plus l'interprète est informatif.

Exemple

L'évaluateur avec environnements/fermetures, qui fonctionnerait dans un langage de premier ordre, *explique* la portée bien plus que le précédent.

Bilan au sujet des évaluateur métacirculaires

Un évaluateur métacirculaire :

- permet de réaliser des extensions de langage à faible coût (cf. Lisp) ;
  - Interprétation métacirculaire de $L_o = L_s \cup \{\text{nouvelles constructions}\}$.
- délègue une bonne partie de la sémantique de L_o à celle de L_s.
  - Y compris les aspects subtils et implicites, e.g., l'ordre d'évaluation.

Observation générale

Plus L_s et L_o sont éloignés, plus l'interprète est informatif.

Exemple

L'évaluateur avec environnements/fermetures, qui fonctionnerait dans un langage de premier ordre, *explique* la portée bien plus que le précédent.

Retour sur une remarque précédente

Cas intéressant : L_s est le langage mathématique, expressif et restreint.

Plan

1. Introduction
2. Équivalence syntaxique et substitution
3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs
4. Conclusion

Le langage L2

On ajoute à L1 :

- des références qui peuvent contenir des valeurs arbitraires ;
- la séquence avec une valeur triviale pour élément neutre (type **unit**).

```
type t = Var of Id.t              (* occurrence "x", "y", etc. *)
       | Int of int               (* entier litéral "42", etc. *)
       | Add of t * t             (* somme "e1 + e2" *)
       | Bool of bool             (* booléen litéral "true", "false" *)
       | If of t * t * t          (* conditionnelle *)
       | Let of t * bound1        (* déf. locale "let e1 be x in e2" *)
       | Fun of bound1            (* fonction anonyme "fun x -> e" *)
       | App of t * t             (* application "e1 e2" *)
       | Ref of t                 (* allocation "ref e" *)
       | Read of t                (* déréférencement "!e" *)
       | Asgn of t * t            (* assignation "e1 := e2" *)
       | Unit                     (* 0-uplet *)
       | Seq of t * t             (* séquence "e1; e2" *)
and bound1 = { bound : Id.t; body : bound1; }
```

Un évaluateur métacirculaire pour L2

```
type env = (Id.t * value) list
and value = VInt of int | VBool of bool | VUnit
          | VFun of bound1 * env | VRef of value ref

let rec eval : env -> t -> value = fun env e -> match e with
  | Add (e1, e2) -> add_values (eval env e1) (eval env e2)
  | Ref e -> VRef (ref (eval env e))
  | Read e -> read_value (eval env e)
  | Asgn (e1, e2) -> asgn_values (eval env e1) (eval env e2)
  | Unit -> VUnit
  | Seq (e1, e2) -> seq_value (eval env e1) env e2
  ...

and read_value = function VRef r -> !r | _ -> failwith "ill-typed"

and asgn_values v1 v2 =
  match v1 with VRef r -> r := v2; VUnit | _ -> failwith "ill-typed"

and seq_value v1 env e2 =
  match v1 with VUnit -> eval env e2 | _ -> failwith "ill-typed"
```

Références et métacircularité

```
let e =
  let x = Var "x" in
  Let (Ref (Int 0), { bound = "x";
        body = Add (Seq (Asgn (x, Int 1), Read x), Read x); })
```

Vers quel entier s'évalue l'expression `e` ?

Références et métacircularité

```
let e =
  let x = Var "x" in
  Let (Ref (Int 0), { bound = "x";
          body = Add (Seq (Asgn (x, Int 1), Read x), Read x); })
```

Vers quel entier s'évalue l'expression `e` ?

```
utop # open Lang2;;
utop # let e = let x = Var "x" in Let (Ref (Int 0), ...);;
val e : t = let(ref 0, x.(((x := 1); !x) + !x))
utop # eval [] e;;
- : value = 1
```

Qu'est-ce qui explique ce résultat dans la définition de `eval` ?

Références et métacircularité

```
let e =
  let x = Var "x" in
  Let (Ref (Int 0), { bound = "x";
          body = Add (Seq (Asgn (x, Int 1), Read x), Read x); })
```

Vers quel entier s'évalue l'expression `e` ?

```
utop # open Lang2;;
utop # let e = let x = Var "x" in Let (Ref (Int 0), ...);;
val e : t = let(ref 0, x.(((x := 1); !x) + !x))
utop # eval [] e;;
- : value = 1
```

Qu'est-ce qui explique ce résultat dans la définition de `eval` ?

- Il dépend de l'ordre d'évaluation des arguments de fonction en OCaml.

Références et métacircularité

```
let e =
  let x = Var "x" in
  Let (Ref (Int 0), { bound = "x";
        body = Add (Seq (Asgn (x, Int 1), Read x), Read x); })
```

Vers quel entier s'évalue l'expression `e` ?

```
utop # open Lang2;;
utop # let e = let x = Var "x" in Let (Ref (Int 0), ...);;
val e : t = let(ref 0, x.(((x := 1); !x) + !x))
utop # eval [] e;;
- : value = 1
```

Qu'est-ce qui explique ce résultat dans la définition de `eval` ?

- Il dépend de l'ordre d'évaluation des arguments de fonction en OCaml.
- Formellement, celui-ci n'est pas défini. Donc celui de $\mathrm{L}2$ non plus !

Références et métacircularité

```
let e =
  let x = Var "x" in
  Let (Ref (Int 0), { bound = "x";
          body = Add (Seq (Asgn (x, Int 1), Read x), Read x); })
```

Vers quel entier s'évalue l'expression `e` ?

```
utop # open Lang2;;
utop # let e = let x = Var "x" in Let (Ref (Int 0), ...);;
val e : t = let(ref 0, x.(((x := 1); !x) + !x))
utop # eval [] e;;
- : value = 1
```

Qu'est-ce qui explique ce résultat dans la définition de `eval` ?

- Il dépend de l'ordre d'évaluation des arguments de fonction en OCaml.
- Formellement, celui-ci n'est pas défini. Donc celui de $\mathrm{L}2$ non plus !
  - Cf. section *Basic expressions* (7.2) du manuel d'OCaml.

Références et métacircularité

```
let e =
  let x = Var "x" in
  Let (Ref (Int 0), { bound = "x";
          body = Add (Seq (Asgn (x, Int 1), Read x), Read x); })
```

Vers quel entier s'évalue l'expression `e` ?

```
utop # open Lang2;;
utop # let e = let x = Var "x" in Let (Ref (Int 0), ...);;
val e : t = let(ref 0, x.(((x := 1); !x) + !x))
utop # eval [] e;;
- : value = 1
```

Qu'est-ce qui explique ce résultat dans la définition de `eval` ?

- Il dépend de l'ordre d'évaluation des arguments de fonction en OCaml.
- Formellement, celui-ci n'est pas défini. Donc celui de $\mathrm{L}2$ non plus !
  - Cf. section *Basic expressions* (7.2) du manuel d'OCaml.
- En pratique, OCaml évalue les arguments de droite à gauche.

Références et métacircularité

```
let e =
  let x = Var "x" in
  Let (Ref (Int 0), { bound = "x";
          body = Add (Seq (Asgn (x, Int 1), Read x), Read x); })
```

Vers quel entier s'évalue l'expression `e` ?

```
utop # open Lang2;;
utop # let e = let x = Var "x" in Let (Ref (Int 0), ...);;
val e : t = let(ref 0, x.(((x := 1); !x) + !x))
utop # eval [] e;;
- : value = 1
```

Qu'est-ce qui explique ce résultat dans la définition de `eval` ?

- Il dépend de l'ordre d'évaluation des arguments de fonction en OCaml.
- Formellement, celui-ci n'est pas défini. Donc celui de $\mathrm{L}2$ non plus !
  - Cf. section *Basic expressions* (7.2) du manuel d'OCaml.
- En pratique, OCaml évalue les arguments de droite à gauche.
  - Cf. le rapport de recherche *The ZINC Experiment* de X. Leroy.

Et en OCaml ?

- L'évaluateur métacirculaire n'explique aucunement les références.
- Pour les comprendre, il est utile de réfléchir à la sémantique d'OCaml.

Donner les valeurs des expressions suivantes, placées après `let x = ref 0`.

```
let x = 1 in (let x = 2 in x) + x ≡
x := 1; (x := 2; !x) + !x ≡
x := 1; !x + (x := 2; !x) ≡
let y = x in y := 1; !x + !y ≡
```

Et en OCaml ?

- L'évaluateur métacirculaire n'explique aucunement les références.
- Pour les comprendre, il est utile de réfléchir à la sémantique d'OCaml.

Donner les valeurs des expressions suivantes, placées après `let x = ref 0`.

```
let x = 1 in (let x = 2 in x) + x ≡ 3
x := 1; (x := 2; !x) + !x ≡
x := 1; !x + (x := 2; !x) ≡
let y = x in y := 1; !x + !y ≡
```

Et en OCaml ?

- L'évaluateur métacirculaire n'explique aucunement les références.
- Pour les comprendre, il est utile de réfléchir à la sémantique d'OCaml.

Donner les valeurs des expressions suivantes, placées après `let x = ref 0`.

```
let x = 1 in (let x = 2 in x) + x ≡ 3
x := 1; (x := 2; !x) + !x ≡ 3
x := 1; !x + (x := 2; !x) ≡
let y = x in y := 1; !x + !y ≡
```

Et en OCaml ?

- L'évaluateur métacirculaire n'explique aucunement les références.
- Pour les comprendre, il est utile de réfléchir à la sémantique d'OCaml.

Donner les valeurs des expressions suivantes, placées après `let x = ref 0`.

```
let x = 1 in (let x = 2 in x) + x ≡ 3
x := 1; (x := 2; !x) + !x ≡ 3
x := 1; !x + (x := 2; !x) ≡ 4
let y = x in y := 1; !x + !y ≡
```

Et en OCaml ?

- L'évaluateur métacirculaire n'explique aucunement les références.
- Pour les comprendre, il est utile de réfléchir à la sémantique d'OCaml.

Donner les valeurs des expressions suivantes, placées après `let x = ref 0`.

```
let x = 1 in (let x = 2 in x) + x ≡ 3
x := 1; (x := 2; !x) + !x ≡ 3
x := 1; !x + (x := 2; !x) ≡ 4
let y = x in y := 1; !x + !y ≡ 2
```

Et en OCaml ?

- L'évaluateur métacirculaire n'explique aucunement les références.
- Pour les comprendre, il est utile de réfléchir à la sémantique d'OCaml.

Donner les valeurs des expressions suivantes, placées après `let x = ref 0`.

```
let x = 1 in (let x = 2 in x) + x ≡ 3
x := 1; (x := 2; !x) + !x ≡ 3
x := 1; !x + (x := 2; !x) ≡ 4
let y = x in y := 1; !x + !y ≡ 2
```

Observations immédiates

- L'assignation n'est pas locale ; elle ne repose pas sur la substitution.
- L'évaluation des deux dernières expressions n'est pas une pure valeur.
  - Elle modifie le contenu de x pour la suite du programme.
- Ces expressions "impures" sont sensibles à l'ordre d'évaluation.

Gestion de la mémoire

Les références manipulent des adresses abstraites dans une mémoire.

Gestion de la mémoire

Les références manipulent des adresses abstraites dans une mémoire.

```
module Address : sig
  type t                          (* un type de données abstrait... *)
  val compare : t -> t -> int (* ... inextinguible ... *)
  val fresh : unit -> t           (* ... et totalement ordonné. *)
end
```

Gestion de la mémoire

Les références manipulent des adresses abstraites dans une mémoire.

```
module Address : sig
  type t                          (* un type de données abstrait... *)
  val compare : t -> t -> int (* ... inextinguible ... *)
  val fresh : unit -> t           (* ... et totalement ordonné. *)
end

module Memory = struct
  include Map.Make(Address)
  let alloc v0 m = let a = Address.fresh () in a, add a v0 m
  let read a m = find a m
  let asgn a v m = if mem a m then add a v mem else raise Not_found
end
```

Gestion de la mémoire

Les références manipulent des adresses abstraites dans une mémoire.

```
module Address : sig
  type t                          (* un type de données abstrait... *)
  val compare : t -> t -> int (* ... inextinguible ... *)
  val fresh : unit -> t           (* ... et totalement ordonné. *)
end

module Memory = struct
  include Map.Make(Address)
  let alloc v0 m = let a = Address.fresh () in a, add a v0 m
  let read a m = find a m
  let asgn a v m = if mem a m then add a v mem else raise Not_found
end
```

Remarque : un modèle abstrait de la mémoire

- Ici la mémoire n'est pas un tableau, ni les adresses des indices.
- En particulier, l'arithmétique de pointeur n'est pas possible.

Évaluation des références (1/2)

```
type env = (Id.t * value) list and value = VInt of int
                                         | VBool of bool
                                         | VFun of bound1 * env
                                         | VRef of Address.t

type mem = value Memory.t
```

Quel serait le type d'un évaluateur écrit en OCaml purement fonctionnel ?

Évaluation des références (1/2)

```
type env = (Id.t * value) list and value = VInt of int
                                         | VBool of bool
                                         | VFun of bound1 * env
                                         | VRef of Address.t

type mem = value Memory.t
```

Quel serait le type d'un évaluateur écrit en OCaml purement fonctionnel ?

```
val eval : env -> t -> mem -> value * mem
```

Évaluation des références (1/2)

```
type env = (Id.t * value) list and value = VInt of int
                                         | VBool of bool
                                         | VFun of bound1 * env
                                         | VRef of Address.t

type mem = value Memory.t
```

Quel serait le type d'un évaluateur écrit en OCaml purement fonctionnel ?

```
val eval : env -> t -> mem -> value * mem
```

N.B. : ne suppose pas l'existence de références dans le métalangage !

Évaluation des références (2/2)

```
let rec eval env e m =
  match e with
   | Var x -> List.assoc x env, m
   | Add (e1, e2) -> let v1, m = eval env e1 m in
                     let v2, m = eval env e2 m in
                     add_values v1 v2, m
   | Fun b -> VFun (b, env), m
     ...
   | Ref e -> let v, m = eval env e m in
              let a, m = Memory.alloc v m in VRef a, m
   | Read e -> let v, m = eval env e m in read_value v m
   | Asgn (e1, e2) -> let v1, m = eval env e1 m in
                      let v2, m = eval env e2 m in
                      asgn_values v1 v2 m

and read_value v m = match v with
   | VRef a -> Memory.read a m, m | _ -> failwith "ill-typed"

and asgn_values v1 v2 m = match v1 with
   | VRef a -> VUnit, Memory.asgn a v2 m | _ -> failwith "ill-typed"
```

Programmer mieux pour programmer moins

```
let rec eval : env -> t -> mem -> value * mem = fun env e m ->
  match e with
  | Var x -> List.assoc x env, m
  | Add (e1, e2) -> let v1, m = eval env e1 m in
                    let v2, m = eval env e2 m in
                    add_values v1 v2, m
  | Fun b -> VFun (b, env), m
  ...
```

De quelques effets désagréables du changement de type de `eval`

- Nous avons dû modifier l'interprétation de chaque construction.
- Les cas préexistants sont pollués par de la bureaucratie.

Programmer mieux pour programmer moins

```
let rec eval : env -> t -> mem -> value * mem = fun env e m ->
  match e with
  | Var x -> List.assoc x env, m
  | Add (e1, e2) -> let v1, m = eval env e1 m in
                    let v2, m = eval env e2 m in
                    add_values v1 v2, m
  | Fun b -> VFun (b, env), m
  ...
```

De quelques effets désagréables du changement de type de `eval`

- Nous avons dû modifier l'interprétation de chaque construction.
- Les cas préexistants sont pollués par de la bureaucratie.

Opérations logistiques récurrentes

1. Renvoyer une valeur et la mémoire non-modifiée.
2. Chaîner les transformations de la mémoire.

L'abstraction *monadique*

```
type 'a stateful = mem -> 'a * mem
val return : 'a -> 'a stateful
val ( >>= ) : 'a stateful -> ('a -> 'b stateful) -> 'b stateful
```

L'abstraction *monadique*

```
type 'a stateful = mem -> 'a * mem
val return : 'a -> 'a stateful
val ( >>= ) : 'a stateful -> ('a -> 'b stateful) -> 'b stateful

let rec eval : env -> t -> value stateful = fun env e ->
  match e with
  | Var x -> return (List.assoc x env)
  | Add (e1, e2) -> eval env e1 >>= fun v1 ->
                    eval env e2 >>= fun v2 ->
                    return (add_values v1 v2)
  | Fun b -> return (VFun (b, env))
  ...
```

L'abstraction *monadique*

```
type 'a stateful = mem -> 'a * mem
val return : 'a -> 'a stateful
val ( >>= ) : 'a stateful -> ('a -> 'b stateful) -> 'b stateful

let rec eval : env -> t -> value stateful = fun env e ->
  match e with
  | Var x -> return (List.assoc x env)
  | Add (e1, e2) -> eval env e1 >>= fun v1 ->
                    eval env e2 >>= fun v2 ->
                    return (add_values v1 v2)
  | Fun b -> return (VFun (b, env))
  ...

let return x = fun m -> x, m
let ( >>= ) xm f = fun m -> let x, m = xm m in f x m
```

Les opérateurs de liaison définis par l'utilisateur

En OCaml 4.08+ (2019)

La notation "`let+ x = e1 in e2`" désigne "`(let+) e1 (fun x -> e2)`".

Les opérateurs de liaison définis par l'utilisateur

En OCaml 4.08+ (2019)

La notation "`let+ x = e1 in e2`" désigne "`(let+) e1 (fun x -> e2)`".

```
let ( let+ ) = ( >>= )

let rec eval : env -> t -> value stateful = fun env e ->
  match e with
  | Var x -> return (List.assoc x env)
  | Add (e1, e2) -> let+ v1 = eval env e1 in
                    let+ v2 = eval env e2 in
                    return (add_values v1 v2)
  | Fun b -> return (VFun (b, env))
  ...
```

Les opérateurs de liaison définis par l'utilisateur

En OCaml 4.08+ (2019)

La notation "`let+ x = e1 in e2`" désigne "`(let+) e1 (fun x -> e2)`".

```
let ( let+ ) = ( >>= )

let rec eval : env -> t -> value stateful = fun env e ->
  match e with
  | Var x -> return (List.assoc x env)
  | Add (e1, e2) -> let+ v1 = eval env e1 in
                    let+ v2 = eval env e2 in
                    return (add_values v1 v2)
  | Fun b -> return (VFun (b, env))
  ...
```

Bilan

- Le code est très lisible et proche du code original.
- L'opérateur `(let+)`/`(>>=)` nous force à expliciter l'ordre d'évaluation.

Plan

1. Introduction
2. Équivalence syntaxique et substitution
3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs
4. Conclusion

Le langage L3

On ajoute à L1 les *n*-uplets et le *filtrage de motifs*.

```
type t = Var of Id.t                    (* occurrence "x", "y", etc. *)
       | Int of int                     (* entier litéral "42", etc. *)
       | Add of t * t                   (* somme "e1 + e2" *)
       ...
       | Tuple of t list                (* n-uplets "(e1, ..., eN)" *)
       | Match of exp * branch list (* filtrage "match e with ..." *)
```

Le langage L3

On ajoute à L1 les *n*-uplets et le *filtrage de motifs*.

```
type t = Var of Id.t                    (* occurrence "x", "y", etc. *)
       | Int of int                     (* entier litéral "42", etc. *)
       | Add of t * t                   (* somme "e1 + e2" *)
       ...
       | Tuple of t list                (* n-uplets "(e1, ..., eN)" *)
       | Match of exp * branch list (* filtrage "match e with ..." *)

and branch = pattern * exp              (* branche "| p -> e" *)
```

Le langage L3

On ajoute à L1 les *n*-uplets et le *filtrage de motifs*.

```
type t = Var of Id.t                    (* occurrence "x", "y", etc. *)
       | Int of int                     (* entier litéral "42", etc. *)
       | Add of t * t                   (* somme "e1 + e2" *)
       ...
       | Tuple of t list                (* n-uplets "(e1, ..., eN)" *)
       | Match of exp * branch list (* filtrage "match e with ..." *)

and branch = pattern * exp              (* branche "| p -> e" *)

and pattern = PVar of Id.t              (* motif universel liant *)
            | PWildcard                 (* motif universel non liant *)
            | PInt of int               (* motif constant (entier) *)
            | PBool of bool             (* motif constant (booléen) *)
            | PTuple of pattern list (* motif n-uplet *)
```

Un évaluateur pour L3

```
type env = (Id.t * value) list
and value = VInt of int | VBool of bool
          | VFun of bound1 * env | VTuple of value list

let rec eval : env -> t -> value = fun env e -> match e with
  | Var x -> List.assoc x env
  | Int n -> VInt n
  | Add (e1, e2) -> add_values (eval env e1) (eval env e2)
  ...
  | Tuple es -> VTuple (List.map (eval env) es)
  | Match (e, bs) -> match_value (eval env e) env bs

and match_value v env bs = (* à faire pour le jalon 2 ! *)
```

Un évaluateur pour L3

```
type env = (Id.t * value) list
and value = VInt of int | VBool of bool
          | VFun of bound1 * env | VTuple of value list

let rec eval : env -> t -> value = fun env e -> match e with
  | Var x -> List.assoc x env
  | Int n -> VInt n
  | Add (e1, e2) -> add_values (eval env e1) (eval env e2)
  ...
  | Tuple es -> VTuple (List.map (eval env) es)
  | Match (e, bs) -> match_value (eval env e) env bs

and match_value v env bs = (* à faire pour le jalon 2 ! *)
```

La sémantique de `match v with p1 -> e1 | ... | pN -> eN`

- On évalue la première expression `eI` telle que `pI` accepte la valeur `v`.
- L'évaluation procède dans l'environnement étendu par `pI`.

```
val filter_value : pattern -> value -> env option
```

Plan

1. Introduction
2. Équivalence syntaxique et substitution
3. Évaluation
   - Évaluation naïve
   - Environnements et fermetures
   - Métacircularité
   - Évaluation des références
   - Évaluation du filtrage de motifs
4. Conclusion

Conclusion au sujet de l'interprétation

Les évaluateurs définitionnels

- Évaluer une expression par parcours ascendant de sa syntaxe.
- Inefficaces, mais simples : une *définition* exécutable.
  - Attention à ne pas trop reposer sur le métalangage !
- Il existe des techniques d'interprétation beaucoup plus efficaces.
  - Souvent via une machine abstraite, cf. `ocamlrun`, CPython, Marthe...

Conclusion au sujet de l'interprétation

Les évaluateurs définitionnels

- Évaluer une expression par parcours ascendant de sa syntaxe.
- Inefficaces, mais simples : une *définition* exécutable.
  - Attention à ne pas trop reposer sur le métalangage !
- Il existe des techniques d'interprétation beaucoup plus efficaces.
  - Souvent via une machine abstraite, cf. `ocamlrun`, CPython, Marthe...

Le jalon 2 : un interprète définitionnel pour Hopix

- Des types construits et des enregistrements.
- Les détails du filtrage de motifs.
- Les fonctions (mutuellement) récursives.

Conclusion au sujet de l'interprétation

Les évaluateurs définitionnels

- Évaluer une expression par parcours ascendant de sa syntaxe.
- Inefficaces, mais simples : une *définition* exécutable.
  - Attention à ne pas trop reposer sur le métalangage !
- Il existe des techniques d'interprétation beaucoup plus efficaces.
  - Souvent via une machine abstraite, cf. `ocamlrun`, CPython, Marthe...

Le jalon 2 : un interprète définitionnel pour Hopix

- Des types construits et des enregistrements.
- Les détails du filtrage de motifs.
- Les fonctions (mutuellement) récursives.

La prochaine fois : la vérification de types.