Algorithmes, Types, Preuves

Martin Strecker

INU Champollion

Année 2018/2019

Plan

Plan

Motivation et terminologie (1)

Évaluation stricte : Pour évaluer une application $f\ a_1 \ldots a_n$

1. évaluer les arguments $a_1 \ldots a_n$ pour obtenir des valeurs $v_1 \ldots v_n$
2. affecter $v_1 \ldots v_n$ aux paramètres formels de f
3. évaluer le corps de f

. . . le modèle d'exécution de la plupart des langages de programmation

Exemple :

```
# let f = fun x y z -> x * y + z ;;
val f : int -> int -> int -> int = <fun>
```

```
      f (2 + 3) (2 * 3) 12
 ⟶s   f 5 6 12
 ⟶s   5 * 6 + 12
 ⟶s   42
```

Motivation et terminologie (2)

Désavantage de l'évaluation stricte : certains calculs sont éventuellement inutiles :

```
      f 0 (42 * 42) (2 + 3)
⟶s    f 0 1764 5
⟶s    0 * 1764 + 5
⟶s    5
```

Évaluation paresseuse : évaluation d'une expression uniquement si (et quand) nécessaire

```
      f 0 (42 * 42) (2 + 3)
⟶p    0 * (42 * 42) + (2 + 3)
⟶p    2 + 3
⟶p    5
```

. . . le modèle d'exécution de quelques langages fonctionnels (Haskell, Miranda)

Motivation et terminologie (3)

Situations similaires : conditionnels ou `match` :

```
# let g = fun b x y -> if b then 2 * x else 3 * y;;
val g : bool -> int -> int -> int = <fun>
```

. . . lors de l'évaluation de `g true (3 * 3) (4 * 4)`
Désavantage potentiel (!) de l'évaluation paresseuse : Évaluations multiples.

```
# let h = fun x -> x * x + x ;;
val h : int -> int = <fun>
```

Comparer les deux stratégies sur `h (2 * 2)`

Règles d'évaluation

Format d'une règle typique :

$$\frac{N \longrightarrow N'}{M\ N \longrightarrow M\ N'}$$

Lecture : Si on peut réduire N vers N', alors on peut aussi réduire l'application $M\ N$ vers $M\ N'$

Utilisation : de bas en haut :

$$\frac{\texttt{(fun x -> x + 2)}\ \texttt{((fun y -> 3 * y) 4)}}{\longrightarrow\ \texttt{(fun x -> x + 2)}\ \texttt{12}}$$

...

parce que `((fun y -> 3 * y) 4)` $\longrightarrow$ `12`

Évaluation stricte : Valeurs (1)

On appelle une valeur toute expression qui ne peut plus être réduite *à la tête* :

- constantes : `3, true, 2.5, []`
- variables : `x, b`
- abstractions : `fun x -> e`, où `e` est une expression quelconque (même réductible)
- application d'un constructeur à des expressions qui sont toutes des valeurs ; paire de valeurs :
  `3 ::[], (3, true)`

Évaluation stricte : Valeurs (2)

Exemples : sont des valeurs :

- `Leaf 3`
- `(fun x -> x + 2)`
- `(fun x -> ((fun y -> 3 * y) x))`
  NB : `((fun y -> 3 * y) x)` est réductible !

ne sont pas de valeurs :

- `(fun x -> x + 2) 3`

Évaluation stricte : Règles (1)

Fragment fonctionnel pur :

- *Réduction de l'argument :*

$$\frac{N \longrightarrow_s N'}{M\ N \longrightarrow_s M\ N'}$$

- *Réduction de la fonction :*

$$\frac{M \longrightarrow_s M'}{M\ v \longrightarrow_s M'\ v}$$

où `v` est une valeur

- *Appel de fonction :* si `v` est une valeur :
  - Direct : `(fun x -> e1) v` $\longrightarrow_s$ `e1[x ← v]`
  - Expansion de définition : `f v` $\longrightarrow_s$ `e1[x ← v]` si `f` est définie par `(fun x -> e1)`

Évaluation stricte : Règles (2)

Exemple de réduction :

```
        (fun x -> ((fun y -> 3 * y) x))
         ((fun z -> 2 + z) 5)
⟶s      (fun x -> ((fun y -> 3 * y) x)) 7
⟶s      (fun y -> 3 * y) 7
⟶s      3 * 7 ⟶s 21
```

Non-exemple de réduction :

```
        (fun x -> ((fun y -> 3 * y) x))
         ((fun z -> 2 + z) 5)
```

(pb. : réduction sous `fun`)

```
⟶s      (fun x -> 3 * x) ((fun z -> 2 + z) 5)
```

(pb. : réduction de non-valeur)

```
⟶s      3 * ((fun z -> 2 + z) 5)
```

Évaluation stricte : Règles (3)

`if` : évaluation “semi-paresseuse” :

- *Réduction de la condition :*

$$\frac{M \longrightarrow_s M'}{\texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 \longrightarrow_s \texttt{if } M' \texttt{ then } N_1 \texttt{ else } N_2}$$

- *Sélection de la branche :*
  - $\texttt{if true then } N_1 \texttt{ else } N_2 \longrightarrow_s N_1$
  - $\texttt{if false then } N_1 \texttt{ else } N_2 \longrightarrow_s N_2$

`match... with` : Pareil

Évaluation stricte en Caml

L'ordre d'évaluation n'est pas observable en Caml *sauf*

- pour des programmes qui ne terminent pas :

```
# let rec nontermin () : int = nontermin ();;
val nontermin : unit -> int = <fun>
# (fun x -> 42) 5;;
- : int = 42
# (fun x -> 42) (nontermin ());;
Interrupted.   (* ne termine pas *)
```

- pour des programmes avec effet de bord :

```
# (fun x y -> (print_string "bar"; x + y))
      (print_string "foo"; 3)
        (print_string "baz"; 5);;
bazfoobar- : int = 8
```

Évaluation paresseuse : Règles (1)

Fragment fonctionnel pur :

- *Réduction de la fonction :* (N un terme arbitraire)

$$\frac{M \longrightarrow_s M'}{M\ N \longrightarrow_s M'\ N}$$

- *Appel de fonction :* (N un terme arbitraire)
  - Direct : `(fun x -> e1) N` $\longrightarrow_s$ `e1[x` $\leftarrow$ `N]`
  - Expansion de définition : `f N` $\longrightarrow_s$ `e1[x` $\leftarrow$ `N]` si `f` est définie par `(fun x -> e1)`
- *Réduction de l'argument :*

$$\frac{N \longrightarrow_s N'}{f\ N \longrightarrow_s f\ N'}$$

(si `f` est irréductible par $\longrightarrow_s$ et `f` n'est pas une abstraction)

Évaluation paresseuse : Règles (2)

Règles pour `if` et `match` : Comme pour la réduction stricte

Exemple de réduction :

$$
\begin{array}{ll}
 & \underline{\texttt{(fun x -> ((fun y -> 3 * y) x))}} \\
 & \quad \underline{\texttt{((fun z -> 2 + z) 5)}} \\
\longrightarrow_s & \underline{\texttt{((fun y -> 3 * y) ((fun z -> 2 + z) 5))}} \\
\longrightarrow_s & \texttt{3 * } \underline{\texttt{((fun z -> 2 + z) 5)}} \\
\longrightarrow_s & \texttt{3 * (2 + 5)} \longrightarrow_s \ldots
\end{array}
$$

Comparer la réduction de :

- `(fun x y -> 3 * x) 5 ((fun z -> z + 3) 4)`
- `(fun x -> x * x) ((fun z -> 4 + z) 38)`

par évaluation stricte / paresseuse

Résumé préliminaire

Nous avons vu :

- différentes stratégies d'évaluation d'un programme
- . . . qui ont des conséquences sur l'efficacité

Quelques remarques supplémentaires :

- Nous avons caché certaines subtilités (notamment : *renommage de variables*)
- Si les deux stratégies terminent, le résultat est le même ($\rightsquigarrow$ *confluence*), hors effets de bord
- Si l'évaluation stricte termine, alors aussi l'évaluation paresseuse
- . . . mais l'inverse n'est pas vrai. Donnez un exemple

Details dans l'introduction au lambda-calcul (niveau Master)
Maintenant : quelques applications pratiques

Structures infinies en Haskell (1)

Haskell (en honneur de H. Curry) est un langage fonctionnel paresseux.
Il existent :

- des séquences finies traditionnelles :

```
Hugs> [1 .. 4]
[1,2,3,4]
```

(syntaxe inexistante en Caml)

- des séquences infinies :

```
Hugs> [1 ..]
[1,2,3,4,5,6,7,8,9,10,11,12,.....
2008,2009,{Interrupted!}
```

Structures infinies en Haskell (2)

La fonction `take` prend *n* éléments d'une séquence.
Définition :

```
take n [] = []
take n (x:xs) =
    if n == 0 then [] else x:(take (n-1) xs)
```

Note : (:) en Haskell est (: :) en Caml
Utilisation :

```
Hugs> take 3 [1 .. 5]
[1,2,3]
Hugs> take 7 [1 ..]
[1,2,3,4,5,6,7]
```

Note :

- calculer `[1 ..]` ne termine pas
- . . . mais l'évaluation est paresseuse !

Structures infinies en Haskell (3)

Pareil : Fonctions `map`, sélection, ...
Exemple : sélection des multiples de *n* :

```
select_multiples n xs =
    [x | x <- xs, x `rem` n == 0]
```

Application :

```
Hugs> select_multiples 3 [1 .. 33]
[3,6,9,12,15,18,21,24,27,30,33]
Hugs> take 5 (select_multiples 7 [1 .. ])
[7,14,21,28,35]
```

Structures infinies en Haskell (4)

Le crible d'Ératosthène génère la séquence des nombres premiers

Principe :

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, . . . , 25, . . .
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, . . . , 25, . . .
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, . . . , 25, . . .

Implantation en Haskell :

```
sieve(p:rest) = p:sieve[r|r<-rest, r `rem` p /= 0]
primes = sieve [2..]
```

Application :

```
Hugs> take 12 primes
[2,3,5,7,11,13,17,19,23,29,31,37]
```

Structures infinies en Caml (1)

Type des séquences infinies :

```
type 'a seq =
    Nil
    | Cons of 'a * (unit -> 'a seq)
```

Ici, `unit` et le type dont le seul élément est `()`.
La définition `'a seq` est presque celle des listes traditionnelles (*hypothétique*) :

```
type 'a list =
    []
    | (::) of 'a * 'a list
```

...sauf que `(unit -> ...)` retarde l'évaluation

Structures infinies en Caml (2)

Génération d'une séquence infinie :

```
# let rec fromq k = Cons (k, fun() -> fromq(k+1));;
val fromq : int -> int seq = <fun>
```

`fromq 1` correspond à `[1..]` en Haskell, mais produit uniquement le début de la séquence :

```
# fromq 1 ;;
- : int seq = Cons (1, <fun>)
```

Structures infinies en Caml (3)

Sélection d'éléments :

```
# let rec takeq n = function
    Nil -> []
  | Cons(x, xq) ->
    if n = 0 then [] else x::(takeq (n-1) (xq()));;
  val takeq : int -> 'a seq -> 'a list = <fun>
# takeq 5 (fromq 3) ;;
- : int list = [3; 4; 5; 6; 7]
```

Tracer l'exécution de cet appel

Stratégies de recherche (1)

Des problèmes de recherche se posent dans plusieurs domaines :

- *Logistique :*
  - Trouver le chemin le plus court entre A et B
  - Trouver le chemin le plus large entre A et B (permettant un débit maximal)
- *Jeux :* Trouver des mouvements qui permettent de gagner
- *Résolution de contraintes :* par exemple : charger un avion avec un nombre de paquets
- *Preuves :* Appliquer des règles de manière à prouver une proposition

Stratégies de recherche (2)

Démarche :

- Partir d'une solution partielle
  Ex. : chemin incomplet entre *A* et *B*
- Appliquer des opérateur qui étendent la solution partielle
  Ex. : ajouter un tronçon au chemin
- Jusqu'à aboutir à une solution complète

Représentation comme arbre de recherche où

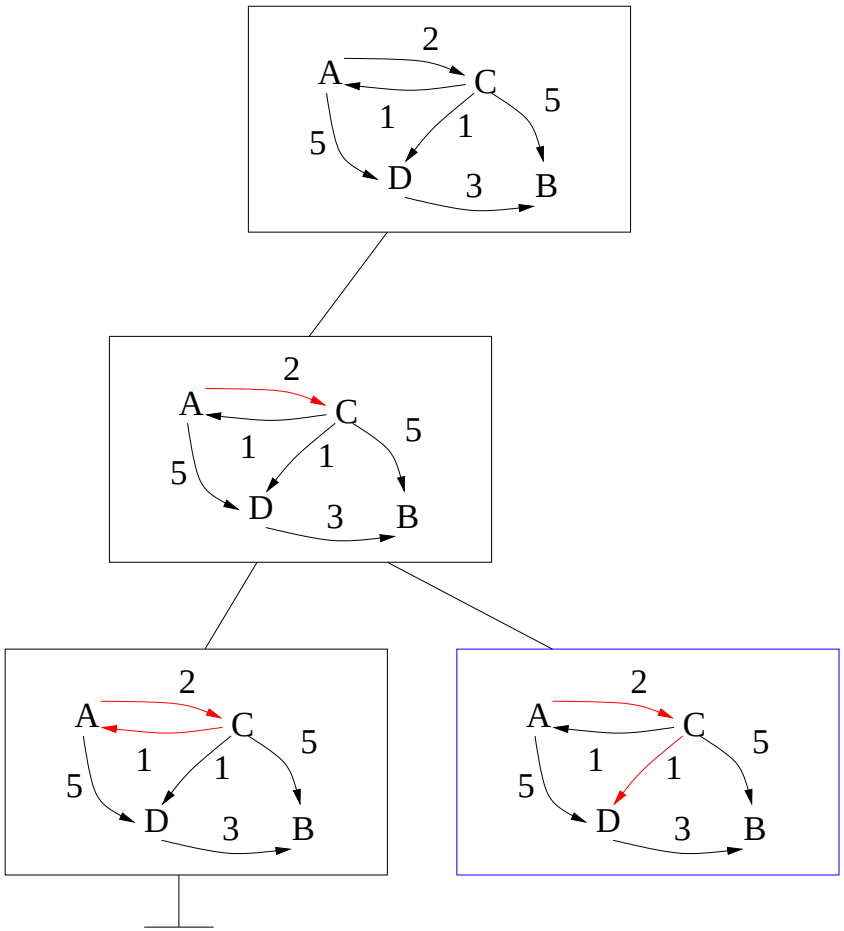
- les noeuds sont les solutions (partielles ou complètes)
- les arcs sont l'application des opérateurs

Il existe différentes manières de construire cet arbre . . .

Stratégies de recherche (3)

Recherche en profondeur (*depth first*)

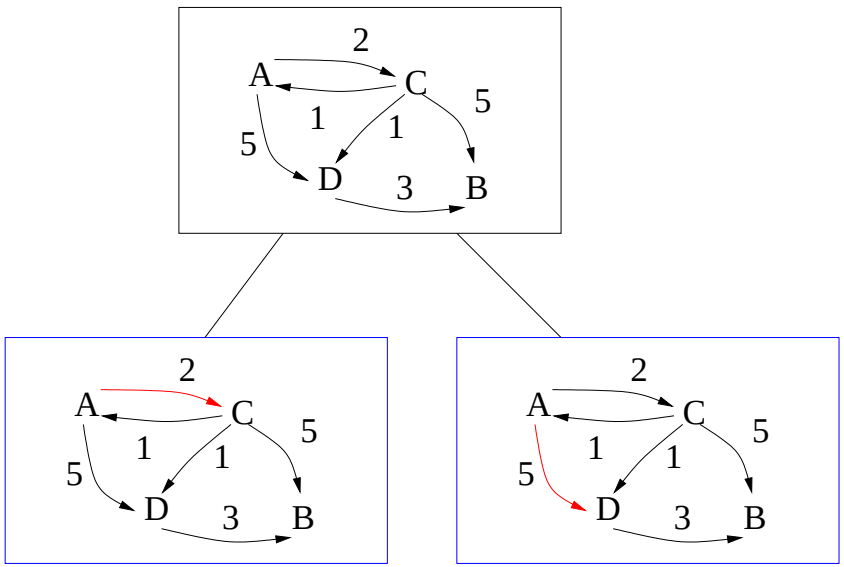
- Un noeud est actif
- Explorer récursivement les fils du noeud actif, de gauche à droite
- Arrêter la descente en cas d'échec



Stratégies de recherche (4)

Recherche en largeur (*breadth first*)

- Tous les noeuds d'un même niveau sont actifs
- Explorer récursivement tous les fils du niveau suivant
- Enlever les noeuds qui ne contribuent pas à une solution



Stratégies de recherche (5)

L'implantation de `depthfirst` et `breadthfirst` s'appuie sur

- une fonction-paramètre `next` qui calcule tous les successeurs d'un noeud
  Exemple : extension d'un chemin par un tronçon
- une fonction-paramètre `sol` qui teste si un noeud est une solution
  Exemple : vérifier que le chemin va de *A* à *B*

Stratégies de recherche (6)

Implantation recherche en profondeur

```
let depthfirst next sol x =
  let rec dfs = function
      [] -> Nil
    | y :: ys ->
        if sol y
        then Cons(y, fun () -> dfs (next y @ ys))
        else dfs (next y @ ys)
  in dfs [x]
```

`x` est la racine de l'arbre de recherche.

Stratégies de recherche (7)

Implantation recherche en largeur

```
let breadthfirst next sol x =
  let rec bfs = function
      [] -> Nil
    | y :: ys ->
        if sol y
        then Cons(y, fun () -> bfs (ys @ next y))
        else bfs (ys @ next y)
  in bfs [x]
```

Plan

Motivation et problématique (1)

But : Faire des preuves équationnelles

Exemple : Étant donné un ensemble d'équations :

- *foldr_filter :*
```
foldr f (filter p xs) a =
foldr (fun x r -> if (p x) then (f x r) else r)
xs a
```
- *filter_map :*
```
filter p (map g xs) =
foldr (fun x r -> if (p (g x)) then (g x)::r
else r) xs []
```
- *foldr_map :*
```
foldr f (map g xs) a =
foldr (comp f g) xs a
```

Comment montrer :

```
foldr f (filter p (map g xs)) a =
  foldr (comp
          (fun x r -> if p x then f x r else r)
          g) xs a
```

Motivation et problématique (2)

Éléments clés :

- *Orienter les équations*, par exemple
  `foldr f (map g xs) a` $\longrightarrow$
  `foldr (comp f g) xs a`
- *Appliquer les équations* uniquement dans le sens $\longrightarrow$

Les équations orientées sont appelées *règles de réécriture*.

Questions à se poser :

- *Confluence :* Est-ce qu'on peut appliquer les règles dans n'importe quel ordre ?
- *Complétude :* Est-ce que l'orientation des équations n'entraîne pas une perte d'information ?
- *Terminaison :* Est-ce que le processus de réécriture s'arrête ?

Motivation et problématique (3)

Exemple (suite) :
Application de *filter_map* à

```
foldr f (filter p (map g xs)) a = foldr (comp..) xs a
```

produit :

```
foldr f (foldr (...) xs []) a = foldr (comp ..) xs a
```

(*improvable* avec les équations connues)
Cependant : Application de *foldr_filter*

```
foldr (...) (map g xs) a = foldr (comp..) xs a
```

suivi de *foldr_map*

```
foldr (comp..) xs a = foldr (comp..) xs a
```

. . . vrai par reflexivité

Termes

Les termes sont composés de

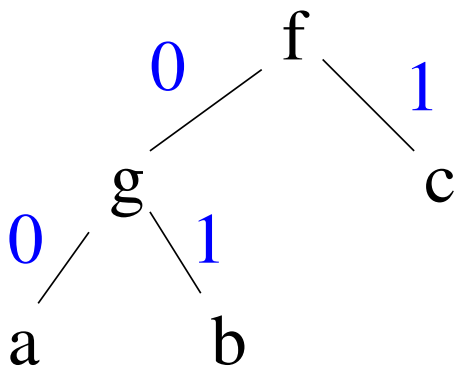
- Variables $x, y, z \ldots$
- Constantes $a, b, c, \ldots, f, g, h \ldots$
- Applications : $(f\ a)$

Un sous-terme de la forme `fun ->` `...` est traité comme une constante.

Positions

Positions : Liste de nombres désignant le sous-terme
Notation : sous-terme de t à la position p : $t|_p$
Dans le terme $f\ (g\ a\ b)\ c$

- c est à la position [1]
  donc : $f\ (g\ a\ b)\ c|_{[1]} = c$
- b est à la position [0; 1]
- $f\ (g\ a\ b)\ c$ est à la position []



Remplacement d'un terme s à la position p dans un terme t
Notation : $t[p \leftarrow s]$
Exemple : $(f\ (g\ a\ b)\ c)\ [[0] \leftarrow (h\ a)] = f\ (h\ a)\ c$
Implanter la fonction

- `pos` qui calcule le sous-terme d'un terme à une position
- $t[p \leftarrow s]$

Règles

Une règle a la forme $l \longrightarrow r$, où

- l et r sont des termes
- l n'est pas une variable
  (un système avec une règle $x \longrightarrow t$ ne terminerait jamais)
- $fv(r) \subseteq fv(l)$
  (on ne génère pas de variables)
  Exemple : $f\ x \longrightarrow g\ y$ n'est pas valide

Application d'une règle (1)

Application d'une règle à la racine (position $[\,]$) :
Ingrédients :

- Règle de réécriture $l \longrightarrow r$
- Terme t à réécrire

Procédure :

- Trouver une substitution σ telle que $l\sigma = t$
- Le résultat est $r\sigma$

Exemples : Règle : $R \equiv (f\ x\ b \longrightarrow g\ x)$

- Réécrire $(f\ (g\ a)\ b)$
  - Substitution : $\sigma = [x \leftarrow (g\ a)]$
  - Résultat : $g\ (g\ a)$

  On écrit : $(f\ (g\ a)\ b) \longrightarrow_R (g\ (g\ a))$
- Réécriture de $(f\ (g\ a)\ c)$ n'est pas possible

Application d'une règle (2)

Application d'une règle à la position p

Ingrédients :

- Règle de réécriture $l \longrightarrow r$
- Terme t à réécrire
- Position p

Procédure :

- Trouver une substitution σ telle que $l\sigma = t|_p$
- Le résultat est $t[p \leftarrow r\sigma]$

Exemple : Règle : $g\ x \longrightarrow h\ x\ x$

- Réécrire $(f\ (g\ a)\ b)$ à la position [0]
  - Substitution : $\sigma = [x \leftarrow a]$
  - Résultat : $(f\ (h\ a\ a)\ b)$
- Réécriture à la position [1] n'est pas possible

Application d'une règle (3)

Attention : La substitution lors de la réécriture s'applique uniquement à la règle et non pas au terme à réécrire
Exemples : Étant donné la règle $g\ x\ a \longrightarrow f\ x$

- Réécrire $h\ (g\ a\ x)\ x$
  Constat : la règle n'est pas applicable
- Réécrire $h\ (g\ b\ a)\ x$
  Le résultat est $h\ (f\ b)\ x$
  et non pas $h\ (f\ b)\ b$

Relations d'équivalence et de réduction (1)

Définitions : Un ensemble de règles $\mathcal{R} = \{l_1 \longrightarrow r_1, \ldots, l_n \longrightarrow r_n\}$ engendre les relations suivantes sur les termes :

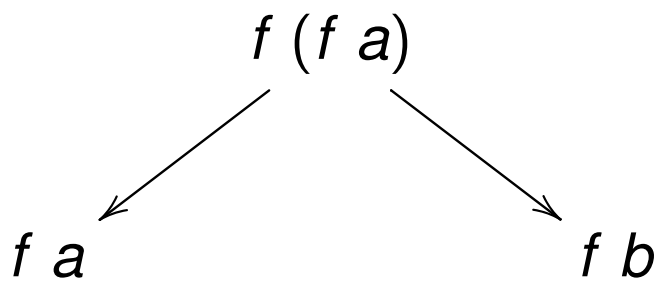
- $s \longrightarrow_{\mathcal{R}} t$ si $s \longrightarrow t$ à l'aide d'un $l_i \longrightarrow r_i \in \mathcal{R}$
- $\overset{+}{\longrightarrow}_{\mathcal{R}}$ est la fermeture transitive de $\longrightarrow_{\mathcal{R}}$
- $\overset{*}{\longrightarrow}_{\mathcal{R}}$ est la fermeture reflexive et transitive de $\longrightarrow_{\mathcal{R}}$
- $\overset{*}{\longleftrightarrow}_{\mathcal{R}}$ est la fermeture reflexive, transitive et symmétrique de $\longrightarrow_{\mathcal{R}}$
- Un terme s est *réductible* s'il existe un t tel que $s \longrightarrow_{\mathcal{R}} t$
- Un terme t est une *forme normale* de $\mathcal{R}$ s'il est irréductible pour $\mathcal{R}$.

On omet l'indice $\mathcal{R}$ si l'ensemble de règles est sous-entendu.

Relations d'équivalence et de réduction (2)

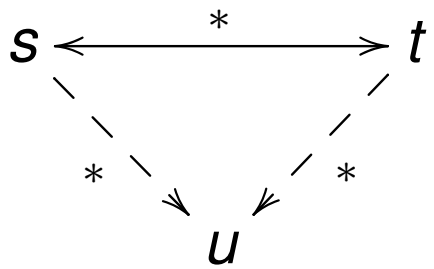
Exemples : Soit $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$

- $f\ (f\ a) \longrightarrow f\ b$ et $f\ (f\ a) \longrightarrow f\ a$
- $f\ (f\ a) \overset{+}{\longrightarrow} b$, mais non pas $f\ (f\ a) \overset{+}{\longrightarrow} f\ (f\ a)$
- $f\ (f\ a) \overset{*}{\longrightarrow} f\ (f\ a)$ et $f\ (f\ a) \overset{*}{\longrightarrow} b$
- $f\ a \overset{*}{\longleftrightarrow} f\ b$,
  mais ni $f\ a \overset{*}{\longrightarrow} f\ b$ ni $f\ b \overset{*}{\longrightarrow} f\ a$



Church-Rosser et Confluence (1)

Sous quelles conditions peut-on remplacer un raisonnement équationnel par un raisonnement par réécriture ?



Déf. : Deux termes s et t sont joignables (notation : $s \downarrow t$) s'il existe u tel que $s \xrightarrow{*} u$ et $t \xrightarrow{*} u$.

Déf. : Une relation $\longrightarrow$ a la propriété de Church-Rosser si

$\forall s, t.\ s \xleftrightarrow{*} t \Longrightarrow s \downarrow t$

Note historique :

- Alonzo Church (1903-1995)
- John Barkley Rosser (1907-1989)

sont parmi les fondateurs du Lambda-calcul et ont contribué à la théorie des relations de réduction

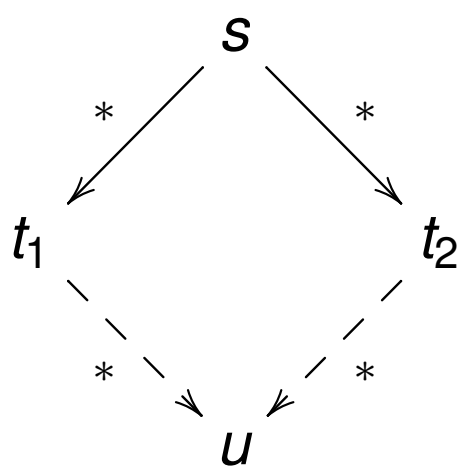
Church-Rosser et Confluence (2)

Déf. : Une relation $\longrightarrow$ est confluente si

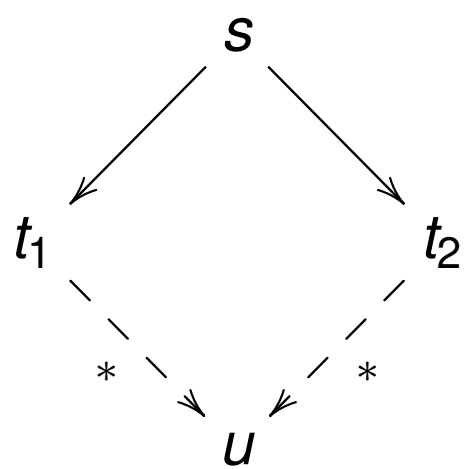
$$\forall s, t_1, t_2 . s \xrightarrow{*} t_1 \wedge s \xrightarrow{*} t_2 \Longrightarrow t_1 \downarrow t_2$$

Déf. : Une relation $\longrightarrow$ est localement confluente si

$$\forall s, t_1, t_2 . s \longrightarrow t_1 \wedge s \longrightarrow t_2 \Longrightarrow t_1 \downarrow t_2$$



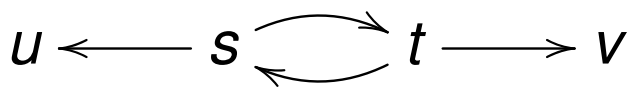
Confluence



Confluence locale

Church-Rosser et Confluence (3)

Théorème (CR –Confluence) : $\longrightarrow$ a la propriété de Church-Rosser si et seulement si $\longrightarrow$ est confluent.
Preuve : voir TD
Fait : "Confluence locale" n'implique pas "confluence"



Lemme (Newman) : Si $\longrightarrow$ termine, alors $\longrightarrow$ est confluent si et seulement si $\longrightarrow$ est localement confluent.
Preuve : voir TD

Church-Rosser et Confluence (4)

Conséquences : Si $\longrightarrow$ termine,

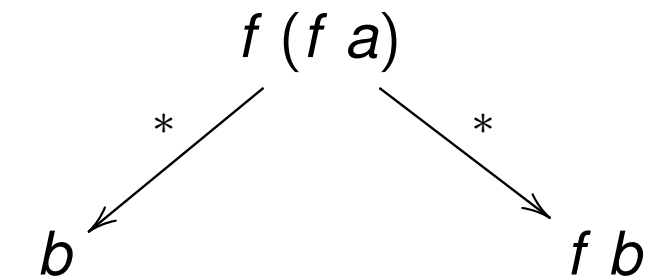
1. vérifier que $\longrightarrow$ est localement confluent
   (voir procédure de Knuth-Bendix . . .)
2. donc (par le Lemme de Newman) : $\longrightarrow$ est confluent
3. donc (par le théorème CR – confluence) : $\longrightarrow$ a la propriété de Church-Rosser
4. en particulier : pour déterminer si $s \overset{*}{\longleftrightarrow} t$:
   1. réduire $s \overset{*}{\longrightarrow} s'$, où s' est irréductible (la réduction termine !)
   2. réduire $t \overset{*}{\longrightarrow} t'$, où t' est irréductible (de même !)
   3. Si $s' = t'$, alors $s \overset{*}{\longleftrightarrow} t$
   4. Si $s' \neq t'$, alors non $s \downarrow t$, donc non $s \overset{*}{\longleftrightarrow} t$

Paires critiques (1)

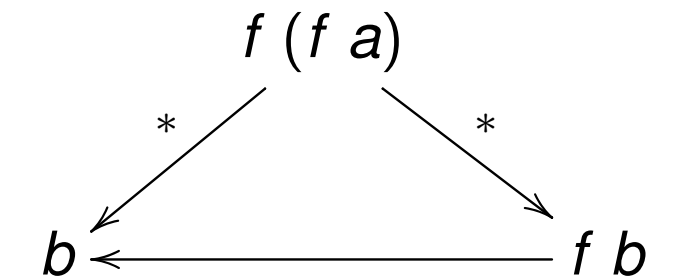
Pour une relation de réduction bien fondée, comment peut-on s'assurer de la confluence locale ?

1. Analyse des paires critiques : Analyse systématique des points de divergence
2. Complétion : Ajout de nouvelles règles pour faire converger les paires critiques $\rightsquigarrow$ procédure de Knuth-Bendix

Exemple : $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$



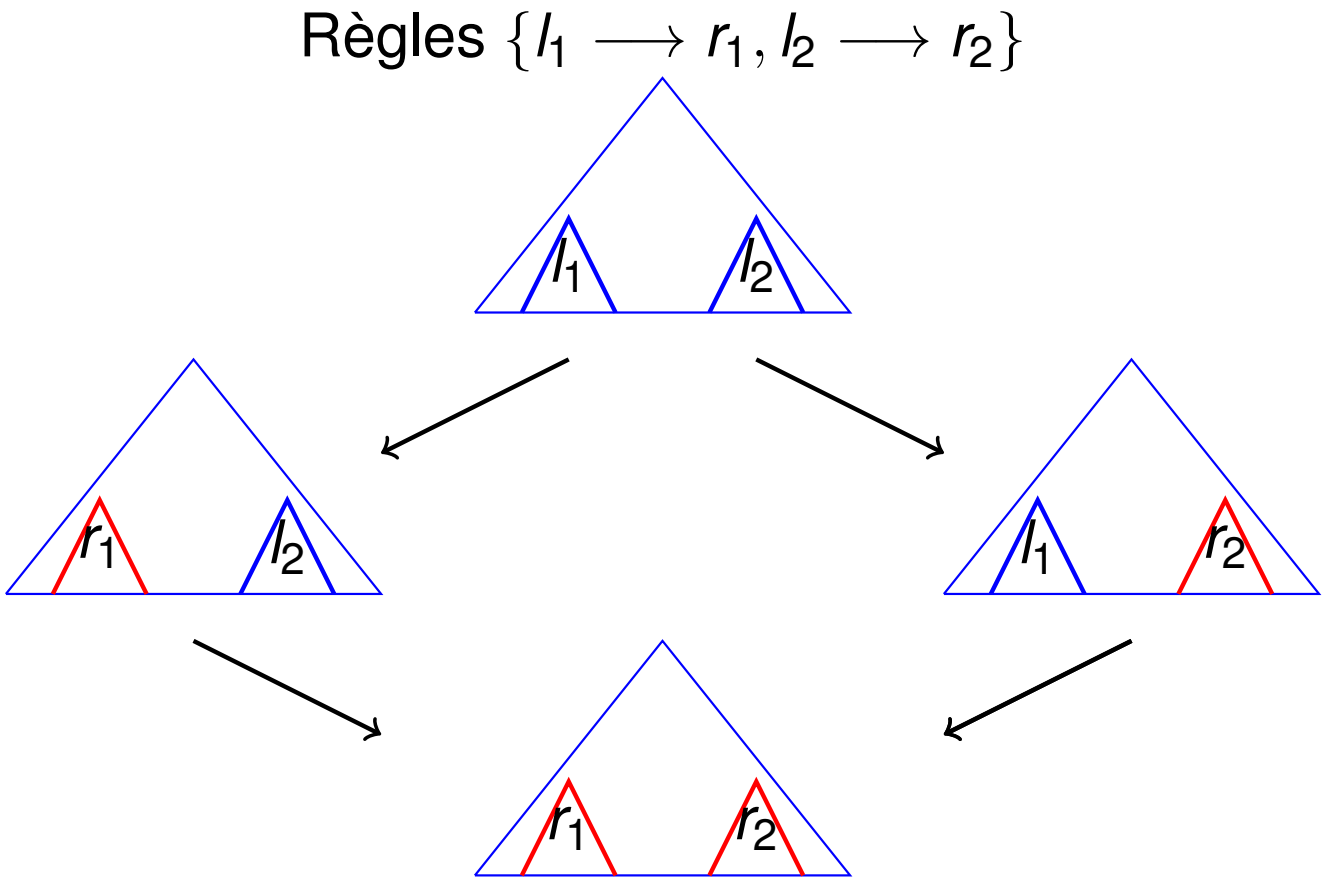
Paire critique : $(b, f\ b)$



Nouvelle règle : $f\ b \longrightarrow b$

Paires critiques (2)

Cas 1 : Réduction de sous-arbres distincts : jamais de paire critique



Paires critiques (3)

Réduction de sous-arbres distincts

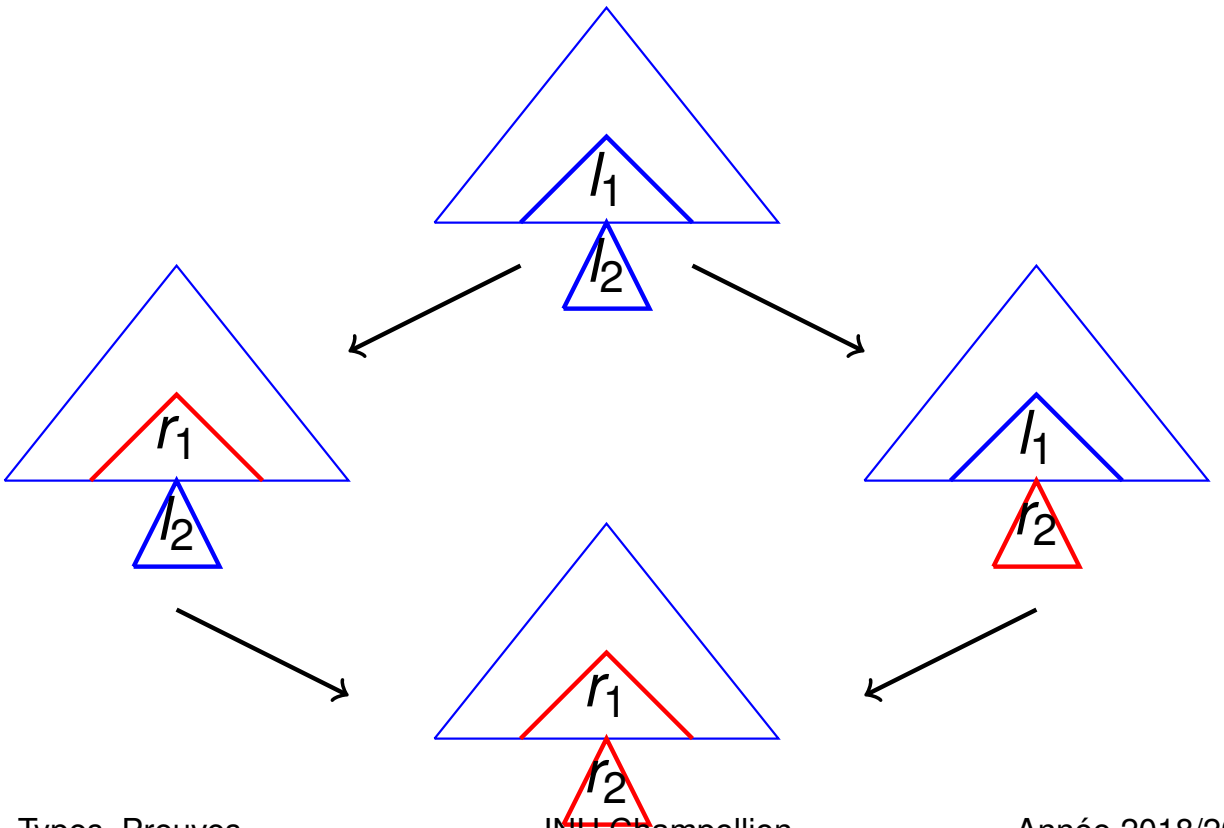
Exemple : Soit $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$

- $g\ \underline{(f\ (f\ b))}\ (f\ a) \longrightarrow g\ (f\ b)\ \underline{(f\ a)} \longrightarrow g\ (f\ b)\ b$
- $g\ (f\ (f\ b))\ \underline{(f\ a)} \longrightarrow g\ \underline{(f\ (f\ b))}\ b \longrightarrow g\ (f\ b)\ b$

Paires critiques (4)

Cas 2 : Sous-arbres qui se chevauchent (position de variable) :
Jamais de paire critique

- Une variable se trouve à la position de $l_2\sigma_2$ dans l_1
- Le sous-terme $l_2\sigma_2$ n'est pas altéré par la règle $l_1 \longrightarrow r_1$
- . . . mais $l_2\sigma_2$ peut être dupliqué ou supprimé



Paires critiques (5)

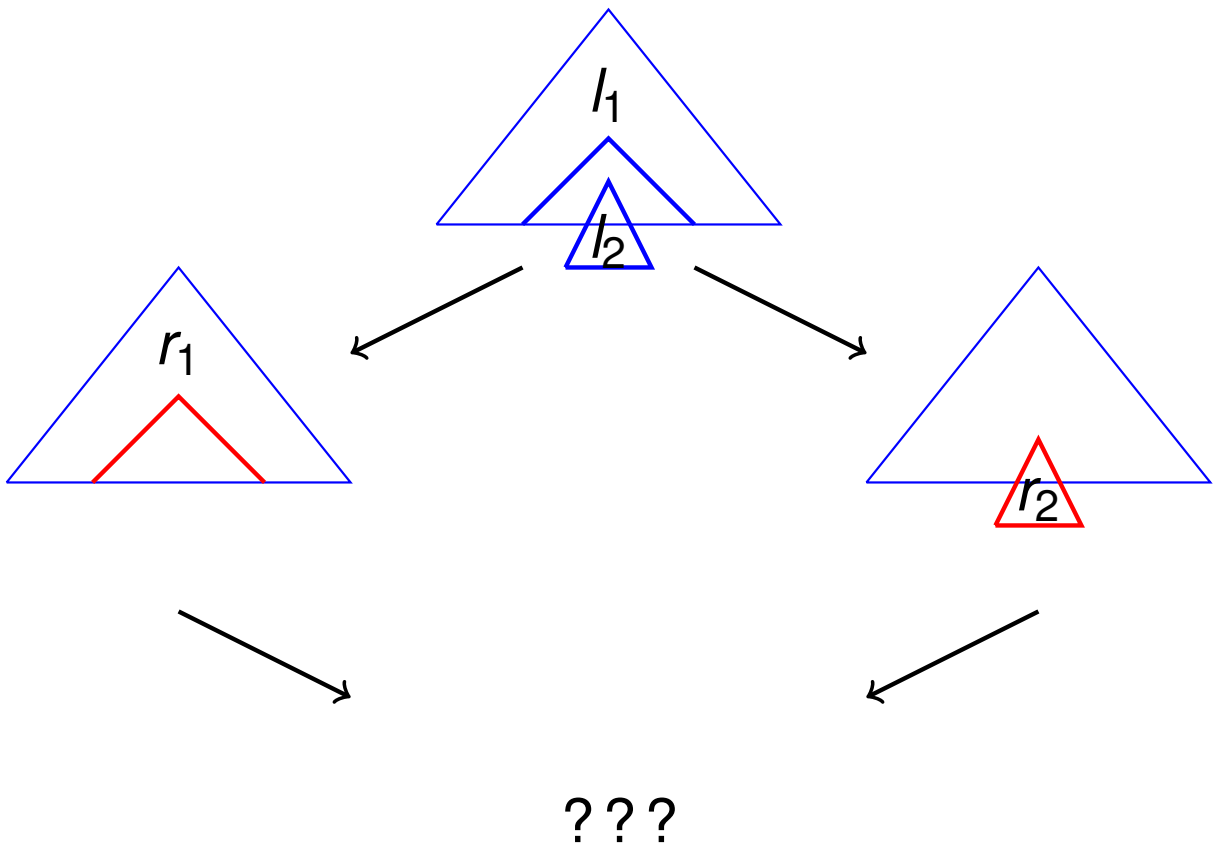
Réduction de sous-arbres distincts

Exemple : Soit $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$

- $\underline{(f\ (f\ (f\ a)))} \longrightarrow (f\ \underline{(f\ a)}) \longrightarrow (f\ b)$
- $(f\ (f\ \underline{(f\ a)})) \longrightarrow \underline{(f\ (f\ b))} \longrightarrow (f\ b)$

Paires critiques (6)

Cas 3 : Sous-arbres qui se chevauchent



Paires critiques (7)

Réduction de sous-arbres distincts

Exemple : Soit $\mathcal{R} = \{f\ (f\ x) \longrightarrow (f\ x), (f\ a) \longrightarrow b\}$

- $\underline{(f\ (f\ a))} \longrightarrow \underline{(f\ a)} \longrightarrow b$
- $(f\ \underline{(f\ a)}) \longrightarrow \underline{(f\ b)}$

Pair critique : $((f\ b),\ b)$
Signification informelle :

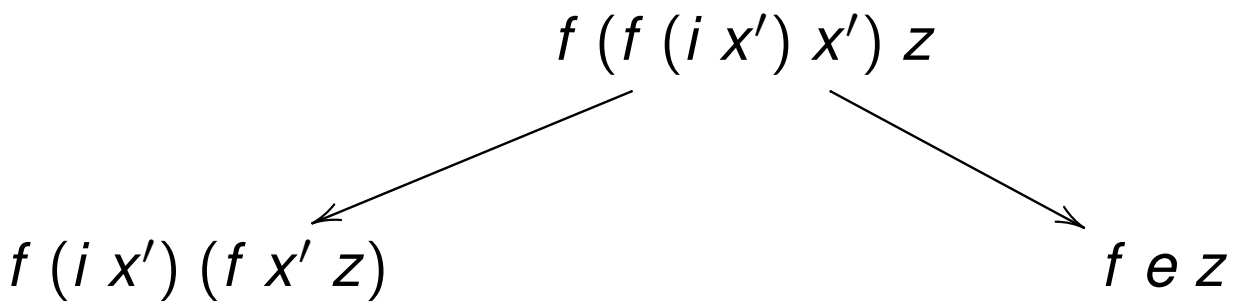
- On ne peut pas prouver $(f\ b) \overset{*}{\longleftrightarrow} b$ par réduction avec $\mathcal{R}$.
- Pourtant : $(f\ b) \overset{*}{\longleftrightarrow} (f\ (f\ a)) \overset{*}{\longleftrightarrow} (f\ a) \overset{*}{\longleftrightarrow} b$

Solution : Ajouter règle $(f\ b) \longrightarrow b$ à $\mathcal{R}$

Paires critiques : Définition formelle

Soient $l_1 \longrightarrow r_1$, $l_2 \longrightarrow r_2$ deux règles tq. $fv(l_1) \cap fv(l_2) = \{\}$.
Soit p une position de l_1 tq. $l_1|_p$ n'est pas une variable.
Soit $\sigma = mgu(l_1|p, l_2)$.
Alors, $(r_1\sigma, l_1\sigma[p \leftarrow r_2\sigma])$ est la paire critique des deux règles.

Attention aux occurrences multiples de variables :
$\mathcal{R} = \{f\ (f\ x\ y)\ z \longrightarrow f\ x\ (f\ y\ z),\ \ f\ (i\ x')\ x' \longrightarrow e\}$



Algorithme de complétion (1)

Procédure de Knuth-Bendix

Entrée : un ensemble E d'équations

si tous les $s = t \in E$ sont orientables
    R := orienter E
sinon échec;

faire
  $R' := R$;
  pour tout $(s, t) \in CP(R)$
    si on peut orienter (s, t) en $l \longrightarrow r$
      $R' := R' \cup \{l \longrightarrow r\}$;
    sinon terminer avec échec
tant que $R' \neq R$

renvoyer R'

Algorithme de complétion (2)

La procédure $CP(R)$ calcule toutes les paires critiques de R
Résultats possibles de l'algorithme de complétion :

- Un ensemble R. *Signification :*
  - R est confluent
  - La fermeture réflexive, symmétrique et transitive de R est E
- non-terminaison de l'algorithme
- échec : il existe des paires critiques qui ne sont pas orientables.
  Signification : ordre sur des termes éventuellement mal choisi

Algorithme de complétion (3)

Exemple : Soit $E = \{f(f\ x) = f\ x,\ f(f\ x) = g\ x,\ g(g\ x) = x\}$

- Orientation : $R_0 = \{f(f\ x) \longrightarrow f\ x,\ f(f\ x) \longrightarrow g\ x,\ g(g\ x) \longrightarrow x\}$
- Paire critique de la superposition de $f(f\ x)$ et $f(f\ x)$ à la pos. [ ] donne $((f\ x), (g\ x))$.
- $R_1 = \{f(f\ x) \longrightarrow f\ x,\ f(f\ x) \longrightarrow g\ x,\ g(g\ x) \longrightarrow x,\ (f\ x) \longrightarrow (g\ x)\}$
- Paire critiques de la superposition
  - de $f(f\ x)$ et $(f\ x)$ à la pos. [0] donne $((f\ x), x)$
    réduction : $f(f\ x) \longrightarrow (f\ x)$ et $f(f\ x) \longrightarrow f(g\ x) \longrightarrow g(g\ x) \longrightarrow x$
  - de $f(f\ x)$ et $(f\ x)$ à la pos. [0] donne $((g\ x), x)$
    réduction : $f(f\ x) \longrightarrow (g\ x)$ et $f(f\ x) \overset{*}{\longrightarrow} x$

Algorithme de complétion (4)

Exemple continué :

- $R_2 = \{f(f\ x) \longrightarrow f\ x,\ f(f\ x) \longrightarrow g\ x,\ g(g\ x) \longrightarrow x,\ (f\ x) \longrightarrow (g\ x),\ (f\ x) \longrightarrow x,\ (g\ x) \longrightarrow x\}$
- Pas d'autres paires critiques

On peut simplifier les règles de R_2, par exemple :

- $g(g\ x) \longrightarrow x$ superflu parce que simulable par deux applications de $(g\ x) \longrightarrow x$
- $f(f\ x) \longrightarrow f\ x$ superflu parce que instance de $(f\ x) \longrightarrow x$
- $f(f\ x) \longrightarrow g\ x$ superflu parce que $f(f\ x) \overset{*}{\longrightarrow} x$ et $g\ x \longrightarrow x$
- $f\ x \longrightarrow g\ x$ superflu parce que $f\ x \longrightarrow x$ et $g\ x \longrightarrow x$

Résultat : $R = \{(f\ x) \longrightarrow x,\ (g\ x) \longrightarrow x\}$

Algorithme de complétion (4)

Exemple continué : Pour
$E = \{f(f\ x) = f\ x,\ f(f\ x) = g\ x,\ g(g\ x) = x\}$
et $R = \{(f\ x) \longrightarrow x,\ (g\ x) \longrightarrow x\}$

1. Preuve de $f(f\ x) = g(f\ x)$
   - Par raisonnement équationnel :
     $f(f\ x) = f(f(f\ x)) = g(f\ x)$
     Nécessite la découverte d'un terme intermédiaire plus complexe
   - De manière automatique, par réduction :
     $f(f\ x) \stackrel{*}{\longrightarrow} x$ et $g(f\ x) \stackrel{*}{\longrightarrow} x$
2. Preuve de $(f\ a) \neq g(f\ b)$, pour constantes a, b
   - $(f\ a) \stackrel{*}{\longrightarrow} a$
   - $g(f\ b) \stackrel{*}{\longrightarrow} b$
   - On conclut $(f\ a) \neq g(f\ b)$, parce que $\longrightarrow$ est confluent et $a \neq b$

Plan

Plan

Problématique (1)

Facile : Programmes *de premier ordre* :

```
let f m n = m + n + 1 ;;
val f : int -> int -> int = <fun>
f 3 4 ;;
- : int = 8
```

- Fonction appelée avec tous ces arguments.
- Arguments ne sont pas des fonctions

$\rightsquigarrow$ modèle des langages impératifs classiques

Problématique (2)

Plus difficile :

- Application partielle
- Fonctions qui renvoient des fonctions
- Fonctions qui prennent des fonctions comme arguments

```
# let g a = f (3 * a) ;;
val g : int -> int -> int = <fun>
# g 2 ;;
- : int -> int = <fun>
# List.map (g 2) [1; 2; 3] ;;
- : int list = [8; 9; 10]
```

Comment compiler des langages fonctionnels ?

Démarche

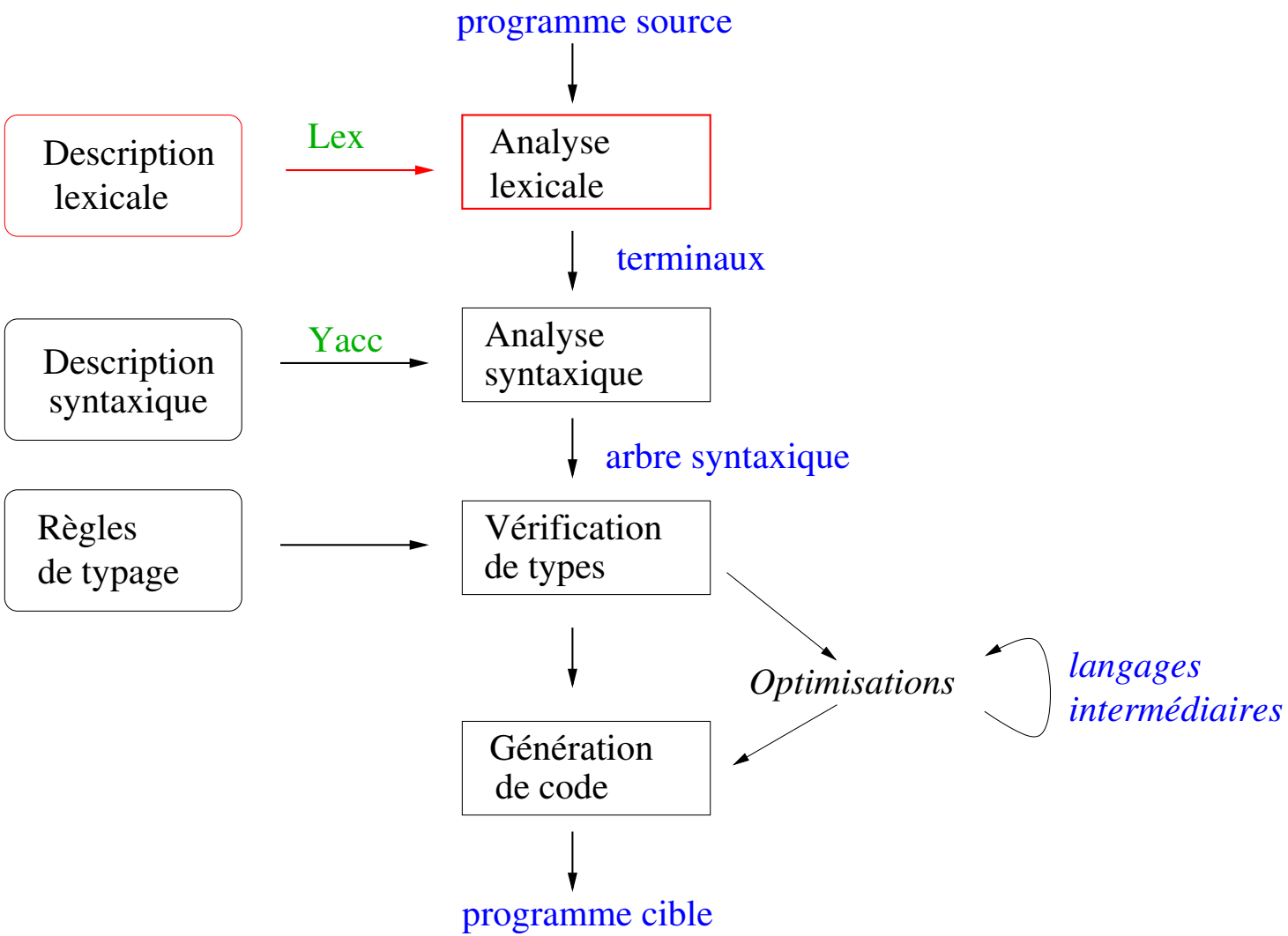
Cours :

1. Compilation d'un fragment fonctionnel pur vers une "machine abstraite" : CAM
   suivant l'article : Cousineau, Curien, Mauny : *The Categorical Abstract Machine*, Science of Computer Programming 8 (1987), pp. 173-202
   [*Note historique* : ML (Meta Language) dévelopé dans les années 1970 par Robin Milner
   CAM + ML $\rightsquigarrow$ Caml]
2. Rajout d'un mécanisme de définitions récursives `let rec ...` (avec quelques restrictions)

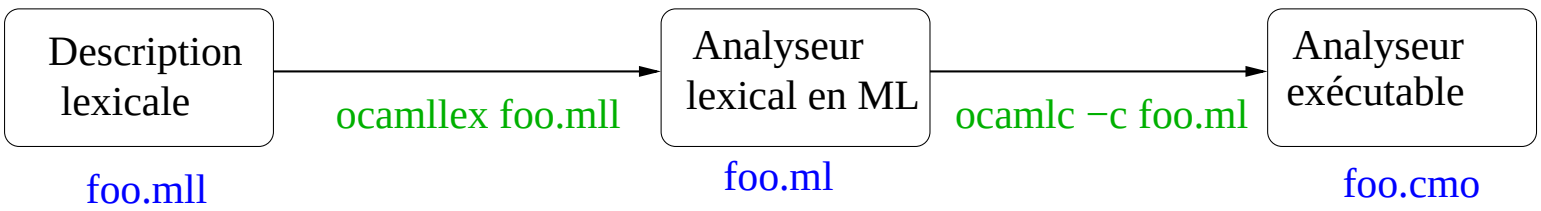
Projet : Compilation de Caml vers Java

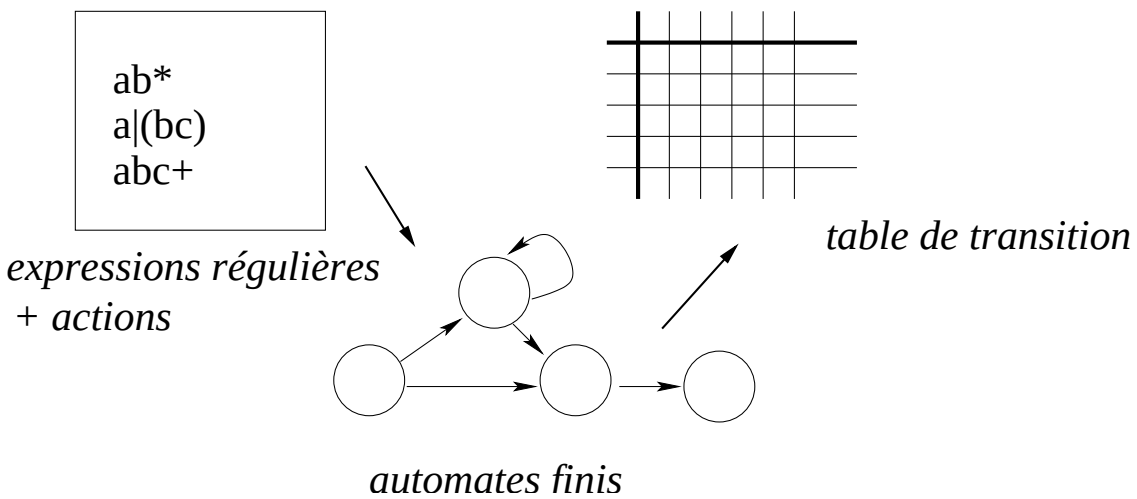
Plan

Situation dans le processus de compilation



Fonctionnement de Lex





Syntaxe des expressions régulières (1)

Caractères simples

`'x'`	le caractère `x`
`_`	(souligné) tout caractère
`'\n'`	newline

Classes de caractères

`['x''y''z']`	l'un des caractères `x`, `y`, `z`
`['A'-'Z']`	les car. `A...Z`
`[^'x''y''z']`	tout caractère sauf `x`, `y`, `z`
`[^'A'-'Z']`	tout caractère sauf `A...Z`
`"abc"`	la chaîne de caractère `abc`
`eof`	fin de l'entrée

Syntaxe des expressions régulières (2)

Opérateurs

`rs`	concaténation
`r|s`	alternative
`r?`	élément optionnel
`r*, r+`	répétition (0 fois ou plus, 1 fois ou plus)

. . . et beaucoup plus.

Regarder le manuel d'Ocaml, chap. 12

Organisation de la description lexicale (1)

```
Déclarations / définitions pour le programme ML
{
  exception Lexerror
}
Abréviations d'expressions régulières
let id = ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']*
Expressions régulières et actions associées
rule main = parse
    id ";"        { print_string "..." }
  | ['0'-'9']+    { print_string "..." }
Autres fonctions et programme principal
{
main (Lexing.from_channel stdin)
}
```

Organisation de la description lexicale (2)

Analyse d'une chaîne de caractères

- entrée : un *lex buffer*
  Ici : `(Lexing.from_channel stdin)`
- analyse : fonction appliquée au *lex buffer*
  Ici : `main`
  En général : Toute fonction `f` définie par
  `rule f` a_1 `...` a_n `= parse ...`
- Sortie : élément d'un type de données. Ici : `unit`

Librairie `Lexing` :

`lexbuf`	variable prédéfinie dans la directive `parse`
`from_channel`	crée un *lex buffer*
	Ex. : `Lexing.from_channel stdin`
`lexeme`	dernier mot reconnu
	Ex. : `Lexing.lexeme lexbuf`

Voir manuel d'Ocaml, chap. 12

Lex : Exemple

```
(* règle paramétrée *)
rule count nl nc = parse
      (* lexbuf devient argument explicit ! *)
    '\n'   { count (nl + 1) (nc + 1) lexbuf }
  | _      { count nl (nc + 1) lexbuf }
  | eof    { (nl, nc) }

{
  let lexbuf = Lexing.from_channel stdin in
  let (nl, nc) = count 0 0 lexbuf in
    Format.printf
     "nb of lines = %d, nb of chars = %d\n" nl nc
}
```

Lex : Comment compiler ?

. . . à la main :

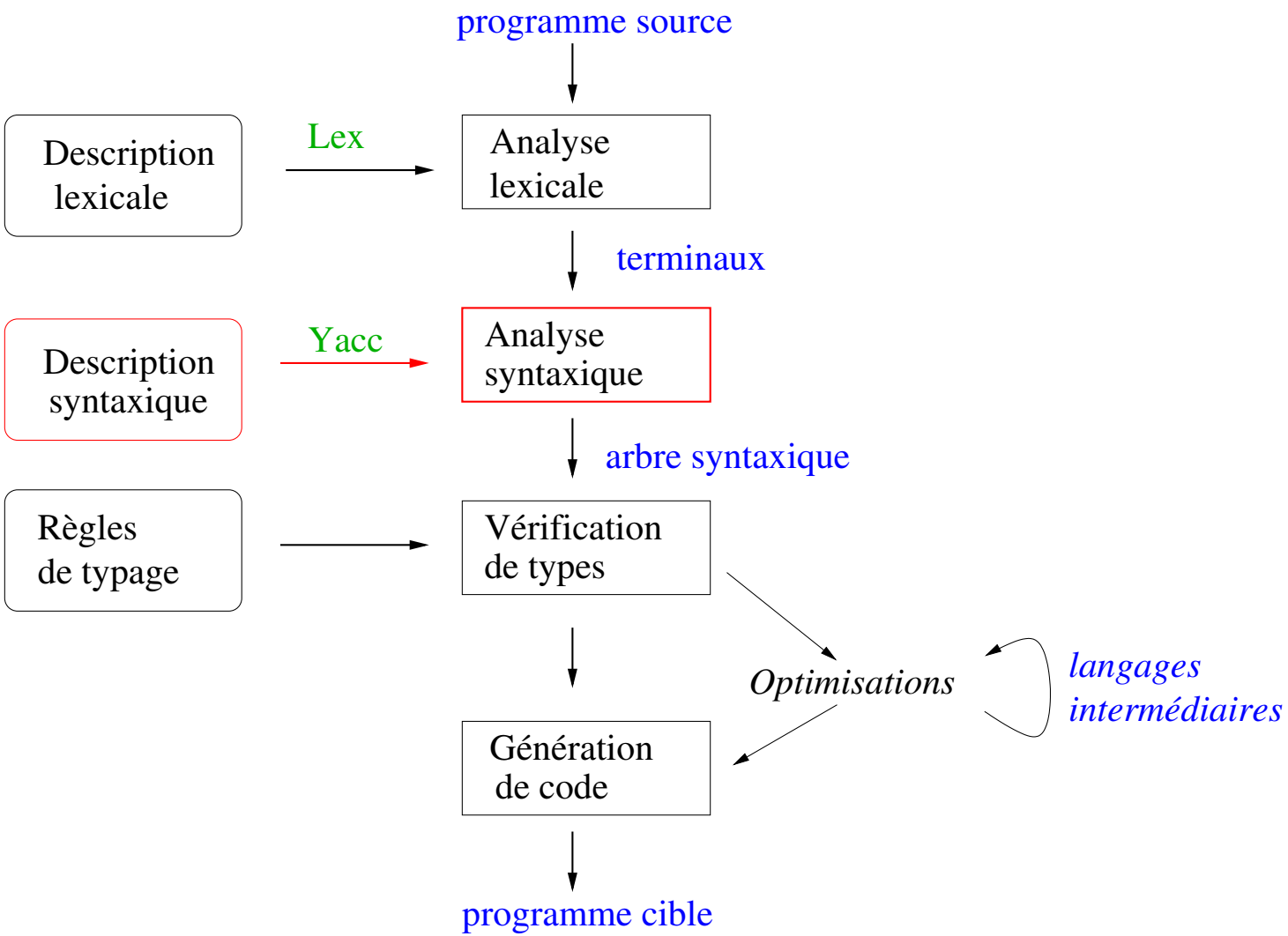
- `ocamllex linecount.mll` ⇝ `linecount.ml`
- `ocamlc -o linecount linecount.ml` ⇝ `linecount`
- Exécuter `linecount < fichier`

. . . éléments d'un Makefile :

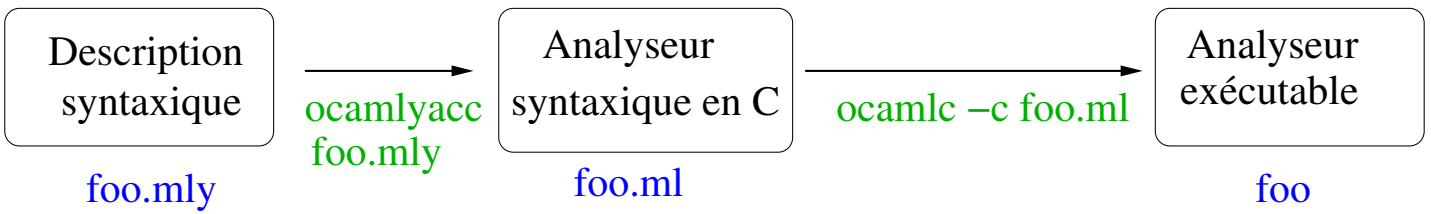
```
linecount: linecount.cmo
	ocamlc -o linecount $<
linecount.cmo: linecount.ml
	ocamlc -c $<
linecount.ml: linecount.mll
	ocamllex $<
```

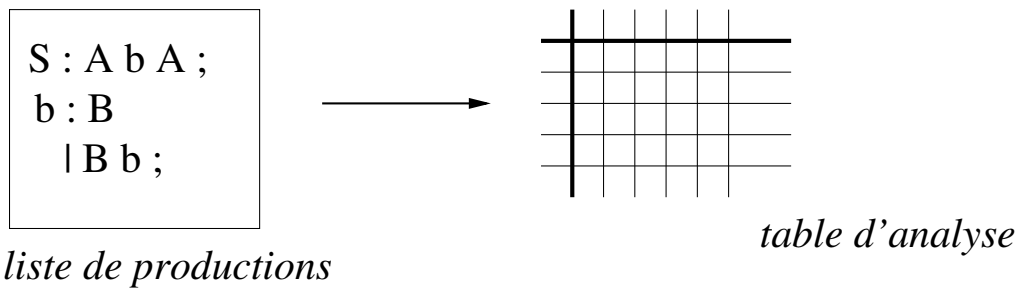
Plan

Situation dans le processus de compilation



Fonctionnement de Yacc





Organisation de la description syntaxique

```
Déclarations / définitions pour le programme ML
%{  open Lang    (* partie optionnelle *)
%}
Déclaration de propriétés de symboles
%start s
%%
Règles de production et actions sémantiques
s : A b A       { printf "..." }
;
b : B           { printf "..." }
  | B b         { printf "..." }
;
%%
Fonctions et programme principal
let main  = ...    (* partie optionnelle *)
```

Déclaration de propriétés de symboles

Racine (déclaration obligatoire)

`%start` *non-terminal*

Terminaux

`%token<`*type*`>` *liste de terminaux*

Non-terminaux (décl. obligatoire pour racine)

`%type<`*type*`>` *liste de non-terminaux*

Associativité et priorité des terminaux

`%left` *liste de terminaux*
`%right` *liste de terminaux*

Exemple :

```
%left PLUS MINUS
%left MULT DIV
```

Actions sémantiques

En général : Expr. d'OCAML comprises entre `{ ... }`

```
b : B        { print_string "règle b1" }
  | B b      { print_string "règle b2" }
;
```

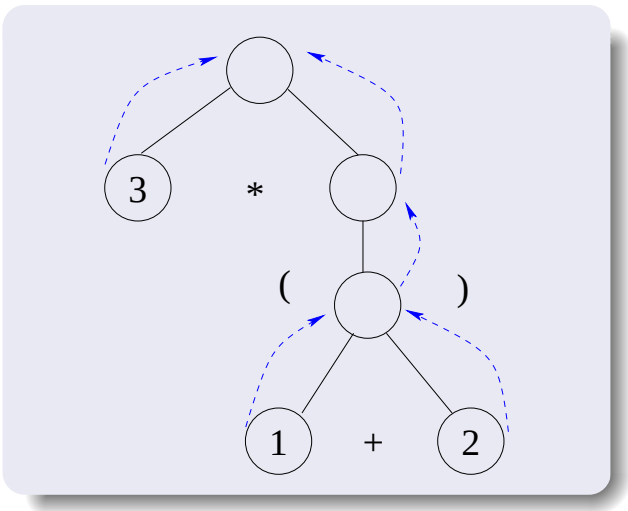
Actions sémantiques

En général : Expr. d'OCAML comprises entre `{ ... }`

```
b : B          { print_string "règle b1" }
  | B b        { print_string "règle b2" }
;
```

Accès aux sous-arbres :

```
e : e PLUS e      { $1 + $3 }
  |  ....
  | LPAR e RPAR     { $2 }
;
```



Conflits

Grammaire ambiguë :

```
s : A b C     {}
|   A B C     {}
;
b : B         {}
;
```

Analyse de conflits :

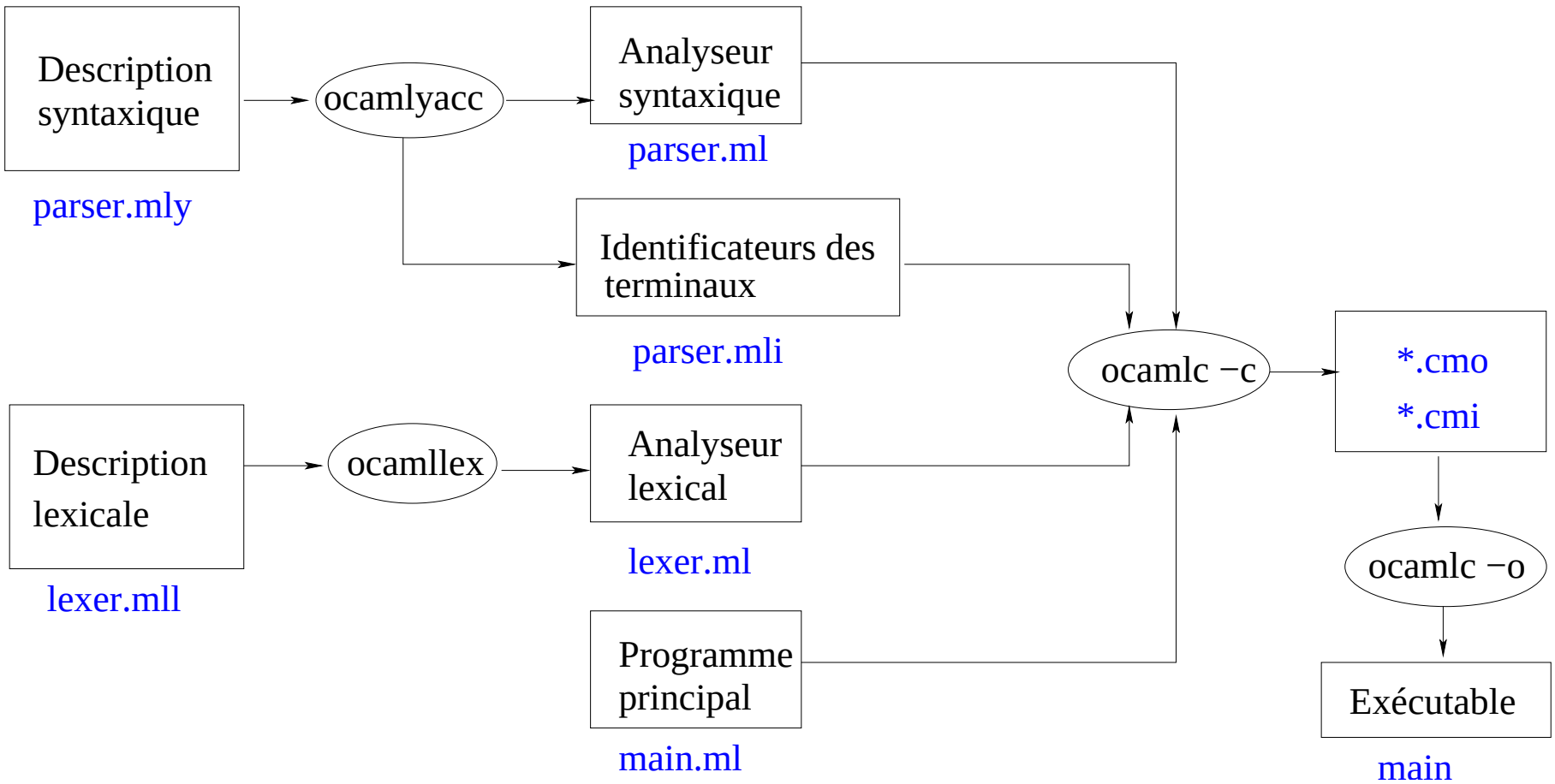
`ocamlyacc -v foo.mly` crée `foo.output`

```
5: shift/reduce conflict (shift 7, reduce 3) on C
state 5
        s : A B . C   (2)
        b : B .   (3)

        C  shift 7
```

Plan

Schéma de compilation



Exemple : Fichier Lex

```
{ open Parser
  exception Eof   ... }

let blancs  = [' ''\t']+
let boolean = 'T'|'F'    ...

rule token = parse
  blancs           { token lexbuf } (* appel recursif *)
| boolean as b     { if b = 'T' then (BCONST true)
                                else (BCONST false) }
| num as i         { INTCONST (int_of_string i)}
| '^'              { OPERATOR }
| '\n'             { EOL }
| eof              { raise Eof }
| _                { Printf.printf
          "unrecogized '%s'\n" (Lexing.lexeme lexbuf);
                     raise Lexerror }
```

Exemple : Fichier Yacc (1)

Déclarations :

```
%token <bool> BCONST
%token <int> INTCONST
%token OPERATOR
%token EOL
%left  OPERATOR
%type <bool> term expr formule
%start formule
```

Exemple : Fichier Yacc (2)

Grammaire :

```
%%
formule: expr EOL { $1 }
;
expr: expr OPERATOR expr { $1 && $3 }
 | term { $1 }
;
term: BCONST { $1 }
 | INTCONST { not ($1 = 0) }
;
```

voir `parser.mly`

Exemple : Programme principal

```
let _ =
  try
    let lexbuf = Lexing.from_channel stdin in
      while true do
        let result = Parser.formule Lexer.token lexbuf in
          print_string ("Result: " ^ (string_of_bool result
          print_newline(); flush stdout
      done
  with Lexer.Eof ->
    Format.printf "Finished\n"; exit 0
```

Plan

Fragment fonctionnel pur (1)

Ingrédients :

- variables : x, y, z
- constantes : 2, 3, *true*, *false*
- opérations élémentaires :
  `fst`, `snd`,
  `+`, `*`, `<=`, `&&` ...
- `if ... then ... else`
- paires : (a, b)
- application de fonction : $(f\ a)$
- abstraction $fun\, x \rightarrow e$

Le type en Caml :

```
type mlexp =
  Var of var
| Bool of bool
| Int of int
| PrimOp of primop
| Cond of
   mlexp * mlexp * mlexp
| Pair of mlexp * mlexp
| App of mlexp * mlexp
| Fn of var * mlexp
```

Fragment fonctionnel pur (2)

Restrictions essentielles : N'est pas représentable dans le fragment fonctionnel pur : *récursion*

```
let rec fac n = ... fac (n - 1) ... in (fac 5)
```

$\rightsquigarrow$ voir extension plus tard

Absence de *types inductifs* (listes, arbres) et *filtrage* (`match ... with`)

Restrictions inessentielles :

Ce qu'on peut simuler : `let` simple :

```
let f = fun n -> n + 2 in (f 3)
```

équivalent à : `(fun n -> n + 2) 3`

Fonctions à plusieurs arguments : `fun x y -> x + y`

équivalent à : `fun x -> (fun y -> x + y)`

Décomposition de motifs : `fun (x, y) -> x + 1`

équivalent à : `fun p -> (fst p) + 1`

Rappel : réductions

Cadre : Évaluation stricte :

- évaluer d'abord les arguments d'une fonction
- remplacer les paramètres de la fonction par les valeurs obtenues

Exemple :

```
      (fun x -> (fun y -> x * y)) (1 + 2) 4
⟶s    (fun x -> (fun y -> x * y)) 3 4
⟶s    (fun y -> 3 * y) 4
⟶s    3 * 4
⟶s    12
```

Réductions : aspects gênants

- Inefficacité de traverser un gros terme lors de la substitution :
  `(fun x -> (fun f -> f 2) (fun y -> x + y)) 3`
  $\longrightarrow_s$ `(fun f -> f 2) (fun y -> 3 + y)`
  et de reconstruire le résultat après.
- Duplication de structures, accroissement des termes :
  `(fun f -> f (f 5)) (fun x -> x + 1)`
  $\longrightarrow_s$ `((fun x -> x + 1) ((fun x -> x + 1) 5))`
- En principe : capture de variables libres par des variables liées :
  `(fun x -> fun y -> x + y) y`
  $\longrightarrow_s$ `fun y -> y + y` (faux !)
  $\longrightarrow_s$ `fun y' -> y + y'` (correct !)
  . . . un non-problème : ici, nous considérons uniquement des termes clos (sans variables libres)

Réductions avec contextes

Idée : Éviter des substitutions en construisant des contextes lors de la réduction :

```
        ((fun x -> (fun y -> x * y)) 3 4)[]
 ⟶s     ((fun y -> x * y) 4)[x := 3]
 ⟶s     (x * y)[y := 4, x := 3]
 ⟶s     (x[y := 4, x := 3] * y[y := 4, x := 3])
 ⟶s     3 * 4
 ⟶s     12
```

Clôtures (1)

Lors d'applications partielles, une application de fonction ne se réduit pas à une valeur scalaire.
Comparer :

- Réduction par substitution :

```
(fun x -> (fun y -> x * y)) 3
```

$\longrightarrow_s$ `(fun y -> 3 * y)`

- Réduction avec contextes :

```
((fun x -> (fun y -> x * y)) 3)[]
```

$\longrightarrow_s$ `(fun y -> x * y)[x:=3]`

Une association d'une *fonction* et d'un *environnement* est appelée une clôture (*en anglais : closure*).

Clôtures (2)

On a intérêt à traiter des clôtures comme des valeurs à part entière :

- pour les renvoyer comme résultat d'une fonction :
  en Caml :

```
# (fun x -> (fun y -> x * y)) 3 ;;
- : int -> int = <fun>
```

- pour les passer à d'autres fonctions :

```
(fun f -> (f 2)) ((fun x -> (fun y -> x * y)) 3)
⟶s (fun f -> (f 2)) ((fun y -> x * y)[x:=3])
⟶s (f 2)[f:= ((fun y -> x * y)[x:=3])]
⟶s (((fun y -> x * y)[x:=3]) 2)
⟶s (x * y)[y:= 2; x:=3]
```

Plan

Les valeurs de la CAM (1)

La partie *environnement* de la clôture est codée par des paires imbriqués (tête de la liste à droite) :
`[y:= 2; x:=3]` est représenté comme : $(((), 3), 2)$
en Caml : `PairV(PairV(NullV, IntV(3)), IntV(2))`

La partie *fonction* de la clôture est représentée par des instructions de la CAM (et pas par le langage source).

Les noms sont omis de l'environnement. Les instructions de la CAM sauront accéder la bonne valeur.
Exemple : Accès à la variable *x* avec les instructions `Fst; Snd`

Les valeurs de la CAM (2)

Ingrédients :

- Valeur nulle : ()
- Variables
- Valeurs correspondant aux constantes
- Paires
- Clôtures

Le type en Caml :

```
type value =
  NullV
| VarV of var
| IntV of int
| BoolV of bool
| PairV of value * value
| ClosureV of code * value
and code = instr list
and instr = ...
```

Le type `instr` des instructions sera défini plus bas.

Configurations de la CAM

Structure : La CAM manipule un triplet :

- Terme (auquel on applique une instruction) : une valeur
- Code (liste d'instructions)
- Pile (*stack* ; pour sauvegarder des éléments intermédiaires)
  Ces éléments peuvent être :
  - une valeur
  - du code

```
type stackelem = Val of value | Cod of code
```

Sémantique opérationnelle de la CAM
Règles de transition lors de l'exécution d'une instruction :

$$(t, instr :: c, st) \longrightarrow_c (t', c', st')$$

Les règles suivantes sont légèrement simplifiées

Instructions de la CAM

Opérations élémentaires :

- Projections d'une paire, *Fst* :

$$(PairV(x, y), Fst :: c, st) \longrightarrow_c (x, c, st)$$

- Projections d'une paire, *Snd* :

$$(PairV(x, y), Snd :: c, st) \longrightarrow_c (y, c, st)$$

- Addition *Add* :

$$(PairV(IntV(m), IntV(n)), Add :: c, st) \longrightarrow_c (IntV(m + n), c, st)$$

- . . . et ainsi de suite pour les autres opérations

Instructions de la CAM

Constante :

$$(t, \mathit{Quote}(v) :: c, st) \longrightarrow_c (v, c, st)$$

Construction d'une paire :

$$(x, \mathit{Cons} :: c, \mathit{Val}(y) :: st) \longrightarrow_c (\mathit{PairV}(y, x), c, st)$$

Instructions de la CAM

Sauvegarde d'une valeur sur la pile :

$$(x, \mathit{Push} :: c, st) \longrightarrow_c (x, c, \mathit{Val}(x) :: st)$$

Échange de la valeur actuelle et sommet de la pile :

$$(x, \mathit{Swap} :: c, \mathit{Val}(y) :: st) \longrightarrow_c (y, c, \mathit{Val}(x) :: st)$$

Instructions de la CAM

Construction d'une clôture

$$(x, Cur(c1) :: c, st) \longrightarrow_c (ClosureV(c1, x), c, st)$$

Application d'une clôture

$$(PairV(ClosureV(cd, y), z), App :: c, st) \longrightarrow_c (PairV(y, z), cd, Cod(c) :: st)$$

Retour d'une fonction

$$(x, Return :: c, Cod(c') :: st) \longrightarrow_c (x, c', st)$$

Instructions de la CAM

Branchement :

- cas *then :*

$$(BoolV(true), Branch(t, e) :: c, Val(x) :: st) \longrightarrow_c (x, t, Cod(c) :: st)$$

- cas *else :*

$$(BoolV(false), Branch(t, e) :: c, Val(x) :: st) \longrightarrow_c (x, e, Cod(c) :: st)$$

Résumé : instructions de la CAM

```
type instr =
  PrimInstr of primop
| Quote of value
| Cons
| Push
| Swap
| Cur of code
| App
| Return
| Branch of code * code
```

Plan

Principe de la compilation

L'essentiel : Le compilateur

- prend comme entrée une expression e du langage source
- et produit une séquence d'instructions *ins* du langage cible : $\mathit{compile}(e) = \mathit{ins}$

Quel rapport entre e et *ins* ?

Correction de la compilation, première version :

- *Langage source (ML) :* on évalue e pour obtenir une valeur v_e :

$$e \longrightarrow_s^* v_e$$

- *Langage cible (CAM) :* on exécute *ins* à partir d'une configuration initiale, pour obtenir une configuration finale :

$$((), \mathit{ins}, []) \longrightarrow_c^* (v_c, [], [])$$

pour des valeurs v_e et v_c qui "correspondent" : $v_e \simeq v_c$ (à préciser !)

Compilation : Constantes

Définition :

- $compile(Bool(b)) = Quote(BoolV(b))$
- $compile(Int(i)) = Quote(IntV(i))$

Vérifier, par exemple pour *Bool* :

- $Bool(b) \longrightarrow_s^* Bool(b)$
- $((), [Quote(BoolV(b))], []) \longrightarrow_c^* (BoolV(b), [], [])$
- avec : $Bool(b) \simeq BoolV(b)$

Compilation : Variables (1)

Difficultés :

- Quelle est la valeur d'une variable (sous la sémantique de réduction) ?
  Elle est contenue dans un environnement !
  Ex. : `y[y := 4, x := 3]` $\longrightarrow_s 4$
- Où se trouve l'environnement ?
  Codé dans le terme (lors d'un accès à une variable).

Correction de la compilation, raffinement :
Si l'évaluation de e dans env produit la valeur v_e : $e[env] \longrightarrow_s^* v_e$
alors $(t_{env}, ins, []) \longrightarrow_c^* (v_c, [], [])$
où t_{env} est le terme qui code l'environnement env

Compilation : Variables (2)

Gestion des variables dans *compile* :

- La fonction prend un argument de plus : *env*
- . . . qui est une liste de *noms de variables*
- (les valeurs sont inconnues lors de la compilation)

Définition :

- $compile(env, Var(v)) = access\ v\ env$

Exemple :

- *Réduction :*
  `(fun x -> fun y -> x) 2 3` $\longrightarrow_s$ `x[y:= 3; x:=2]` $\longrightarrow_s 2$
- $compile([y; x], Var(x)) = [Fst; Snd]$
- *Exécution :*
  $(((), 2), 3), [Fst; Snd], []) \longrightarrow_c (((), 2), [Snd], []) \longrightarrow_c ((2, [], [])$

Compilation : Paires (1)

Définition :

- $compile(env, Pair(e_1, e_2)) =$
  $[Push]@compile(env, e_1)@[Swap]@compile(env, e_2)@[Cons]$

Vérification : Soit $Pair(e_1, e_2)[env] \longrightarrow_s PairV(v_1, v_2)$.

Complétez la preuve !

Compilation : Paires (1)

Définition :

- $compile(env, Pair(e_1, e_2)) =$
  $[Push]@compile(env, e_1)@[Swap]@compile(env, e_2)@[Cons]$

Vérification : Soit $Pair(e_1, e_2)[env] \longrightarrow_s PairV(v_1, v_2)$.

Hypothèse d'induction : Sachant que
$e_1[env] \longrightarrow_s v_1$ et $e_2[env] \longrightarrow_s v_2$ et
$(t_{env}, ce_1, []) \longrightarrow_c^* (v_1, [], [])$ et $(t_{env}, ce_2, []) \longrightarrow_c^* (v_2, [], [])$

Compilation : Paires (1)

Définition :

- $compile(env, Pair(e_1, e_2)) =$
  $[Push]@compile(env, e_1)@[Swap]@compile(env, e_2)@[Cons]$

Vérification : Soit $Pair(e_1, e_2)[env] \longrightarrow_s PairV(v_1, v_2)$.

Hypothèse d'induction : Sachant que
$e_1[env] \longrightarrow_s v_1$ et $e_2[env] \longrightarrow_s v_2$ et
$(t_{env}, ce_1, []) \longrightarrow_c^* (v_1, [], [])$ et $(t_{env}, ce_2, []) \longrightarrow_c^* (v_2, [], [])$

On déduit :
$(t_{env}, [Push]@ce_1@[Swap]@ce_2@[Cons], []) \longrightarrow_c$
$(t_{env}, ce_1@[Swap]@ce_2@[Cons], [t_{env}]) \longrightarrow_c^*$
$(v_1, [Swap]@ce_2@[Cons], [t_{env}]) \longrightarrow_c$
$(t_{env}, ce_2@[Cons], [v_1]) \longrightarrow_c^*$
$(v_2, [Cons], [v_1]) \longrightarrow_c$
$(PairV(v_1, v_2), [], [])$

Compilation : Paires (2)

Le raisonnement précédent est une instance d'une *preuve par induction* sur la relation $\longrightarrow_s$.

Une nouvelle généralisation (code / pile) :
Correction de la compilation, raffinement :
Si l'évaluation de e dans env produit la valeur v_e : $e[env] \longrightarrow_s^* v_e$
alors $(t_{env}, ins@c, st) \longrightarrow_c^* (v_c, c, st)$
où t_{env} est le terme qui code l'environnement env

Compilation : Application de fonction élémentaire

Définition :

- $compile(env, App(PrimOp(p), e)) =$
  $compile(env, e)@[PrimInstr(p)]$

Exemple (simplifié) :

- Compilation :
  $compile(env, App(PrimOp(Add), Pair(2, 3))) =$
  $compile(env, Pair(2, 3))@[PrimInstr(Add)]$
- Exécution :
  $((), compile(env, Pair(2, 3))@[PrimInstr(Add)], []) \longrightarrow_c^*$
  $(PairV(2, 3), [PrimInstr(Add)], []) \longrightarrow_c$
  $(5, [], [])$

Compilation : Abstraction

Définition :

- $compile(env, Fn(v, e)) = [Cur((compile(v :: env, e))@[Return])]$

Vérification :

- Une abstraction `Fn(v, e)` est déjà une valeur
- Une abstraction sous environnement :
  `Fn(v, e)[env]` $\longrightarrow_s$ `Fn(v, e[env])`
- Exécution CAM : $(t_{env}, Cur(c_e@[Return]) :: c, st)$
  $\longrightarrow_c (ClosureV(c_e@[Return], t_{env}), c, st)$
- avec `Fn(v, e[env])` $\simeq ClosureV(c_e@[Return], t_{env})$
  et $c_e = compile(env, e)$

Compilation : Application et Abstraction

Définition :

- pour f autre que *PrimOp* :
  $compile(env, App(f, a)) =$
  $[Push]@compile(env, f)@[Swap]@compile(env, a)@[Cons; App]$

Vérification :

- *Note :* Si f n'est pas une opération élémentaire et $f \longrightarrow_s f'$, alors f' est de la forme $Fn(v, e)$ (typage !)

A faire :

- Compilez `(fun v -> v) 3`
- Exécutez le code résultant

Compilation : Conditionnel

Définition :

- $compile(env, Cond(i, t, e)) = [Push]@compile(env, i)@[Branch(compile(env, t)@[Return], compile(env, e)@[Return])]$

Faire la vérification !

Plan

Syntaxe

Informellement : fonctions mutuellement récursives :

```
let rec even = fun n -> (n = 0)  || odd  (n - 1)
    and  odd = fun n -> (n <> 0) && even (n - 1)
in even 7 ;;
```

Syntaxe abstraite :

```
type mlexp = ...
| Fix of (var * mlexp) list * mlexp
```

Extension de la structure de la CAM

Configuration de la CAM : quadruplet

- *Comme avant :* terme ; code ; pile
- *Nouveau :* pile de définitions de fonctions
  (liste d'association : nom de la fonction, expression de la fonction)

Instructions :

- *Trois nouvelles instructions :* `Call`, `AddDefs`, `RmDefs`
- *Autres instructions :* ne modifient pas la pile des définitions

Extension de la CAM : Instructions

Appel d'une fonction :

$$(t, \mathit{Call}(f) :: c, st, fds) \longrightarrow_c (t, (\mathit{List.assoc}\ f\ fds)@c, st, fds)$$

Exemple :

```
let rec f = fun n -> n + 1 in
(let rec g = fun n -> n + 2 in
let rec f = fun n -> n + 3 in f 4) + f 4 ;;
```

Résultat : 12

Extension de la CAM : Instructions

Ajout de fonctions locales :

$$(t, \mathit{AddDefs}(\mathit{defs}) :: c, st, \mathit{fds}) \longrightarrow_c (t, c, st, \mathit{defs}@\mathit{fds})$$

Retrait de n fonctions locales :

$$(t, \mathit{RmDefs}(n) :: c, st, \mathit{fds}) \longrightarrow_c (t, c, st, \mathit{chop}\ n\ \mathit{fds})$$

où *chop n fds* enlève n éléments de la liste *fds*

Compilation : Environnement (1)

Motivation Comparer :

```
let rec f=fun n-> n+1 in ((fun g-> f 3)(fun m-> m+2))
```

Résultat : 4 et :

```
let rec f=fun n-> n+1 in ((fun f-> f 3)(fun m-> m+2))
```

Résultat : 5

Fragment fonctionnel pur :
Liste de variables, *par ex. :* `[y; x]`
Accès par position : $compile([y; x], Var(x)) = [Fst; Snd]$

Compilation : Environnement (2)

Avec appels récursifs : Environnement est une liste de

- variables *ou*
- liste de noms de fonctions (définitions mutuellement récursives !)

Éléments de l'environnement :

```
type envelem = EVar of var | EDef of var list
```

Exemples :

- env_1 = `[EVar "g"; EDef["f"]]`
- env_2 = `[EVar "f"; EDef["f"]]`

Compilation : Environnement (3)

Écrire nouvelle fonction `access` qui génère :

- une liste de `Fst` / `Snd` pour une `EVar`
- un `Call` pour une `EDef`

Exemples :

- `access "f" [EVar "g"; EDef["f"]]` donne : `[Call "f"]`
- `access "f" [EVar "f"; EDef["f"]]` donne : `[Snd]`

Modifications du compilateur existant :

- $compile(env, Var(v)) = access\ v\ env$
  avec la nouvelle fonction
- $compile(env, Fn(v, e)) = [Cur((compile(EVar(v) :: env, e))@[Return])]$

Compilation de `let rec`

Démarche pour compiler `Fix(defs, e)`

- Compiler les définitions de fonction `defs` pour obtenir `dc` :
  - rajouter les noms des fonctions de `defs` à l'environnement,
  - compiler le corps de chacune des fonctions de `defs` dans cet environnement
- Pour compiler l'expression `e` pour obtenir `ec` :
  - rajouter les noms des fonctions de `defs` à l'environnement,
  - compiler `e` dans cet environnement
- Code généré :

```
[AddDefs dc] @ ec @ [RmDefs (List.length dc)]
```

Attention récursion mutuelle !

Exemple `even / odd` :

La déf de `odd` doit être connue en compilant `even` et inversément.

Plan

Plan

Sémantique : C'est quoi ? (1)

La sémantique d'une langue

- décrit la *signification* des phrases de la langue
- ... contrairement à la *syntaxe*, qui décrit leur structure

Ces deux notions s'appliquent aux langues naturelles et artificielles (langages de programmation).

Sémantique : C'est quoi ? (2)

Prérequis pour déterminer la sémantique d'une phrase : correction syntaxique.

Exemples :

- $(3 + x) - (/ * 8)$
  - syntaxiquement mal formé
- $(3 + x) - (y * 8)$
  - syntaxiquement bien formé
  - signification : si $x = 20$ et $y = 2$, alors le résultat est 7
  - Comment le définir mieux ?

En langue naturelle, la situation est moins nette.

Exemple : Time flies like arrows

Sémantique : Pourquoi ? (1)

Mille et une raisons, parmi lesquelles :

- *Désambiguïser des expressions :*
  Le résultat de `(x = 3) * (x + 2)` pour $x = 1$ est (??) :
  - 15 (évaulation “gauche avant droite”)
  - 9 (évaulation “droite avant gauche”)
- *Clarification de cas extrèmes :*
  `MAXINT + 1` est (??) :
  - `MININT`
  - une erreur

Important à savoir pour le programmeur

Sémantique : Pourquoi ? (2)

- *Équivalence de programmes :*
  Est-il correct d'optimiser

```
while (x > 3) {
    a = f(y);
    ....}
```

  en éliminant le calcul multiple de `a = f(y)` :

```
a = f(y);
while (x > 3) {
    ....}
```

  Techniques utilisées dans les compilateurs modernes

Sémantique : Pourquoi ? (3)

- *Correction de programmes :*
  Est-il possible que le programme

```
if (x * x - 1 == 0) {
    y = 5 / x; }
else {
    y = 5 / x - 1;}
```

  provoque une division par 0 ? . . . pour être sûr que, cette fois, la fusée Ariane n'explose pas

Plan

Définition du langage : Plan

Le langage défini dans la suite sera

- *réduit* par rapport à un langage de programmation réel (comme C)
  Ex. : pas de boucle `for` ; pas de pointeurs
- *idéalisé*
  Ex. : expressions sans effet de bord ; valeurs d'expressions toujours définies

Structure globale du langage

Définition préliminaire : Les valeurs affectées aux variables d'un programme constituent son *état*.
Exemple : $\sigma = \{x = 3; y = 4\}$ est un état pour un programme avec les variables x, y.
Le langage est divisé en trois grandes catégories :

- *Expressions* calculent une *valeur* sans modifier l'état.
  Ex. : `(x + 2) * y` a la valeur 20 dans σ
- *Instructions* modifient l'état sans calculer de valeur.
  Ex. : `x = x + 1` change σ en $\sigma' = \{x = 4; y = 4\}$
- *Procédures* sont des abstractions d'instructions.

Expressions

Deux classes d'expressions, définies de manière inductive :

- Arithmétiques a
  - Nombres n
  - Variables x
  - Opérations binaires : $a_1 + a_2$, $a_1 - a_2$, $a_1 * a_2$
- Boléennes b
  - Constantes `true`, `false`
  - Comparaisons : $a_1 = a_2$, $a_1 \leq a_2$
  - Négation : $\neg b$
  - Conjonction : $b_1 \wedge b_2$

À faire : Concevoir des types `aexpr, bexpr` en Caml

Instructions (1)

Instructions c (définition inductive) :

- Programme vide : `Skip`
- Affectation : $x := a$
- Séquence : c_1 `;` c_2
- Sélection : `if (`b`) {`c_1`} else {`c_2`}`
- Boucle : `while (`b`) {`c`}`

Fonctions, procédures, entrées / sorties : plus tard

À faire : Concevoir un type `com` en Caml

Instructions (2)

Traitement de

- sélection sans `else` :
  Traduire en `if - then - else` :
  `if (`b`) {`c_1`}` devient
  `if (`b`) {`c_1`} else {Skip}`
- boucle `for` :
  Traduire en initialisation ; boucle `while`
- boucle `do .. while` :
  Traduire en séquence ; boucle `while`

Syntaxe concrète / abstraite

La syntaxe concrète est la représentation externe d'une unité syntaxique. Elle

- est parfois redondante. *Ex. :* Accolades superflues dans :

```
if (x > 0) {x = x + 1;}
```

- s'appuie sur des conventions (règles de précédence . . .)
  Ex. : $2 + 3 * 4$ et $2 + (3 * 4)$ sont équivalents

La syntaxe abstraite est la représentation interne. Elle est unique pour éléments syntaxiquement équivalents.
Donner l'arbre syntaxique pour les expressions en haut !

Langage : Résumé

Rappel : Distinction entre :

- Expressions (arithmétique, booléen)
- Instructions

Définitions en sémantique basées sur syntaxe abstraite.
Limitations du langage peuvent

- en partie être levées de manière syntaxique (*ex. :* codage de `for` par `while`)
- nécessitent parfois un cadre plus complexe (*ex. :* pointeurs)

Plan

Sémantique : Principes

On définit la sémantique comme suit :

- pour des *expressions* : sémantique de réduction $\rightarrow_s$
  - *arithmétiques a* : une fonction $\mathcal{A}(a, \sigma)$, qui calcule la valeur de a dans l'état σ
  - *booléennes b* : une fonction $\mathcal{B}(b, \sigma)$, qui calcule la valeur de b dans l'état σ
- pour des *instructions* : sémantique de transition :
  une relation $\langle c, \sigma \rangle \rightarrow_t \sigma'$ qui transforme un état σ en état σ' par exécution de l'instruction c

Sémantique : Manipulation de l'état

Un état σ définit les valeurs de chaque variable du programme.
État comme type abstrait, avec les opérations suivantes :

- État initial σ_0, renvoie 0 pour toute variable
- Mise à jour de la valeur d'une variable x d'un état σ avec une valeur v, écrit : $\sigma.x \leftarrow v$
- Sélection de la valeur d'une variable x d'un état σ, écrit : $\sigma.x$

Quelques équivalences :

- $(\sigma.x \leftarrow v).x = v$
- $(\sigma.x \leftarrow v).y = \sigma.y$ (pour $x \neq y$)

Sémantique : Représentation de l'état

Préalable : type de valeurs :

```
type value =
  NullV
| VarV of var
| IntV of int
| BoolV of bool
...
```

Représentation de l'état en Caml : alternativement :

- *représentation concrète :* liste d'association
  `(var * value) list`
- *représentation abstraite :* comme fonction
  - totale (utilisée dans la suite) : `var -> value`
  - ou partielle : `var -> value option`

Évaluation d'expressions (1)

Définition par récursion sur la structure des expressions :

$$
\begin{array}{lcl}
\mathcal{A}(n,\sigma) & \rightarrow_s & n \\
\mathcal{A}(x,\sigma) & \rightarrow_s & \sigma.x \\
\mathcal{A}(e_1 + e_2,\sigma) & \rightarrow_s & \mathcal{A}(e_1,\sigma) + \mathcal{A}(e_2,\sigma) \\
\ldots & & \\
\mathcal{B}(\texttt{true},\sigma) & \rightarrow_s & \texttt{true} \\
\ldots & &
\end{array}
$$

complétez – et écrivez les fonctions Caml correspondantes :

```
aeval : aexpr -> state -> int
beval : bexpr -> state -> bool
```

Évaluation d'expressions (2)

Calculer

- $\mathcal{A}(x + 4, \{x = 3; y = 2\})$
- $\mathcal{A}(x * 0, \sigma)$

Observations : L'évaluation des expressions est

- une fonction *totale* (toujours définie)
- déterministe

Discussion : Comment traiter une expression a_1 / a_2 ?

Exécution d'instructions

Définition de la relation $\langle c, \sigma \rangle \rightarrow_t \sigma'$ par induction.

Affectation

$$\frac{}{\langle x := a, \sigma \rangle \rightarrow_t (\sigma.x \leftarrow \mathcal{A}(a, \sigma))}$$

Exemple :

$\langle x := x + y, \{x = 5; y = 4\} \rangle \rightarrow_t \{x = 9; y = 4\}$

Exécution d'instructions

Skip

$$\frac{}{\langle \texttt{Skip}, \sigma \rangle \to_t \sigma}$$

Séquence

$$\frac{\langle c_1, \sigma \rangle \to_t \sigma' \quad \langle c_2, \sigma' \rangle \to_t \sigma''}{\langle (c_1 ; c_2), \sigma \rangle \to_t \sigma''}$$

Exemple :

$\langle x := x + y, \{x = 5; y = 4\} \rangle \to_t \{x = 9; y = 4\}$

$\langle y := 1, \{x = 9; y = 4\} \rangle \to_t \{x = 9; y = 1\}$

donc :

$\langle x := x + y; y := 1, \{x = 5; y = 4\} \rangle \to_t \{x = 9; y = 1\}$

Exécution d'instructions

Sélection

$$\frac{\mathcal{B}(b,\sigma) = \mathit{true} \quad \langle c_1, \sigma\rangle \to_t \sigma'}{\langle \texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}},\sigma\rangle \to_t \sigma'}$$

$$\frac{\mathcal{B}(b,\sigma) = \mathit{false} \quad \langle c_2, \sigma\rangle \to_t \sigma'}{\langle \texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}},\sigma\rangle \to_t \sigma'}$$

Exemple :

$\langle$`if (x < 7) {x:=x + y} else {y:=1}`$,\{x = 5; y = 4\}\rangle \to_t$???

Exécution d'instructions

Boucle

$$\frac{\mathcal{B}(b,\sigma) = \mathit{true} \quad \langle c,\sigma\rangle \to_t \sigma' \quad \langle \texttt{while (}b\texttt{) \{}c\texttt{\},}\sigma'\rangle \to_t \sigma''}{\langle \texttt{while (}b\texttt{) \{}c\texttt{\},}\sigma\rangle \to_t \sigma''}$$

$$\frac{\mathcal{B}(b,\sigma) = \mathit{false}}{\langle \texttt{while (}b\texttt{) \{}c\texttt{\},}\sigma\rangle \to_t \sigma}$$

Exemples :

$\langle$`while (x<3) {x:= x+1; y:=x+y},`$\{x = 1; y = 4\}\rangle \to_t ???$

$\langle$`while (x<3) {x:= x-1; y:=x+y},`$\{x = 1; y = 4\}\rangle \to_t ???$

Exécution d'instructions

Observations : L'exécution des instructions

- est une relation déterministe : $\langle c, \sigma \rangle \to_t \sigma'$ et $\langle c, \sigma \rangle \to_t \sigma''$ impliquent $\sigma' = \sigma''$
- ... mais pas totale :
  il existe c, σ pour lesquels il n'y a pas de σ' avec $\langle c, \sigma \rangle \to_t \sigma'$
  Lesquels ?

Plan

Rappel : Induction sur les structures inductives

Exemple des listes :

Définition inductive de la structure :

α *list* est le plus petit ensemble / type défini par :

- $[] \in \alpha$ *list*
- si $x \in \alpha$ et $xs \in \alpha$ *list*, alors $x :: xs \in \alpha$ *list*

Principe d'induction : pour tout prédicat P :

- Si $P([])$
- et $\forall x, xs. P(xs) \longrightarrow P(x :: xs)$
- alors $\forall \ell : \alpha$ *list*. $P(\ell)$

alternativement : définition ensembliste : pour tout ensemble E :
si $[] \in E$ et . . .(*complétez !*)

Induction sur des relations (1)

Définition inductive de la fermeture réflexive-transitive R^* d'une relation R :
R^* est la plus petite relation définie par :

- $R^*(x, x)$
- si $R(x, y)$ et $R^*(y, z)$, alors $R^*(x, z)$

Exercice : montrer que si $R(a, b)$ et $R(b, c)$, alors $R^*(a, c)$

Principe d'induction associé : pour tout prédicat P :

- si $\forall x. P(x, x)$
- et $\forall x,\ y,\ z.\ R(x, y) \longrightarrow R^*(y, z) \longrightarrow P(y, z) \longrightarrow P(x, z)$
- alors $\forall x,\ z.\ R^*(x, z) \longrightarrow P(x, z)$

Induction sur des relations (2)

Preuve par induction sur des relations :
Exemple 1 : Soit $R(a, b)$ et $R(b, c)$ et $\neg R(x, y)$ pour tous les autres $x, y \in \{a, b, c\}$. Montrer $\neg R^*(c, a)$.
Preuve par induction :

- Réécrire $\neg R^*(b, a)$ comme : $\forall x\, z. R^*(x, z) \longrightarrow \neg(x = c \wedge z = a)$
- Appliquer le principe d'induction avec $P(x, z) := \neg(x = c \wedge z = a)$
- Montrer $\forall x. \neg(x = c \wedge x = a)$
  évident pour constantes $a \neq c$
- Montrer $\forall x,\ y,\ z.\ R(x, y) \longrightarrow R^*(y, z) \longrightarrow \neg(y = c \wedge z = a) \longrightarrow \neg(x = c \wedge z = a)$

compléter la preuve

Induction sur des relations (2)

Preuve par induction sur des relations :
Exemple 1 : Soit $R(a, b)$ et $R(b, c)$ et $\neg R(x, y)$ pour tous les autres $x, y \in \{a, b, c\}$. Montrer $\neg R^*(c, a)$.
Preuve par induction :

- Réécrire $\neg R^*(b, a)$ comme : $\forall x\, z. R^*(x, z) \longrightarrow \neg(x = c \wedge z = a)$
- Appliquer le principe d'induction avec $P(x, z) := \neg(x = c \wedge z = a)$
- Montrer $\forall x. \neg(x = c \wedge x = a)$
  évident pour constantes $a \neq c$
- Montrer $\forall x,\ y,\ z.\ R(x, y) \longrightarrow R^*(y, z) \longrightarrow \neg(y = c \wedge z = a) \longrightarrow \neg(x = c \wedge z = a)$

$\forall x,\ y,\ z.\ R(x, y) \longrightarrow R^*(y, z) \longrightarrow (x = c \wedge z = a) \longrightarrow (y = c \wedge z = a)$
$\forall x,\ y,\ z.\ R(c, y) \longrightarrow R^*(y, a) \longrightarrow (y = c)$
vrai parce que $\neg\exists y. R(c, y)$.
Faire la preuve de : $\neg R^*(b, a)$.

Induction sur des relations (3)

Exemple 2 : montrer que $R*$ est transitive :
$\forall a, b, c.\ R^*(a, b) \longrightarrow (R^*(b, c) \longrightarrow R^*(a, c))$
Preuve par induction : fixer a, b, c, instancier
$P(x, z) := (R^*(z, c) \longrightarrow R^*(x, c))$
compléter la preuve

Induction sur la relation d'exécution

Principe d'induction pour la relation d'exécution :

- si $\forall x\, \sigma.\ P((\texttt{Skip}, \sigma), \sigma)$
- si $\forall x,\ a,\ \sigma.\ P((\texttt{x} := \texttt{a}, \sigma), (\sigma.x \leftarrow \mathcal{A}(a, \sigma)))$
- si $\langle c_1, \sigma\rangle \rightarrow_t \sigma' \longrightarrow P((c_1, \sigma), \sigma') \longrightarrow \langle c_2, \sigma'\rangle \rightarrow_t \sigma'' \longrightarrow P((c_2, \sigma'), \sigma'') \longrightarrow P(((c_1; c_2), \sigma), \sigma'')$
- ...
- alors $\forall c,\ \sigma,\ \sigma'.\ \langle c, \sigma\rangle \rightarrow_t \sigma' \longrightarrow P((c, \sigma), \sigma')$

Plan

Extraction d'instructions communes (1)

. . . d'une sélection (*Exemple*) :

original . . .

```
if (x < 5) {
  y = x + 1;
  x = x + 1;
  }
else {
  y = x + 1;
  x = x - 1;
  }
```

transformé . . .

```
y = x + 1;
if (x < 5) {
  x = x + 1;
  }
else {
  x = x - 1;
  }
```

Les deux programmes sont équivalents (pourquoi ?), mais le deuxième est plus court / lisible.

Extraction d'instructions communes (2)

original . . .

```
n = 5;
x = 0;
y = 3;
while (x < 10) {
  y = f(x, y);
  z = g(n);
  x = x + 1;
  }
```

transformé . . .

```
n = 5;
x = 0;
y = 3;
z = g(n);
while (x < 10) {
  y = f(x, y);
  x = x + 1;
  }
```

Les deux programmes sont équivalents, mais le code transformé est plus efficace.

Extraction d'instructions communes (3)

Schéma général de l'extraction d'une sélection :

original . . .

```
if (b) { c1; c2; }
else { c1; c3; }
```

transformé . . .

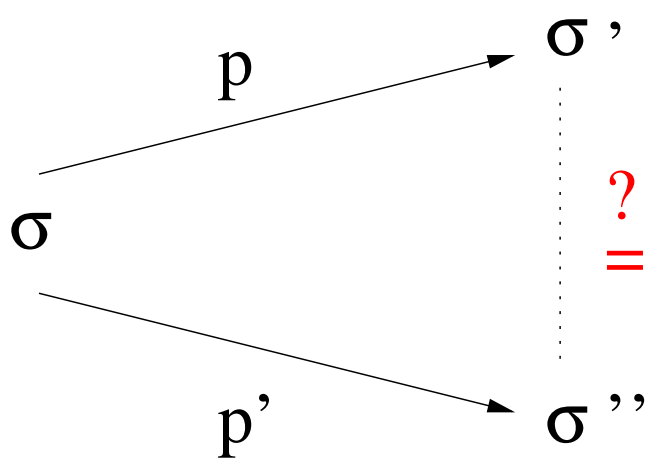
```
c1;
if (b) { c2; }
else { c3; }
```

Cette transformation est-elle correcte ? On verra . . .
Soit p le programme original, p' le programme transformé.

Preuves d'équivalence de programmes (1)

Équivalence de programmes : p et p' sont dits *équivalents*, $p \sim p'$, s'ils induisent le même changement d'état : pour tout $\sigma, \sigma', \sigma''$:

- si $\langle p, \sigma \rangle \rightarrow_t \sigma'$
- et $\langle p', \sigma \rangle \rightarrow_t \sigma''$
- alors $\sigma' = \sigma''$



Preuves d'équivalence de programmes (2)

Principe de preuve "standard" : Inductions de règle imbriquées :
Si $\langle p, \sigma \rangle \to_t \sigma'$ est de la forme :

- $\langle \texttt{Skip}, \sigma \rangle \to_t \sigma$: alors, si $\langle p', \sigma \rangle \to_t \sigma''$ est de la forme :
  - $\langle \texttt{Skip}, \sigma \rangle \to_t \sigma$, montrer $\sigma = \sigma$
  - $\langle (c_1; c_2), \sigma \rangle \to_t \sigma_2$, montrer $\sigma = \sigma_2$
  - etc.
- $\langle (c_1; c_2), \sigma \rangle \to_t \sigma_2$: alors, si ...
- $\langle \texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}}, \sigma \rangle \to_t \sigma_2$, avec $\mathcal{B}(b, \sigma) = \mathit{true}$
- $\langle \texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}}, \sigma \rangle \to_t \sigma_2$, avec $\mathcal{B}(b, \sigma) = \mathit{false}$
- etc.

Preuves d'équivalence de programmes (3)

Appliqué à l'extraction d'instructions communes :
Seuls cas applicables :

- $\langle \texttt{if (}b\texttt{) \{}c_1\texttt{;}c_2\texttt{;\} else \{}c_1\texttt{;}c_3\texttt{\}}, \sigma\rangle \to_t \sigma_1$,
  avec $\mathcal{B}(b, \sigma) = \mathit{true}$
  - $\langle (c_1\texttt{;if (}b\texttt{) \{}c_2\texttt{;\} else \{}c_3\texttt{\}}), \sigma\rangle \to_t \sigma_2$,
    montrer $\sigma_1 = \sigma_2$
- pareil, avec $\mathcal{B}(b, \sigma) = \mathit{false}$

Premier cas : Simplifier :
$\langle \texttt{if (}b\texttt{) \{}c_1\texttt{;}c_2\texttt{;\} else \{}c_1\texttt{;}c_3\texttt{\}}, \sigma\rangle \to_t \sigma_1$
devient
$\langle c_1\texttt{;}c_2\texttt{;}, \sigma\rangle \to_t \sigma_1$

Preuves d'équivalence de programmes (4)

Il reste à montrer : Si

- $\langle c_1; c_2; , \sigma \rangle \rightarrow_t \sigma_1$ et
- $\langle (c_1; \texttt{if (}b\texttt{) \{}c_2;\texttt{\} else \{}c_3\texttt{\}}), \sigma \rangle \rightarrow_t \sigma_2$ et
- $\mathcal{B}(b, \sigma) = \mathit{true}$

alors $\sigma_1 = \sigma_2$
Est-ce vrai ? ?

Preuves d'équivalence de programmes (4)

Il reste à montrer : Si

- $\langle c_1; c_2;, \sigma\rangle \rightarrow_t \sigma_1$ et
- $\langle (c_1; \texttt{if (}b\texttt{) \{}c_2;\texttt{\} else \{}c_3\texttt{\}}), \sigma\rangle \rightarrow_t \sigma_2$ et
- $\mathcal{B}(b, \sigma) = \mathit{true}$

alors $\sigma_1 = \sigma_2$
Est-ce vrai ? ? Pour compléter la preuve :

- Définir une fonction *vars*(b)
  (ensemble des variables contenues dans l'expression b)
- Définir une fonction *modifs*(c)
  (ensemble des variables modifiées dans l'instruction c)
- Trouver une précondition pour l'équivalance des programmes.
  Laquelle ?

Quelques exercices

- Dériver de nouvelles règles, par exemple pour `do .. while (b)`
- montrer que `do {c1; c2 } while (b)` peut être transformé en `c1; do {c2 } while (b)` si `c2` et `b` ne "dépendent pas" de `c1`.