Le rôle du λ-calcul dans les assistants de preuve

TIPE - Cycles & Boucles

Victor ROBERT (Candidat 38406)

Introduction

La vérification est une nécessité

Problématisation

Comment la vérification d'une preuve se ramène-t-elle à la vérification du typage d'un λ-terme ?

Plan

1. Les λ-termes typés
2. Les arbres de preuve
3. Une bijection entre les deux
4. Implémentation d'un vérificateur en OCaml

Le λ-calcul

Définition (Λ, *"λ-termes"*)

Λ est le plus petit ensemble tel que

- $\forall x \in \mathcal{V},\ x \in \Lambda$
- $\forall (x,t) \in \mathcal{V} \times \Lambda,\ (\lambda x.\ t) \in \Lambda$
- $\forall (s,t) \in \Lambda^2,\ (s\ t) \in \Lambda$

$\equiv_\alpha$: deux termes ne différant que par le nom de leurs variables liées sont identifiés.

Exemples : $\lambda x.\ x \equiv_\alpha \lambda y.\ y$ $\qquad$ $\Omega = (\lambda x.\ x\ x)(\lambda x.\ x\ x)$

Définition ($\rightarrow_\beta$, *"β-réduction"*)

Soit $x \in \mathcal{V}$, $(s,t,u) \in \Lambda^3$.

$\rightarrow_\beta$ est la plus petite relation sur Λ vérifiant

- $((\lambda x.\ t)\ u) \rightarrow_\beta t[x \leftarrow u]$ **(évaluation)**
- **Si** $s \rightarrow_\beta t$ **alors**
  - $(s\ u) \rightarrow_\beta (t\ u),\quad (u\ s) \rightarrow_\beta (u\ t),\quad (\lambda x.\ s) \rightarrow_\beta (\lambda x.\ t)$

La β-réduction est une règle de réécriture

$$(\lambda x.\ t)\ u\ \to_\beta\ t[x \leftarrow u]$$

Application d'une règle jusqu'à une forme stable.

Exemple de $\lambda f.\, \lambda x.\ f\ (f\ x)$:

$$\begin{aligned}(\lambda f.\, \lambda x.\ f\ (f\ x))\ g\ z\ &\to_\beta\ (\lambda x.\ g\ (g\ x))\ z \\ &\to_\beta\ g\ (g\ z) \qquad \text{(forme normale)}\end{aligned}$$

Comme la β-réduction, ce système applique des règles de transformation jusqu'à atteindre une forme stable

- matches
- apply et subst

```
$ ocaml moteur.ml

  ((((\f. (\x. (f (f x)))) g) z)
->
  (((\x. (g (g x))) z)
->
  (g (g z))
```

Le λ-calcul simplement typé

Définitions

$\mathcal{T}$ est le plus petit ensemble stable par $+$, $\times$, $\rightarrow$, contenant les types de base et $\bot$.

Un contexte Γ est une liste finie d'associations $(x : A)$ où $(x, A) \in \mathcal{V} \times \mathcal{T}$.

Relation de typage

$\Gamma \vdash (t : A)$ *"t est de type A dans le contexte* Γ*"* est la plus petite relation vérifiant :

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash (x : A)}\ \text{Var} \qquad \frac{\Gamma, (x : A) \vdash (t : B)}{\Gamma \vdash (\lambda(x : A).\ t : A \rightarrow B)}\ \text{Abstr}$$

$$\frac{\Gamma \vdash (t : A \rightarrow B) \quad \Gamma \vdash (u : A)}{\Gamma \vdash (t\ u : B)}\ \text{Appli}$$

Et de même pour les types $\times$, $+$ et $\bot$.

L'isomorphisme de Curry-Howard

L'observation

Déduction naturelle	**λ-calcul simplement typé**
$\dfrac{A \in \Gamma}{\Gamma \vdash A}$ ax	$\dfrac{(x{:}A) \in \Gamma}{\Gamma \vdash (x : A)}$ Var
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to_i$	$\dfrac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda(x{:}A).t : A \to B}$ Abstr
$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B} \to_e$	$\dfrac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$ Appli

À notation près, ces systèmes sont les mêmes.

Logique	**λ-calcul simplement typé**
Proposition A	Type $A \in \mathcal{T}$
Preuve de A	**Terme** de type A
$A \Rightarrow B$	$A \to B$
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$\bot$	type vide
Modus ponens	Application
Élimination des coupures	β-réduction

Vérifier une preuve = vérifier qu'un terme est bien typé

Les arbres de preuve : $\mathcal{P}(\Gamma, A)$

$\mathcal{P}(\Gamma, A)$: arbres de preuves de $A \in \mathcal{T}$ depuis Γ, en exploitant l'analogie syntaxique avec la relation de typage.

$$\forall (x : A) \in \Gamma, \quad \mathsf{ax}(x) \in \mathcal{P}(\Gamma, A)$$

$$\forall \pi \in \mathcal{P}(\Gamma, x{:}A, B), \quad \to_i (x, A, \pi) \in \mathcal{P}(\Gamma, A \to B)$$

$$\forall \pi_1 \in \mathcal{P}(\Gamma, A{\to}B),\ \forall \pi_2 \in \mathcal{P}(\Gamma, A), \quad \to_e (\pi_1, \pi_2) \in \mathcal{P}(\Gamma, B)$$

...

$\mathcal{P}(\Gamma, A)$: constructeurs pour $\times$, $+$, $\bot$

$\forall \pi_1 \in \mathcal{P}(\Gamma, A),\ \forall \pi_2 \in \mathcal{P}(\Gamma, B), \quad \wedge_i(\pi_1, \pi_2) \in \mathcal{P}(\Gamma, A \times B)$

$\forall \pi \in \mathcal{P}(\Gamma, A \times B), \quad \wedge_e^g(\pi) \in \mathcal{P}(\Gamma, A) \quad$ et $\quad \wedge_e^d(\pi) \in \mathcal{P}(\Gamma, B)$

$\forall \pi \in \mathcal{P}(\Gamma, A), \quad \vee_i^g(\pi, B) \in \mathcal{P}(\Gamma, A + B)$

$\forall \pi \in \mathcal{P}(\Gamma, B), \quad \vee_i^d(A, \pi) \in \mathcal{P}(\Gamma, A + B)$

$\forall \pi \in \mathcal{P}(\Gamma, A{+}B),\ \pi_1 \in \mathcal{P}(\Gamma, x{:}A, C),\ \pi_2 \in \mathcal{P}(\Gamma, y{:}B, C),$
$\vee_e(\pi, x, \pi_1, y, \pi_2) \in \mathcal{P}(\Gamma, C)$

$\forall \pi \in \mathcal{P}(\Gamma, \bot), \quad \bot_e(\pi, A) \in \mathcal{P}(\Gamma, A)$

Construction de $\varphi : \mathcal{P}(\Gamma, A) \to \{t \in \Lambda \mid \Gamma \vdash (t : A)\}_{/\equiv_\alpha}$

On remplace chaque constructeur de $\mathcal{P}$ par le constructeur de terme correspondant.

$$\forall x \in \mathcal{V}, \quad \varphi(\mathsf{ax}(x)) = x$$

$$\forall \pi \in \mathcal{P}(\Gamma, x{:}A, B), \quad \varphi(\to_i (x, A, \pi)) = \lambda(x : A).\ \varphi(\pi)$$

$$\forall \pi_1 \in \mathcal{P}(\Gamma, A \to B),\ \forall \pi_2 \in \mathcal{P}(\Gamma, A), \quad \varphi(\to_e (\pi_1, \pi_2)) = \varphi(\pi_1)\ \varphi(\pi_2)$$

...

φ : cas pour $\times$, $+$, $\bot$

$\forall (\pi_1, \pi_2) \in \mathcal{P}(\Gamma, A) \times \mathcal{P}(\Gamma, B), \quad \varphi(\wedge_i(\pi_1, \pi_2)) = (\varphi(\pi_1),\ \varphi(\pi_2))$
$\forall \pi \in \mathcal{P}(\Gamma, A \times B), \quad \varphi(\wedge_e^g(\pi)) = p_1(\varphi(\pi))$ **et** $\varphi(\wedge_e^d(\pi)) = p_2(\varphi(\pi))$
$\forall \pi \in \mathcal{P}(\Gamma, A), \quad \varphi(\vee_i^g(\pi, B)) = i_1(\varphi(\pi))$
$\forall \pi \in \mathcal{P}(\Gamma, B), \quad \varphi(\vee_i^d(A, \pi)) = i_2(\varphi(\pi))$
$\forall \pi \in \mathcal{P}(\Gamma, A + B),\ \forall \pi_1 \in \mathcal{P}(\Gamma, x{:}A, C),\ \forall \pi_2 \in \mathcal{P}(\Gamma, y{:}B, C),$
$\varphi(\vee_e(\pi, x, \pi_1, y, \pi_2)) = \mathsf{cas}(\varphi(\pi),\ x.\varphi(\pi_1),\ y.\varphi(\pi_2))$
$\forall \pi \in \mathcal{P}(\Gamma, \bot), \quad \varphi(\bot_e(\pi, A)) = \mathsf{absurde}(\varphi(\pi))$

Construction de $\psi : \{t \in \Lambda \mid \Gamma \vdash (t : A)\}_{/\equiv_\alpha} \to \mathcal{P}(\Gamma, A)$

Induction sur la dérivation de typage de $\Gamma \vdash (t : A)$:

$\forall x \in \mathcal{V}, \quad \psi(x) = \mathsf{ax}(x)$

$\forall t$ t.q. $\Gamma, (x : A) \vdash (t : B), \quad \psi(\lambda(x : A).\ t) =\!\!\to_i (x, A, \psi(t))$

$\forall (t_1, t_2)$ t.q. $\Gamma \vdash (t_1 : A \to B)$ et $\Gamma \vdash (t_2 : A), \quad \psi(t_1\ t_2) =\!\!\to_e (\psi(t_1), \psi(t_2))$

...

ψ : cas pour $\times$, $+$, $\bot$

$\forall (s,t)$ t.q. $\Gamma \vdash (s : A)$ et $\Gamma \vdash (t : B)$, $\quad \psi((s,t)) = \wedge_i(\psi(s), \psi(t))$
$\forall t$ t.q. $\Gamma \vdash (t : A \times B)$, $\quad \psi(p_1(t)) = \wedge_e^g(\psi(t))$ **et** $\psi(p_2(t)) = \wedge_e^d(\psi(t))$
$\forall t$ t.q. $\Gamma \vdash (t : A)$, $\quad \psi(i_1(t)) = \vee_i^g(\psi(t), B)$
$\forall t$ t.q. $\Gamma \vdash (t : B)$, $\quad \psi(i_2(t)) = \vee_i^d(A, \psi(t))$
$\forall (t, u_1, u_2)$ t.q. $\Gamma \vdash (t : A + B)$, $\Gamma, (x : A) \vdash (u_1 : C)$, $\Gamma, (y : B) \vdash (u_2 : C)$, $\psi(\mathsf{cas}(t,\, x.u_1,\, y.u_2)) = \vee_e(\psi(t), x, \psi(u_1), y, \psi(u_2))$
$\forall t$ t.q. $\Gamma \vdash (t : \bot)$, $\quad \psi(\mathsf{absurde}(t)) = \bot_e(\psi(t), A)$

Théorème de Curry-Howard

Pour tout Γ et tout $A \in \mathcal{T}$:

$$\mathcal{P}(\Gamma, A) \simeq \{t \in \Lambda \mid \Gamma \vdash (t : A)\}_{/\equiv_\alpha}$$

Démonstration. On montre $\psi \circ \varphi = \mathrm{id}$ et $\varphi \circ \psi = \mathrm{id}$ par double induction structurelle.

Cas de base : $\psi(\varphi(\mathsf{ax}(x))) = \psi(x) = \mathsf{ax}(x)$

Cas $\to_i (x, A, \pi)$ (H.I. : $\psi(\varphi(\pi)) = \pi$) :

$$\begin{aligned}\psi(\varphi(\to_i (x, A, \pi))) &= \psi(\lambda(x : A).\, \varphi(\pi)) \\ &= \to_i (x, A, \underbrace{\psi(\varphi(\pi))}_{=\pi}) \\ &= \to_i (x, A, \pi)\end{aligned}$$

Conséquence

À toute preuve correcte on peut associer un λ-terme bien typé.
Le typage est décidable: on peut implémenter un vérificateur de preuve.

Implémentation en OCaml

Les types

Trois types suffisent :

- `type_t` : les propositions/types $\mathcal{T}$ (constructeurs `Simple`, `Impl`)
- `terme_t` : les λ-termes simplement typés Λ
- `arbre_t` : les arbres de preuve $\mathcal{P}(\Gamma, A)$, un constructeur par règle :

$$\texttt{Hypothese (ctx, x, a)} \quad \leftrightarrow \mathsf{ax}(x)$$

$$\texttt{ImplIntr (ctx, x, a, p)} \quad \leftrightarrow \rightarrow_i (x, A, \pi)$$

$$\texttt{ImplElim (ctx, p, q)} \quad \leftrightarrow \rightarrow_e (\pi_1, \pi_2)$$

(les constructeurs $\times$, $+$, $\bot$ s'ajouteraient sur le même modèle)

Le vérificateur : `concl`: implémentation de φ

`concl : arbre_t → type_t`: On parcourt de l'arbre et on vérifie que chaque règle est correctement appliquée, et renvoie la proposition démontrée (sinon, lève une exception) :

```
let rec concl = function
  | Hypothese (ctx, x, a) ->
      if a = ctx_mem x ctx then a else failwith "type incoherent!"
  | ImplElim (_, p, q) -> (
      match concl p, concl q with
      | Impl (a, b), a' when a = a' -> b
      | _ -> failwith "elim ratee!")
  | ImplIntr (_, _, a, p) -> Impl (a, concl p)
```

En parallèle, `construire_terme : arbre_t → terme_t` réalise φ en lisant le λ-terme sur l'arbre.

Recherche automatique de preuve

Par Curry-Howard, prouver A équivaut à vérifier le type A. On construit donc un terme par récurrence dirigée par le but (`construire_terme : int → ctx_t → type_t → terme_t`) :

- cas de base : on cherche dans Γ une hypothèse $h : A_1 \to \cdots \to A_n \to$ but, puis on prouve récursivement chaque A_i (règle $\to_e$)
- cas $A \to B$: Introduction d'une hypothèse $x : A$ et on prouve B (règle $\to_i$)

  Le procédé est correct : tout terme renvoyé est bien typé par construction.

Résultats

```
$ ./prouveur "(P -> Q) -> (Q -> R) -> (P -> R)"
[+] Prouvable : oui
[*] Terme :
  ([x1 : (P -> Q)] .
    ([x2 : (Q -> R)] .
      ([x3 : P] . (x2 (x1 x3)))))        <- la transitivité

$ ./prouveur "P -> Q"
[-] Prouvable : non

$ ./prouveur "((P -> Q) -> P) -> P"
[-] Prouvable : non
```

Conclusion

Bilan

Par l'isomorphisme de Curry-Howard, $\Gamma \vdash A$ est démontrable en déduction naturelle si et seulement si il existe t tel que $\Gamma \vdash (t : A)$.

Annexes

Variables libres (FV)

- $\forall x \in \mathcal{V},\ \mathrm{FV}(x) = \{x\}$
- $\forall (x, t) \in \mathcal{V} \times \Lambda,\ \mathrm{FV}(\lambda x.\ t) = \mathrm{FV}(t) \setminus \{x\}$
- $\forall (s, t) \in \Lambda^2,\ \mathrm{FV}(s\ t) = \mathrm{FV}(s) \cup \mathrm{FV}(t)$

Définition ($\equiv_\alpha$)

Soit $(x, y) \in \mathcal{V}^2$, $(s, t, u, v) \in \Lambda^4$.

- **Si** $y \neq x$ et $y \notin \mathrm{FV}(t)$ **alors** $\lambda x.\ t \equiv_\alpha \lambda y.\ t[x \leftarrow y]$
- **Si** $s \equiv_\alpha t$ **alors** $\lambda x.\ s \equiv_\alpha \lambda x.\ t$
- **Si** $s \equiv_\alpha t$ et $u \equiv_\alpha v$ **alors** $s\ u \equiv_\alpha t\ v$

Propriétés du STLC

Théorème (normalisation forte, Tait 1967)

Tout terme bien typé se β-réduit en un nombre fini d'étapes vers une forme normale unique (Church-Rosser, 1936).

Corollaire

Il n'existe pas de terme clos de type $\bot$: le STLC est cohérent.
$\Omega = (\lambda x.\ x\ x)(\lambda x.\ x\ x)$, boucle infinie paradigmatique, n'est pas typable.

Code OCaml : `matches` et `apply`

`matches pattern target` : tente de faire correspondre un motif à une expression. Renvoie l'environnement de substitution si le motif correspond, liste vide sinon.
`apply env expr` : applique un environnement de substitution à une expression.
`subst rule expr` : applique une règle de réécriture à une expression en utilisant `matches` puis `apply`.
Ces trois fonctions constituent le moteur du système de réécriture, dont la β-réduction est un cas particulier.

```
type var = string
type t = string

(* Deux types simples A et B définis par la grammaire A, B ::= x | A => B *)
type type_t = Simple of t | Impl of type_t * type_t

(* Lambda terme *)
type terme_t =
  | Variable of var
  | Abstraction of var * type_t * terme_t
  | Application of terme_t * terme_t

(* Contexte de typage *)
type ctx_t = (var * type_t) list

(* Un séquent est la donnée d'un contexte de typage, d'un lambda terme et d'un
type simple *)
type sequent_t = ctx_t * terme_t * int

type arbre_t =
  | Hypothese of ctx_t * var * type_t
  | ImplIntr of ctx_t * var * type_t * arbre_t
  | ImplElim of ctx_t * arbre_t * arbre_t

type token =
  | Nom of string (* Nom des variables *)
  | ImplT (* -> *)
  | G (* ( *)
  | D (* ) *)

let rec ctx_mem (x : string) = function
  | [] -> failwith "..."
  | (y, t) :: tl -> if x = y then t else ctx_mem x tl

let rec concl = function
  | Hypothese (ctx, x, a) ->
      if a = ctx_mem x ctx then a else failwith "type incohérent!"
```

```
let rec concl = function
  | Hypothese (ctx, x, a) ->
      if a = ctx_mem x ctx then a else failwith "type incohérent!"
  | ImplElim (_, p, q) -> (
      match (concl p, concl q) with
      | Impl (a, b), a' when a = a' -> b
      | _ -> failwith "elim ratée!")
  | ImplIntr (_, _, a, p) -> Impl (a, concl p)

let rec construire_terme : arbre_t -> terme_t = function
  | Hypothese (_, x, _) -> Variable x
  | ImplIntr (_, x, a, p) -> Abstraction (x, a, construire_terme p)
  | ImplElim (_, p, q) -> Application (construire_terme p, construire_terme q)

and fmt_type : type_t -> string = function
  | Simple x -> x
  | Impl (a, b) -> Printf.sprintf "(%s -> %s)" (fmt_type a) (fmt_type b)

and fmt_terme2 t =
  let rec aux i = function
    | Variable x -> x
    | Abstraction (x, a, t) ->
        let ind_str = ref "" in
        for _ = 0 to i do
          ind_str := !ind_str ^ "  "
        done;
        Printf.sprintf "\n%s(λ[%s : %s] . %s)" !ind_str x (fmt_type a)
          (aux (i + 1) t)
    | Application (s, t) -> Printf.sprintf "(%s %s)" (aux i s) (aux i t)
  in
  aux 0 t

and normaliser : type_t -> type_t list * type_t = function
  | Impl (a, b) ->
      let args, res = normaliser b in
      (a :: args, res)
```

```
and normaliser : type_t -> type_t list * type_t = function
  | Impl (a, b) ->
      let args, res = normaliser b in
      (a :: args, res)
  | t -> ([], t)

exception Pas_prouvable

(* Implémentation des variables de De Brujin *)
let suivant =
  let c = ref 0 in
  fun () ->
    incr c;
    "x" ^ string_of_int !c

let rec construire_terme (d : int) (ctx : ctx_t) : type_t -> terme_t =
  if d < 0 then raise Pas_prouvable;

  function
  | Impl (a, b) ->
      let var = suivant () in
      let corps = construire_terme (d - 1) ((var, a) :: ctx) b in
      (* (x_{n+1} : a) . corps *)
      Abstraction (var, a, corps)
  | Simple _ as but ->
      let rec aux = function
        | [] -> raise Pas_prouvable
        | (var, t) :: tl ->
            let args, t' = normaliser t in
            if t' <> but then aux tl
            else
              List.map (construire_terme (d - 1) ctx) args
              |> List.fold_left
                   (fun acc u -> Application (acc, u))
                   (Variable var)
      in
```

```
let tokenize (s : string) : token list =
  let n = String.length s in
  let is_letter c = ('A' <= c && c <= 'Z') || ('a' <= c && c <= 'z') in
  let is_alphanum c = is_letter c || ('0' <= c && c <= '9') || c = '_' in
  let rec aux i acc =
    if i >= n then List.rev acc
    else
      match s.[i] with
      | ' ' | '\n' | '\t' -> aux (i + 1) acc
      | '(' -> aux (i + 1) (G :: acc)
      | ')' -> aux (i + 1) (D :: acc)
      | '-' ->
          if i + 1 < n && s.[i + 1] = '>' then aux (i + 2) (ImplT :: acc)
          else failwith "tokenize: '-' !"
      | c when is_letter c ->
          let j = ref i in
          while !j < n && is_alphanum s.[!j] do
            incr j
          done;
          let name = String.sub s i (!j - i) in
          aux !j (Nom name :: acc)
      | c -> failwith (Printf.sprintf "caractère inattendu: %c" c)
  in
  aux 0 []

let rec parse_impl ts =
  let lhs, ts = parse_atom ts in
  match ts with
  | ImplT :: tl ->
      let rhs, ts' = parse_impl tl in
      (Impl (lhs, rhs), ts')
  | _ -> (lhs, ts)

and parse_atom = function
  | Nom x :: tl -> (Simple x, tl)
  | G :: tl -> (
      let ty, tl' = parse_impl tl in
```

```
let parse_s =
  match tokenize s |> parse_impl with
  | t, [] -> t
  | _ -> failwith "tokens restants"

let () =
  if Array.length Sys.argv <> 2 then (
    Printf.eprintf "Usage: %s \"formule\"\n" Sys.argv.(0);
    exit 1);

  let input = Sys.argv.(1) in
  let ty = parse input in
  Printf.printf "[+] Formule : %s\n" (fmt_type ty);

  try
    let term = construire_terme 40 [] ty in
    Printf.printf "[+] Prouvable : oui\n";
    Printf.printf "[*] Terme : %s\n" (fmt_terme2 term)
  with
  | Pas_prouvable -> Printf.printf "[-] Prouvable : non\n"
  | Failure msg ->
      Printf.eprintf "[-] Erreur: %s\n" msg;
      exit 1
```

```
type term =
  | Var of string (* variable objet : x, y, g, z, ...        *)
  | Meta of string (* méta-variable de motif : A, B, C       *)
  | Lam of string * term (* abstraction : \x. t                *)
  | App of term * term (* application : (t t)                 *)
  | Op of string * term list (* opérateur algébrique : "+", "*", ...    *)

type regle = { nom : string; lhs : term; rhs : term }
type env = (string * term) list

let rec to_string = function
  | Var x | Meta x -> x
  | Lam (x, b) -> Printf.sprintf "(\\%s. %s)" x (to_string b)
  | App (s, t) -> Printf.sprintf "(%s %s)" (to_string s) (to_string t)
  | Op (f, [ a; b ]) ->
      Printf.sprintf "(%s %s %s)" (to_string a) f (to_string b)
  | Op (f, ts) ->
      Printf.sprintf "%s(%s)" f (String.concat ", " (List.map to_string ts))

let rec matches (env : env) (motif : term) (cible : term) : env option =
  match (motif, cible) with
  | Meta x, t -> (
      match List.assoc_opt x env with
      | None -> Some ((x, t) :: env)
      | Some t' -> if t' = t then Some env else None)
  | Var a, Var b when a = b -> Some env
  | Lam (x, p), Lam (y, q) when x = y -> matches env p q
  | App (p1, p2), App (c1, c2) -> (
      match matches env p1 c1 with
      | Some env' -> matches env' p2 c2
      | None -> None)
  | Op (f, ps), Op (g, cs) when f = g && List.length ps = List.length cs ->
      List.fold_left2
        (fun acc p c -> match acc with Some e -> matches e p c | None -> None)
        (Some env) ps cs
```

moteur.ml

:colorscheme monoglow-light

```
        (fun acc p c -> match acc with Some e -> matches e p c | None -> None)
        (Some env) ps cs
  | _ -> None

let rec apply (env : env) (motif : term) : term =
  match motif with
  | Meta x -> (
      match List.assoc_opt x env with
      | Some t -> t
      | None -> failwith ("méta-variable libre : " ^ x))
  | Var _ as t -> t
  | Lam (x, b) -> Lam (x, apply env b)
  | App (s, t) -> App (apply env s, apply env t)
  | Op (f, ts) -> Op (f, List.map (apply env) ts)

let subst (r : regle) (t : term) : term option =
  match matches [] r.lhs t with
  | Some env -> Some (apply env r.rhs)
  | None -> None

let rec free_vars = function
  | Var x -> [ x ]
  | Meta _ -> []
  | Lam (x, b) -> List.filter (fun y -> y <> x) (free_vars b)
  | App (s, t) -> free_vars s @ free_vars t
  | Op (_, ts) -> List.concat_map free_vars ts

let fresh =
  let c = ref 0 in
  fun base ->
    incr c;
    base ^ "'" ^ string_of_int !c

let rec subst_var (x : string) (u : term) (t : term) : term =
  match t with
```

moteur.ml

```
let beta_racine = function
  | App (Lam (x, body), arg) -> Some (subst_var x arg body)
  | _ -> None

let rec un_pas (regles : regle list) (t : term) : term option =
  match beta_racine t with
  | Some t' -> Some t'
  | None -> (
      let par_regle =
        List.fold_left
          (fun acc r -> match acc with Some _ -> acc | None -> subst r t)
          None regles
      in
      match par_regle with
      | Some t' -> Some t'
      | None -> (
          match t with
          | Var _ | Meta _ -> None
          | Lam (x, b) -> (
              match un_pas regles b with
              | Some b' -> Some (Lam (x, b'))
              | None -> None)
          | App (s, u) -> (
              match un_pas regles s with
              | Some s' -> Some (App (s', u))
              | None -> (
                  match un_pas regles u with
                  | Some u' -> Some (App (s, u'))
                  | None -> None))
          | Op (f, ts) ->
              let rec descend prefixe = function
                | [] -> None
                | x :: reste -> (
                    match un_pas regles x with
```

moteur.ml

:colorscheme monoglow-light

```
let rec normalise (profondeur : int) (regles : regle list) (t : term) : term =
  if profondeur <= 0 then t
  else
    match un_pas regles t with
    | Some t' -> normalise (profondeur - 1) regles t'
    | None -> t

let trace profondeur regles t =
  let rec go profondeur t =
    Printf.printf "    %s\n" (to_string t);
    if profondeur <= 0 then ()
    else
      match un_pas regles t with
      | Some t' ->
          Printf.printf "  ->\n";
          go (profondeur - 1) t'
      | None -> ()
  in
  go profondeur t

let () =
  let facto =
    {
      nom = "factorisation";
      lhs =
        Op
          ( "+",
            [
              Op ("*", [ Meta "A"; Meta "B" ]); Op ("*", [ Meta "A"; Meta "C" ]);
            ] );
      rhs = Op ("*", [ Meta "A"; Op ("+", [ Meta "B"; Meta "C" ]) ]);
    }
  in
  let e1 =
```

moteur.ml

12 fewer lines; before #58 1 second ago

```
        Printf.printf "  ->\n";
        go (profondeur - 1) t'
    | None -> ()
 in
 go profondeur t

let () =
 let facto =
   {
     nom = "factorisation";
     lhs =
       Op
         ( "+",
           [
             Op ("*", [ Meta "A"; Meta "B" ]); Op ("*", [ Meta "A"; Meta "C" ]);
           ] );
     rhs = Op ("*", [ Meta "A"; Op ("+", [ Meta "B"; Meta "C" ]) ]);
   }
 in
 let e1 =
   Op ("+", [ Op ("*", [ Var "x"; Var "y" ]); Op ("*", [ Var "x"; Var "z" ]) ])
 in
 (* Printf.printf "[factorisation]  %s\n  =>  %s\n\n" (to_string e1)
   (to_string (normalise 100 [ facto ] e1)); *)

 let church2 = Lam ("f", Lam ("x", App (Var "f", App (Var "f", Var "x")))) in
 let e2 = App (App (church2, Var "g"), Var "z") in
 trace 100 [] e2;
 Printf.printf "\n"
(*
 let k = Lam ("x", Lam ("y", Var "x")) in
 let e3 = App (k, Var "y") in
 Printf.printf "[capture evitee] %s  =>  %s\n" (to_string e3)
   (to_string (normalise 100 [] e3))*)
```

moteur.ml [+]