TIPE: Typage et vérification formelle dans Système F .

Théo Bessel

Numéro candidat : 10002

Motivation

Les méthodes de preuves assistées par ordinateur :

Motivation

Les méthodes de preuves assistées par ordinateur :

- Permettent de vérifier qu'un programme est bien conforme à une spécification donnée

Motivation

Les méthodes de preuves assistées par ordinateur :

- Permettent de vérifier qu'un programme est bien conforme à une spécification donnée
- Permettent ainsi d'éviter qu'un programme dit critique ait des comportements non prévus

Motivation

Les méthodes de preuves assistées par ordinateur :

- Permettent de vérifier qu'un programme est bien conforme à une spécification donnée
- Permettent ainsi d'éviter qu'un programme dit critique ait des comportements non prévus
- Peuvent également servir à démontrer des théorèmes de mathématiques

1. Quelques exemples de preuves (langage : Agda)

Ex 1 : Logique propositionnelle

```
module example where
open import Relation.Binary.PropositionalEquality

data P : Set where
    p : P
```

1. Quelques exemples de preuves

Ex 1 : Logique propositionnelle

```
module example where
open import Relation.Binary.PropositionalEquality

data P : Set where
    p : P

id : P -> P
id x = x
```

1. Quelques exemples de preuves

Ex 1 : Logique propositionnelle

```
module example where
open import Relation.Binary.PropositionalEquality

data P : Set where
    p : P

id : P -> P
id x = x

id_proof : (x : P) -> id x == x
id_proof p = ?
```

1. Quelques exemples de preuves

Ex 1 : Logique propositionnelle

```
module example where
open import Relation.Binary.PropositionalEquality

data P : Set where
    p : P

id : P -> P
id x = x

id_proof : (x : P) -> id x == x
id_proof p = 0{ }
```

Après avoir fait : C-c C-l

1. Quelques exemples de preuves

Ex 1 : Logique propositionnelle

```
module example where
open import Relation.Binary.PropositionalEquality

data P : Set where
    p : P

id : P -> P
id x = x

id_proof : (x : P) -> id x == x
id_proof p = refl
```

Après avoir fait : C-c C-a

1. Quelques exemples de preuves

Ex 2 : Fonction somme

```
module example2 where

-- ======================================================================
-- Prove some properties on equality
-- ======================================================================

=-stab : (a b : N) -> (c : N) -> a == b -> (a + c) == (b + c)

=-rev-stab : (a b : N) -> (c : N) -> a == b -> (c + a) == (c + b)

=-sym : {a b : N} -> a == b -> b == a

=-trans : (a b c : N) -> a == b -> b == c -> a == c

-- ======================================================================
-- Prove some properties on multiplication
-- ======================================================================

*+-dist : (k n m : N) -> (k * (n + m)) == ((k * n)+(k * m))

*-comm : (a b : N) -> (a * b) == (b * a)
```

1. Quelques exemples de preuves

Ex 2 : Fonction somme

```
-- ======================================================================
-- Prove the correction of sum function defined below
-- ======================================================================

sum : N -> N
sum zero = zero
sum (suc n) = suc n + sum n

sum-proof : (n : N) -> (two * (sum n)) == (n * (suc n))
sum-proof zero = equal
sum-proof (suc n) = (
  =-trans (two * sum (suc n)) (two * ((suc n)+(sum n))) (suc n * suc (suc n))
    equal
    (=-trans (two * (suc n + sum n)) ((two * (suc n))+(two * (sum n))) (suc n * suc (suc n))
      (*+-dist two (suc n) (sum n))
      (=-trans ((two * suc n)+(two * sum n)) ((two * suc n)+(n * (suc n))) ((suc n * suc (suc n)))
        (=-rev-stab (two * sum n) (n * suc n) (two * suc n) (sum-proof n) )
         (=-trans ((two * suc n) + (n * suc n)) (((suc n) * two)+(n * suc n)) (suc n * suc (suc n))
          (=-stab (two * suc n) (suc n * two) (n * suc n) (*-comm two (suc n)))
          (=-trans (((suc n) * two)+(n * suc n)) (((suc n) * two) + ((suc n) * n)) (suc n * suc (suc n))
            (=-rev-stab (n * suc n) (suc n * n) (suc n * two) (*-comm n (suc n)))
            (=-sym (*+-dist (suc n) two n))
          )
        )
      )
    )
  )
```

1. Quelques exemples de preuves

Ex 2 : Fonction somme

$$
\begin{aligned}
2 \cdot (\mathrm{sum}(n+1)) &= 2 \cdot [(n+1) + \mathrm{sum}(n)] && (1)\\
&= 2 \cdot (n+1) + 2 \cdot [\mathrm{sum}(n)] && (2)\\
&= 2 \cdot (n+1) + n \cdot (n+1) && (3)\\
&= (n+1) \cdot 2 + n \cdot (n+1) && (4)\\
&= (n+1) \cdot 2 + (n+1) \cdot n && (5)\\
&= (n+1) \cdot n + (n+1) \cdot 2 && (6)\\
&= (n+1) \cdot (n+2) && (7)
\end{aligned}
$$

2. Un outil central : le λ-calcul

Le λ-calcul se base sur l'évaluation et la réduction d'expressions appelées λ-expressions.

2. Un outil central : le λ-calcul

Le λ-calcul se base sur l'évaluation et la réduction d'expressions appelées λ-expressions.

Elles sont définies par induction comme suit :

2. Un outil central : le λ-calcul

Le λ-calcul se base sur l'évaluation et la réduction d'expressions appelées λ-expressions.

Elles sont définies par induction comme suit :

- Les variables notées $a, b, c, ..., z$ sont des λ-expressions

2. Un outil central : le λ-calcul

Le λ-calcul se base sur l'évaluation et la réduction d'expressions appelées λ-expressions.

Elles sont définies par induction comme suit :

- Les variables notées $a, b, c, ..., z$ sont des λ-expressions
- Une abstraction $\lambda x.u$ où x est une variable et u une λ-expression est une λ-expression

2. Un outil central : le λ-calcul

Le λ-calcul se base sur l'évaluation et la réduction d'expressions appelées λ-expressions.

Elles sont définies par induction comme suit :

- Les variables notées $a, b, c, ..., z$ sont des λ-expressions
- Une abstraction $\lambda x.u$ où x est une variable et u une λ-expression est une λ-expression
- Une application d'une λ-expression à une autre est une λ-expression, on note $u\ v$ ou $(u\ v)$

2. Un outil central : le λ-calcul

Avec la syntaxe d'OCaml :

2. Un outil central : le λ-calcul

Avec la syntaxe d'OCaml :

```
type expr =
    | Var of string
    | Abs of { param : string; term : expr }
    | App of { func : expr; arg : expr };;
```

3. La correspondance de Curry-Howard

- Assure une équivalence entre système de preuve et de calcul

3. La correspondance de Curry-Howard

- Assure une équivalence entre système de preuve et de calcul
- Pour le lambda calcul, cela se traduit de la manière suivante :

3. La correspondance de Curry-Howard

- Assure une équivalence entre système de preuve et de calcul

- Pour le lambda calcul, cela se traduit de la manière suivante :

Lambda calcul	**Logique**
Variable x	$\dfrac{}{\Gamma \vdash x}\text{ax}$

3. La correspondance de Curry-Howard

- Assure une équivalence entre système de preuve et de calcul
- Pour le lambda calcul, cela se traduit de la manière suivante :

Lambda calcul	**Logique**
Variable x	$\dfrac{}{\Gamma \vdash x}\,\mathrm{ax}$
Abstraction $\lambda x.u$	$\dfrac{\Gamma, x \vdash u}{\Gamma \vdash x \to u}\,\to_{\mathrm{i}}$

3. La correspondance de Curry-Howard

- Assure une équivalence entre système de preuve et de calcul

- Pour le lambda calcul, cela se traduit de la manière suivante :

Lambda calcul	**Logique**
Variable x	$\dfrac{}{\Gamma \vdash x}\,\mathrm{ax}$
Abstraction $\lambda x.u$	$\dfrac{\Gamma, x \vdash u}{\Gamma \vdash x \to u}\,\to_{\mathrm{i}}$
Application $(u\ v)$	$\dfrac{\Gamma \vdash u \to v \quad \Gamma \vdash u}{\Gamma \vdash v}\,\to_{\mathrm{e}}$

3. La correspondance de Curry-Howard

Chaque système de calcul est équivalent à un système formel plus ou moins complet et plus ou moins cohérent.

3. La correspondance de Curry-Howard

Chaque système de calcul est équivalent à un système formel plus ou moins complet et plus ou moins cohérent.

Nom	**Lambda calcul**	**Logique**
STLC	Lambda calcul simplement typé	Déduction naturelle

3. La correspondance de Curry-Howard

Chaque système de calcul est équivalent à un système formel plus ou moins complet et plus ou moins cohérent.

Nom	**Lambda calcul**	**Logique**
STLC	Lambda calcul simplement typé	Déduction naturelle
System F	Lambda calcul avec polymorphisme	Logique intuitionniste du second ordre

3. La correspondance de Curry-Howard

Chaque système de calcul est équivalent à un système formel plus ou moins complet et plus ou moins cohérent.

Nom	**Lambda calcul**	**Logique**
STLC	Lambda calcul simplement typé	Déduction naturelle
System F	Lambda calcul avec polymorphisme	Logique intuitionniste du second ordre
System F-ω	Lambda calcul avec types dépendants	Logique intuitionniste d'ordre supérieur

4. Le λ-calcul simplement typé, un modèle plus complexe

On va vouloir typer les λ-expressions

4. Le λ-calcul simplement typé, un modèle plus complexe

On va vouloir typer les λ-expressions

Le typage se fait suivant quatre règles d'inférences :

4. Le λ-calcul simplement typé, un modèle plus complexe

On va vouloir typer les λ-expressions

Le typage se fait suivant quatre règles d'inférences :

$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\,\mathrm{var}$	

4. Le λ-calcul simplement typé, un modèle plus complexe

On va vouloir typer les λ-expressions

Le typage se fait suivant quatre règles d'inférences :

$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\,\text{var}$	$\dfrac{c \text{ est de type primitif } \tau}{\Gamma \vdash c : \tau}\,\text{cst}$

4. Le λ-calcul simplement typé, un modèle plus complexe

On va vouloir typer les λ-expressions

Le typage se fait suivant quatre règles d'inférences :

$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\,\mathrm{var}$	$\dfrac{c \text{ est de type primitif } \tau}{\Gamma \vdash c : \tau}\,\mathrm{cst}$
$\dfrac{\Gamma, x : \tau_1 \vdash y : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . y) : (\tau_1 \to \tau_2)}\,\mathrm{abs}$	

4. Le λ-calcul simplement typé, un modèle plus complexe

On va vouloir typer les λ-expressions

Le typage se fait suivant quatre règles d'inférences :

$\dfrac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\,\text{var}$	$\dfrac{c \text{ est de type primitif } \tau}{\Gamma \vdash c : \tau}\,\text{cst}$
$\dfrac{\Gamma, x : \tau_1 \vdash y : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 . y) : (\tau_1 \to \tau_2)}\,\text{abs}$	$\dfrac{\Gamma \vdash f : \tau_1 \to \tau_2 \quad \Gamma \vdash x : \tau_1}{\Gamma \vdash (f\ x) : \tau_2}\,\text{app}$

5. Implémentation en langage OCaml

- Un parseur pour la syntaxe du λ-calcul simplement typé :

```
let id = \x.x;
let perm = \x:A.\y:B.x;
let modus = \f:(A -> B).\x:A.(f x);
let trans = \f:(B -> C).\g:(A -> B).\x:A.(f (g x));
EOF
```

5. Implémentation en langage OCaml

- Un parseur pour la syntaxe du λ-calcul simplement typé :

```
let id = \x.x;
let perm = \x:A.\y:B.x;
let modus = \f:(A -> B).\x:A.(f x);
let trans = \f:(B -> C).\g:(A -> B).\x:A.(f (g x));
EOF
```

- Un interpréteur pour ce langage permettant de retourner des valeurs et d'implémenter des fonctions primitives

5. Implémentation en langage OCaml

- Un parseur pour la syntaxe du λ-calcul simplement typé :

```
let id = \x.x;
let perm = \x:A.\y:B.x;
let modus = \f:(A -> B).\x:A.(f x);
let trans = \f:(B -> C).\g:(A -> B).\x:A.(f (g x));
EOF
```

- Un interpréteur pour ce langage permettant de retourner des valeurs et d'implémenter des fonctions primitives

- Un vérificateur de type permettant également l'inférence automatique

5. Implémentation en langage OCaml

Le parseur utilise des tables de transition, son fonctionnement est celui d'un automate à pile :

```
module Token = struct
    type token =
        | ID_NAME of string     (* foo *)
        | INT of int            (* 43 *)
        | LAMBDA                (* \ *)
        | COLON                 (* : *)
        | DOT                   (* . *)
        | ARROW                 (* -> *)
        | LPAR                  (* ( *)
        | RPAR                  (* ) *)
        | LET                   (* let *)
        | EQUAL                 (* = *)
        | SEMICOLON             (* ; *)
        | EOF;;                 (* EOF *)
end;;
```

On a également rajouté des paramètres de type aux λ-termes

5. Implémentation en langage OCaml

Les règles d'inférence de type sont implémentées dans une fonction qui utilise un contexte de typage :

```
let rec infer_type (env : t_expr Env.t) (expr : expr) : t_expr =
  match expr with
    (* -- Sequent 1 : -- *)
    | Var x -> begin match Env.find_opt x env with
      | Some t_expr -> t_expr
      | None -> failwith "[Type error] : variable not typed in context" end
    (* -- Sequent 2 : -- *)
    | Int _ -> T_Int
    (* -- Sequent 3 : -- *)
    | Abs { param; t_param; term } -> begin match t_param with
      | Some t_param -> let env = Env.add param t_param env in
        let t_term = infer_type env term in
        T_Arrow { t_param = t_param; t_term = t_term }
      | None -> let aux p = begin match Env.find_opt p env with
        | Some t_param -> t_param
        | None -> failwith "[Type error] : variable not typed in context" end
      in let t_param = aux param in
      let t_term = infer_type env term in
      T_Arrow { t_param = t_param; t_term = t_term } end
    (* -- Sequent 4 : -- *)
    | App { func; arg } -> begin let t_func = infer_type env func in
      let t_arg = infer_type env arg in match t_func with
        | T_Int -> failwith "[Type error] : T_Int is not a valid function type"
        | T_Arrow { t_param; t_term } when Type.equal t_param t_arg -> t_term
        | _ -> failwith "[Type error] : this application type is not a valid function type"
      end;;
```

6. Polymorphisme paramétrique et Système F

Pour mettre en place le polymorphisme paramétrique on introduit une abstraction de type Λ et on définit deux règles de typage :

6. Polymorphisme paramétrique et Système F

Pour mettre en place le polymorphisme paramétrique on introduit une abstraction de type Λ et on définit deux règles de typage :

Règles d'inférence de système F

$$\frac{\Gamma \vdash u : \tau}{\Gamma \vdash \Lambda\alpha.u : \forall\alpha.\tau}\ \forall_{\mathrm{i}}$$

6. Polymorphisme paramétrique et Système F

Pour mettre en place le polymorphisme paramétrique on introduit une abstraction de type Λ et on définit deux règles de typage :

Règles d'inférence de système F

$$\frac{\Gamma \vdash u : \tau}{\Gamma \vdash \Lambda\alpha.u : \forall\alpha.\tau}\ \forall_{\mathrm{i}}$$

$$\frac{\Gamma \vdash u : \forall\alpha.\tau}{\Gamma \vdash u\ [\sigma] : \tau[\alpha/\sigma]}\ \forall_{\mathrm{e}}$$

6. Polymorphisme paramétrique et Système F

Cela permet ainsi de définir des types plus complexes, par exemple :

6. Polymorphisme paramétrique et Système F

Cela permet ainsi de définir des types plus complexes, par exemple :

$$\mathrm{Bool} = \forall\tau.\ (\tau \to \tau \to \tau)$$

6. Polymorphisme paramétrique et Système F

Cela permet ainsi de définir des types plus complexes, par exemple :

$$\mathrm{Bool} = \forall \tau.\ (\tau \to \tau \to \tau)$$

Il y a exactement deux objets de ce type dans système F :

$$\Lambda \tau.\ \lambda x : \tau.\ \lambda y : \tau.\ x$$

et

$$\Lambda \tau.\ \lambda x : \tau.\ \lambda y : \tau.\ y$$

qu'on appelle donc True et False

6. Polymorphisme paramétrique et Système F

On mène le même raisonnement pour définir les entiers naturels :

6. Polymorphisme paramétrique et Système F

On mène le même raisonnement pour définir les entiers naturels :

$$\mathrm{Nat} = \forall \tau.\ (\tau \to (\tau \to \tau) \to \tau)$$

6. Polymorphisme paramétrique et Système F

On mène le même raisonnement pour définir les entiers naturels :

$$\mathrm{Nat} = \forall \tau.\ (\tau \to (\tau \to \tau) \to \tau)$$

Les objets de ce type dans système F sont alors de la forme :

$$\Lambda \tau.\ \lambda x : \tau.\ \lambda f : (\tau - > \tau).\ (f^n\ x)$$

en notant f^n l'itérée de f et cette écriture encode alors l'entier naturel n

6. Polymorphisme paramétrique et Système F

Cela permet également de définir des types dépendant d'autres types, par exemple :

6. Polymorphisme paramétrique et Système F

Cela permet également de définir des types dépendant d'autres types, par exemple :

$$\text{List } A = \forall A.\ \forall \tau.\ \tau \to (A \to \tau \to \tau) \to \tau$$

6. Polymorphisme paramétrique et Système F

Cela permet également de définir des types dépendant d'autres types, par exemple :

$$\mathrm{List}\ A = \forall A.\ \forall \tau.\ \tau \to (A \to \tau \to \tau) \to \tau$$

Les objets de ce type dans système F sont alors

$$\Lambda A.\ \Lambda \tau.\ \lambda x : A.\ \lambda f : (A \to \tau \to \tau).\ x$$

qui représente la liste vide appelée "Nil" et

6. Polymorphisme paramétrique et Système F

Cela permet également de définir des types dépendant d'autres types, par exemple :

$$\mathrm{List}\ A = \forall A.\ \forall \tau.\ \tau \to (A \to \tau \to \tau) \to \tau$$

Les objets de ce type dans système F sont alors

$$\Lambda A.\ \Lambda \tau.\ \lambda x : A.\ \lambda f : (A \to \tau \to \tau).\ x$$

qui représente la liste vide appelée $''\mathrm{Nil}''$ et

$$\Lambda A.\ \Lambda \tau.\ \lambda h : A.\ t : (\mathrm{List}\ A).\ \lambda x : A.\ \lambda f : (A \to \tau \to \tau).\ f\ h\,(t[\tau]\ f\ x)$$

qui est un constructeur de liste appelé $''\mathrm{Cons}''$

6. Polymorphisme paramétrique et Système F

Liste des définitions

```
let id = \x.x;

let perm = \x:A.\y:B.x;

let modus = \f:(A -> B).\x:A.(f x);

let trans = \f:(B -> C).\g:(A -> B).\x:A.(f (g x));


let id_abs = @T.\x:T.x;

let id_abs_eval = @T.\x:T.x [A];


let true = @T.\x:T.\y:T.x;

let false = @T.\x:T.\y:T.x;

let empty_list = @A.@T.\x:A.\f:(A -> T -> T).x;

EOF
```

6. Polymorphisme paramétrique et Système F

Types inférés par l'algorithme

```
id : T -> T
- : unit = ()
perm : A -> B -> A
- : unit = ()
modus : (A -> B) -> A -> B
- : unit = ()
trans : (B -> C) -> (A -> B) -> A -> C
- : unit = ()
id_abs : %T (T -> T)
- : unit = ()
id_abs_eval : A -> A
- : unit = ()
true : %T (T -> T -> T)
- : unit = ()
false : %T (T -> T -> T)
- : unit = ()
empty_list : %A (%T (A -> (A -> T -> T) -> A))
- : unit = ()
```

7. Extension à Système F-ω

Pour étendre encore notre système de calcul à Système F-ω, on ajoute un système de types dépendants :

7. Extension à Système F-ω

Pour étendre encore notre système de calcul à Système F-ω, on ajoute un système de types dépendants :

En plus d'autoriser à nos types de dépendre d'autres types, on autorise qu'ils dépendent également de valeurs

7. Extension à Système F-ω

Pour étendre encore notre système de calcul à Système F-ω, on ajoute un système de types dépendants :

En plus d'autoriser à nos types de dépendre d'autres types, on autorise qu'ils dépendent également de valeurs

Exemple des vecteurs :

7. Extension à Système F-ω

Pour étendre encore notre système de calcul à Système F-ω, on ajoute un système de types dépendants :

En plus d'autoriser à nos types de dépendre d'autres types, on autorise qu'ils dépendent également de valeurs

Exemple des vecteurs :

- Si une liste est un type polymorphe $\mathrm{List}\ A$

7. Extension à Système F-ω

Pour étendre encore notre système de calcul à Système F-ω, on ajoute un système de types dépendants :

En plus d'autoriser à nos types de dépendre d'autres types, on autorise qu'ils dépendent également de valeurs

Exemple des vecteurs :

- Si une liste est un type polymorphe $\mathrm{List}\ A$
- Un vecteur est un type dépendant $\mathrm{Vec}\ A\ n$

7. Extension à Système F-ω

On dispose alors d'un système de calcul :

7. Extension à Système F-ω

On dispose alors d'un système de calcul :

- Très expressif : on peut définir les types de données usuels à partir des λ-expressions

7. Extension à Système F-ω

On dispose alors d'un système de calcul :

- Très expressif : on peut définir les types de données usuels à partir des λ-expressions
- Permettant d'écrire des programmes d'une grande diversité

7. Extension à Système F-ω

On dispose alors d'un système de calcul :

- Très expressif : on peut définir les types de données usuels à partir des λ-expressions
- Permettant d'écrire des programmes d'une grande diversité
- Permettant de formaliser et de vérifier ces programmes

7. Extension à Système F-ω

On dispose alors d'un système de calcul :

- Très expressif : on peut définir les types de données usuels à partir des λ-expressions
- Permettant d'écrire des programmes d'une grande diversité
- Permettant de formaliser et de vérifier ces programmes
- Permettant de faire de la démonstration de théorèmes mathématiques

8. Annexe

parser.ml :

```
module Env = Map.Make (String);;

(* Abstract syntax tree *)
module Type = struct (* To define types *)
  type t_expr =
    | T_Int
    | T_Var of string
    | T_Arrow of { t_param : t_expr; t_term : t_expr }
    | T_Forall of { t_param : string; t_ret : t_expr };;

  let rec subst (t : t_expr) (t1 : string) (t2 : t_expr) : t_expr =
    match t with
      | T_Arrow { t_param; t_term } -> T_Arrow {
        t_param = subst t_param t1 t2;
        t_term = subst t_term t1 t2
        }
      | T_Int -> T_Int
      | T_Var name when name = t1 -> t2
      | T_Var name -> T_Var name
      | T_Forall { t_param=t_param; t_ret=t_ret } when t_param = t1 ->
       T_Forall { t_param=t_param; t_ret=t_ret }
      | T_Forall { t_param=t_param; t_ret=t_ret } ->
       T_Forall { t_param=t_param; t_ret=(subst t_ret t1 t2) };;

  let rec equal (t1 : t_expr) (t2 : t_expr) : bool =
    match (t1, t2) with
      | T_Int, T_Int -> true
      | T_Var a, T_Var b -> a = b
      | T_Arrow { t_param=t_param1; t_term=t_term1 },
        T_Arrow { t_param=t_param2; t_term=t_term2 } ->
```

8. Annexe

```
        (equal t_param1 t_param2) && (equal t_term1 t_term2)
      | T_Forall { t_param=t_param1; t_ret=t_ret1 },
        T_Forall { t_param=t_param2; t_ret=t_ret2 } ->
        let t_ret2_1 =
          subst t_ret2 t_param2 (T_Var t_param1) in
          equal t_ret1 t_ret2_1
      | _ -> false;;

  let rec print_type (t : t_expr) : unit =
    let rec aux (t : t_expr) : string =
      match t with
        | T_Int -> "int"
        | T_Var s -> s
        | T_Arrow { t_param=p; t_term=q } ->
          let left =
            begin match p with
              | T_Arrow _ -> "(" ^ (aux p) ^ ")"
              | _ -> aux p
            end
          in left ^ " -> " ^ (aux q)
        | T_Forall { t_param=p; t_ret=q } -> "%" ^ p ^ " (" ^ (aux q) ^ ")"
    in Printf.printf "%s\n" (aux t);;
end;;

module Expr = struct (* To define -terms *)
  open Type;;

  type expr =
    | Int of int
    | Var of string
```

8. Annexe

```
      | Abs of { param : string; t_param : t_expr option; term : expr } (* param:t_param.term *)
      | App of { func : expr; arg : expr }                   (* (func arg) *)
      | Ty_Abs of { param : string; term : expr }
      | Ty_App of { func : expr; arg : t_expr };;
  end;;

  module Token = struct
    type token =
      | ID_NAME of string   (* foo *)
      | INT of int        (* 43 *)

      | LAMBDA        (* \ *)
      | BIGLAMBDA       (* @ *)
      | FORALL        (* # *)
      | COLON         (* : *)
      | DOT         (* . *)
      | ARROW         (* -> *)

      | LPAR          (* ( *)
      | RPAR          (* ) *)

      | LBRACK        (* [ *)
      | RBRACK        (* ] *)

      | LET           (* let *)
      | EQUAL         (* = *)
      | SEMICOLON       (* ; *)

      | EOF;;         (* EOF *)
```

8. Annexe

```
let token_to_string token = match token with
  | ID_NAME s   -> "ID_NAME \"" ^ s ^ "\""
  | INT s   -> "INT \"" ^ string_of_int s ^ "\""

  | LAMBDA -> "LAMBDA"
  | BIGLAMBDA -> "BIGLAMBDA"
  | FORALL -> "FORALL"
  | COLON -> "COLON"
  | DOT -> "DOT"
  | ARROW -> "ARROW"

  | LPAR -> "LPAR"
  | RPAR -> "RPAR"

  | LBRACK -> "LBRACK"
  | RBRACK -> "RBRACK"

  | LET -> "LET"
  | EQUAL -> "EQUAL"
  | SEMICOLON -> "SEMICOLON"

  | EOF -> "EOF";;

let rec print_token_list (tokens : token list) : unit =
  match tokens with
    | [] -> Printf.printf "\n\n"
    | h::t -> Printf.printf "%s; " (token_to_string h); print_token_list t;;

let is_alpha c = match c with | 'a'..'z' | 'A'..'Z' -> true | _ -> false;;
let is_digit c = match c with | '0'..'9' -> true | _ -> false;;
```

8. Annexe

```
let lambda = '\\';;
let biglambda = '@';;
let forall = '%';;
let is_lambda c = (c = lambda);;
let is_forall c = (c = forall);;
let is_biglambda c = (c = biglambda);;

let transition_table =
let table = Hashtbl.create 10 in
  Hashtbl.add table 0 (function
    | c when is_lambda c -> 0
    | c when is_biglambda c -> 0
    | c when is_forall c -> 0
    | ' ' | '\n' | ':' | '.' | '(' | ')' | '[' | ']' | '=' | ';' -> 0
    | 'E' -> 4
    | 'l' -> 7
    | c when is_alpha c -> 1
    | c when is_digit c -> 2
    | '-' -> 3
    | _ -> 10
  );
  Hashtbl.add table 1 (function
    | c when is_lambda c -> 0
    | c when is_biglambda c -> 0
    | c when is_forall c -> 0
    | ' ' | '\n' | ':' | '.' | '(' | ')' | '[' | ']' | '=' | ';' -> 0
    | 'E' -> 4
    | 'l' -> 7
    | c when is_alpha c -> 1
```

8. Annexe

```
    | c when is_digit c -> 1
    | '_' -> 1
    | '-' -> 3
    | _ -> 10
  );
  Hashtbl.add table 2 (function
    | c when is_lambda c -> 0
    | c when is_biglambda c -> 0
    | c when is_forall c -> 0
    | ' ' | '\n' | ':' | '.' | '(' | ')' | '[' | ']' | '=' | ';' -> 0
    | 'E' -> 4
    | 'l' -> 7
    | c when is_alpha c -> 1
    | c when is_digit c -> 2
    | '-' -> 3
    | _ -> 10
  );
  Hashtbl.add table 3 (function
    | '>' -> 0
    | _ -> 10
  );
  Hashtbl.add table 4 (function
    | c when is_lambda c -> 0
    | c when is_biglambda c -> 0
    | c when is_forall c -> 0
    | ' ' | '\n' | ':' | '.' | '(' | ')' | '[' | ']' | '=' | ';' -> 0
    | 'E' -> 4
    | 'l' -> 7
    | 'O' -> 5
    | c when is_alpha c -> 1
```

8. Annexe

```
    | c when is_digit c -> 1
    | '-' -> 3
    | _ -> 10
  );
  Hashtbl.add table 5 (function
    | c when is_lambda c -> 0
    | c when is_biglambda c -> 0
    | c when is_forall c -> 0
    | ' ' | '\n' | ':' | '.' | '(' | ')' | '[' | ']' | '=' | ';' -> 0
    | 'E' -> 4
    | 'l' -> 7
    | 'F' -> 6
    | c when is_alpha c -> 1
    | c when is_digit c -> 1
    | '-' -> 3
    | _ -> 10
  );
  Hashtbl.add table 6 (function
    | _ -> 6
  );
  Hashtbl.add table 7 (function
    | c when is_lambda c -> 0
    | c when is_biglambda c -> 0
    | c when is_forall c -> 0
    | ' ' | '\n' | ':' | '.' | '(' | ')' | '[' | ']' | '=' | ';' -> 0
    | 'E' -> 4
    | 'l' -> 7
    | 'e' -> 8
    | c when is_alpha c -> 1
    | c when is_digit c -> 1
```

8. Annexe

```
      | '-' -> 3
      | _ -> 10
    );
    Hashtbl.add table 8 (function
      | c when is_lambda c -> 0
      | c when is_biglambda c -> 0
      | c when is_forall c -> 0
      | ' ' | '\n' | ':' | '.' | '(' | ')' | '[' | ']' | '=' | ';' -> 0
      | 'E' -> 4
      | 'l' -> 7
      | 't' -> 9
      | c when is_alpha c -> 1
      | c when is_digit c -> 1
      | '-' -> 3
      | _ -> 10
    );
    Hashtbl.add table 9 (function
      | c when is_lambda c -> 0
      | c when is_biglambda c -> 0
      | c when is_forall c -> 0
      | ' ' | '\n' | ':' | '.' | '(' | ')' | '[' | ']' | '=' | ';' -> 0
      | c when is_alpha c -> 1
      | c when is_digit c -> 1
      | _ -> 10
    );

    Hashtbl.add table 10 (function | _ -> 10);              (* Trap state *)
  table;;

let tokenize (str : string) : token list =
```

8. Annexe

```
let n = String.length str in
let state = ref 0 in
let buffer = ref "" in
let tokens = ref [] in
let i = ref 0 in
while !i < n do
  let c = str.[!i] in
  let nstate = try
    Hashtbl.find transition_table !state c
  with Not_found -> 10 in
  begin match nstate with
    | 0 | 3 | 9 -> begin match !state with
      | 0 -> begin
        match c with
          | c when is_lambda c -> tokens := LAMBDA::(!tokens)
          | c when is_biglambda c -> tokens := BIGLAMBDA::(!tokens)
          | c when is_forall c -> tokens := FORALL::(!tokens)
          | ':' -> tokens := COLON::(!tokens)
          | '.' -> tokens := DOT::(!tokens)
          | '(' -> tokens := LPAR::(!tokens)
          | ')' -> tokens := RPAR::(!tokens)
          | '[' -> tokens := LBRACK::(!tokens)
          | ']' -> tokens := RBRACK::(!tokens)
          | '=' -> tokens := EQUAL::(!tokens)
          | ';' -> tokens := SEMICOLON::(!tokens)
          | _ -> ()
        end
      | 1 -> if !buffer <> "" then tokens := (ID_NAME !buffer)::(!tokens) else (); begin
        match c with
          | c when is_lambda c -> tokens := LAMBDA::(!tokens)
```

8. Annexe

```
        | c when is_biglambda c -> tokens := BIGLAMBDA::(!tokens)
        | c when is_forall c -> tokens := FORALL::(!tokens)
        | ':' -> tokens := COLON::(!tokens)
        | '.' -> tokens := DOT::(!tokens)
        | '(' -> tokens := LPAR::(!tokens)
        | ')' -> tokens := RPAR::(!tokens)
        | '[' -> tokens := LBRACK::(!tokens)
        | ']' -> tokens := RBRACK::(!tokens)
        | '=' -> tokens := EQUAL::(!tokens)
        | ';' -> tokens := SEMICOLON::(!tokens)
        | _ -> ()
      end
    | 2 -> if !buffer <> "" then tokens := (INT (int_of_string !buffer))::(!tokens)
      else (); begin match c with
        | c when is_lambda c -> tokens := LAMBDA::(!tokens)
        | c when is_biglambda c -> tokens := BIGLAMBDA::(!tokens)
        | c when is_forall c -> tokens := FORALL::(!tokens)
        | ':' -> tokens := COLON::(!tokens)
        | '.' -> tokens := DOT::(!tokens)
        | '(' -> tokens := LPAR::(!tokens)
        | ')' -> tokens := RPAR::(!tokens)
        | '[' -> tokens := LBRACK::(!tokens)
        | ']' -> tokens := RBRACK::(!tokens)
        | '=' -> tokens := EQUAL::(!tokens)
        | ';' -> tokens := SEMICOLON::(!tokens)
        | _ -> ()
      end
    | 3 -> tokens := ARROW::(!tokens); begin match c with
        | c when is_lambda c -> tokens := LAMBDA::(!tokens)
        | c when is_biglambda c -> tokens := BIGLAMBDA::(!tokens)
```

8. Annexe

```
          | c when is_forall c -> tokens := FORALL::(!tokens)
          | ':' -> tokens := COLON::(!tokens)
          | '.' -> tokens := DOT::(!tokens)
          | '(' -> tokens := LPAR::(!tokens)
          | ')' -> tokens := RPAR::(!tokens)
          | '[' -> tokens := LBRACK::(!tokens)
          | ']' -> tokens := RBRACK::(!tokens)
          | '=' -> tokens := EQUAL::(!tokens)
          | ';' -> tokens := SEMICOLON::(!tokens)
          | _ -> ()
        end
      | 4 -> tokens := (ID_NAME !buffer)::(!tokens); begin match c with
          | c when is_lambda c -> tokens := LAMBDA::(!tokens)
          | c when is_biglambda c -> tokens := BIGLAMBDA::(!tokens)
          | c when is_forall c -> tokens := FORALL::(!tokens)
          | ':' -> tokens := COLON::(!tokens)
          | '.' -> tokens := DOT::(!tokens)
          | '(' -> tokens := LPAR::(!tokens)
          | ')' -> tokens := RPAR::(!tokens)
          | '[' -> tokens := LBRACK::(!tokens)
          | ']' -> tokens := RBRACK::(!tokens)
          | '=' -> tokens := EQUAL::(!tokens)
          | ';' -> tokens := SEMICOLON::(!tokens)
          | _ -> ()
        end
      | 5 -> tokens := (ID_NAME !buffer)::(!tokens); begin match c with
          | c when is_lambda c -> tokens := LAMBDA::(!tokens)
          | c when is_biglambda c -> tokens := BIGLAMBDA::(!tokens)
          | c when is_forall c -> tokens := FORALL::(!tokens)
          | ':' -> tokens := COLON::(!tokens)
```

8. Annexe

```
        | '.' -> tokens := DOT::(!tokens)
        | '(' -> tokens := LPAR::(!tokens)
        | ')' -> tokens := RPAR::(!tokens)
        | '[' -> tokens := LBRACK::(!tokens)
        | ']' -> tokens := RBRACK::(!tokens)
        | '=' -> tokens := EQUAL::(!tokens)
        | ';' -> tokens := SEMICOLON::(!tokens)
        | _ -> ()
      end
    | 7 -> tokens := (ID_NAME !buffer)::(!tokens); begin match c with
        | c when is_lambda c -> tokens := LAMBDA::(!tokens)
        | c when is_biglambda c -> tokens := BIGLAMBDA::(!tokens)
        | c when is_forall c -> tokens := FORALL::(!tokens)
        | ':' -> tokens := COLON::(!tokens)
        | '.' -> tokens := DOT::(!tokens)
        | '(' -> tokens := LPAR::(!tokens)
        | ')' -> tokens := RPAR::(!tokens)
        | '[' -> tokens := LBRACK::(!tokens)
        | ']' -> tokens := RBRACK::(!tokens)
        | '=' -> tokens := EQUAL::(!tokens)
        | ';' -> tokens := SEMICOLON::(!tokens)
        | _ -> ()
      end
    | 8 -> if nstate = 9 then () else tokens := (ID_NAME !buffer)::(!tokens);
       begin match c with
        | c when is_lambda c -> tokens := LAMBDA::(!tokens)
        | c when is_biglambda c -> tokens := BIGLAMBDA::(!tokens)
        | c when is_forall c -> tokens := FORALL::(!tokens)
        | ':' -> tokens := COLON::(!tokens)
        | '.' -> tokens := DOT::(!tokens)
```

8. Annexe

```
        | '(' -> tokens := LPAR::(!tokens)
        | ')' -> tokens := RPAR::(!tokens)
        | '[' -> tokens := LBRACK::(!tokens)
        | ']' -> tokens := RBRACK::(!tokens)
        | '=' -> tokens := EQUAL::(!tokens)
        | ';' -> tokens := SEMICOLON::(!tokens)
        | _ -> ()
      end
    | 9 -> tokens := LET::(!tokens); begin match c with
        | c when is_lambda c -> tokens := LAMBDA::(!tokens)
        | c when is_biglambda c -> tokens := BIGLAMBDA::(!tokens)
        | c when is_forall c -> tokens := FORALL::(!tokens)
        | ':' -> tokens := COLON::(!tokens)
        | '.' -> tokens := DOT::(!tokens)
        | '(' -> tokens := LPAR::(!tokens)
        | ')' -> tokens := RPAR::(!tokens)
        | '[' -> tokens := LBRACK::(!tokens)
        | ']' -> tokens := RBRACK::(!tokens)
        | '=' -> tokens := EQUAL::(!tokens)
        | ';' -> tokens := SEMICOLON::(!tokens)
        | _ -> ()
      end
    | _ ->  ()
   end; buffer := ""; state := nstate; incr i
 | 1 | 2 -> buffer := !buffer ^ (String.make 1 c); state := nstate; incr i
 | 4 | 5 | 7 | 8 -> begin match !state with
   | _ -> buffer := !buffer ^ (String.make 1 c); state := nstate; incr i
 end
 | 6 -> tokens := EOF::(!tokens); state := nstate; i := n
 | 10 -> failwith "[Parsing Error] : Syntax error in parsed expression\n"
```

8. Annexe

```
      | _ -> failwith "[Parsing Error] : Invalid state\n"
    end
  done;
  let h::t = !tokens in
  if h = EOF then
    List.rev !tokens
  else
    failwith "[Parsing Error] : End Of File (EOF) not found\n";;
end;;

module Dic = struct
  open Type;;
  open Expr;;
  open Token;;

  type 'a dic = Dic of { keys : string list; values : 'a list};;

  let rec find (d : 'a dic) (k : string) : 'a =
    match d with
      | Dic { keys=[]; values=[] } -> Printf.printf "%s" k;
       failwith "[Error] : key not found in dictionnay"
      | Dic { keys=(h::t); values=(vh::vt) } ->
        if h = k then vh
        else find (Dic { keys=t; values=vt }) k;;

  let get_keys (d : 'a dic) : string list =
    match d with
      | Dic { keys; values } -> keys;;
end;;
```

8. Annexe

```
module Parse = struct
  open Type;;
  open Expr;;
  open Token;;
  open Dic;;

  let rec split (tokens : token list) (e : token) : ((token list) * (token list)) =
    match tokens with
      | [] -> [], []
      | h::t when h=e -> [],t
      | h::t -> let b,a = split t e in h::b,a;;

  let parse (tokens : token list) : (expr dic) =
    let rec parse_app (token : token list) (tokens : token list) : expr =
      let rec aux (tokens : token list) (acc : token list list) : expr =
        match tokens with
          | [] -> begin let rec apply (l : token list list) : expr =
              begin match l with
                | [] -> failwith "[Parsing Error] : Empty application"
                | [y] -> parse_expr y
                | y::rest -> App { func=(apply rest); arg=(parse_expr y) }
              end
            in apply acc end
          | (INT n)::rest -> aux rest ([INT n]::acc)
          | (ID_NAME name)::rest -> aux rest ([ID_NAME name]::acc)
          | LPAR::rest -> let expr, queue = split rest RPAR in
                aux queue (expr::acc)
          | _ -> failwith "[Parsing Error] : code 1"
      in match token with
        | [] -> aux tokens []
```

8. Annexe

```
    | _ -> aux tokens [token]
and parse_type (tokens : token list) : t_expr =
  match tokens with
    | FORALL::(ID_NAME param)::rest -> T_Forall { t_param=param; t_ret=(parse_type rest) }
    | [ID_NAME "int"] -> T_Int
    | [ID_NAME name] -> T_Var name
    | LPAR::rest ->
      let expr, queue = split rest RPAR in
      begin
        match queue with
          | [] -> parse_type expr
          | other -> let rec aux (l : token list) (acc : token list) =
                    match l with
                      | [] -> acc
                      | RPAR::t -> aux t (RPAR::acc)
                      | _ -> failwith "[Parsing Error] : code 2"
                    in parse_type (expr @ (aux other []))
      end
    | rest -> let t_param, t_term = split rest ARROW in
        T_Arrow { t_param=(parse_type t_param); t_term=(parse_type t_term) }
    | _ -> failwith "[Parsing Error] : code 3"
and parse_expr (tokens : token list) : expr =
  let (a,b) = split tokens LBRACK in
  if b = [] then begin
    match tokens with
      | [INT n] -> Int n
      | (INT _)::_ -> failwith "[Parsing Error] : code 4"
      | [ID_NAME name] -> Var name
      | [ID_NAME f; ID_NAME x] -> App { func=(Var f); arg=(Var x) }
      | (ID_NAME _)::_ -> failwith "[Parsing Error] : code 5"
```

8. Annexe

```
        | LPAR::rest ->
          let expr, queue = split rest RPAR in
          begin
            match queue with
              | [] -> parse_app [] expr
              | RPAR::other -> let rec aux (l : token list) (acc : token list) =
                        match l with
                          | [] -> acc
                          | RPAR::t -> aux t (RPAR::acc)
                        in parse_app [] (expr @ (aux other [RPAR]))
              | _ -> parse_app expr queue
          end
        | LAMBDA::(ID_NAME param)::COLON::rest ->
          let t_param, term = split rest DOT in
          Abs { param=param; t_param=Some (parse_type t_param); term=(parse_expr term) }
        | LAMBDA::(ID_NAME param)::DOT::rest ->
          Abs { param=param; t_param=None; term=(parse_expr rest) }
        | BIGLAMBDA::(ID_NAME param)::DOT::rest ->
          Ty_Abs { param=param; term=(parse_expr rest) }
        | _ -> failwith "[Parsing Error] : code 6"
    end else let btype, expr = split b RBRACK in begin match expr with
      | [] -> begin Ty_App { func=(parse_expr a); arg=(parse_type btype) } end
      | _ -> App { func=
        Ty_App { func=(parse_expr a); arg=(parse_type btype) }; arg=(parse_expr expr)
        }
    end in
  let rec parse_defs (tokens : token list) (acc1 : string list) (acc2 : expr list) : (expr dic) =
    match tokens with
      | LET::SEMICOLON::rest -> parse_defs rest acc1 acc2
      | LET::(ID_NAME name)::EQUAL::rest ->
```

8. Annexe

```
          let def, defs = split rest SEMICOLON in
          parse_defs defs (name::acc1) ((parse_expr def)::acc2)
        | EOF::_ -> Dic { keys=(List.rev acc1); values=(List.rev acc2) }
        | _ -> failwith "[Parsing Error] : End Of File (EOF) not found or invalid definition\n"
    in parse_defs tokens [] [];;
end;;
```

8. Annexe

interpreter.ml :

```
#use "parser.ml"

module Value = struct            (* To define interpretater output *)
 open Expr;;

 type value =
  | V_Int of int
  | V_Closure of { term : expr; param : string; env : value Env.t }
  | V_Forall of { term : expr; env : value Env.t }
  | V_Native of (value -> value);;
end;;

module Interpreter = struct            (* For code interpretation *)
 open Type;;
 open Expr;;
 open Value;;

 let rec interpret (env : value Env.t) (expr : expr) : value =
  match expr with
   | Int n -> V_Int n
   | Var x -> Env.find x env
   | Abs { param; t_param; term } -> V_Closure { term; param; env }
   | App { func; arg } -> begin
    let arg = interpret env arg in
    match interpret env func with
     | V_Int _ -> failwith "[Type error] : V_Int is not a function"
     | V_Closure { term; param; env } -> interpret (Env.add param arg env) term
     | V_Native f -> f arg
    end
```

8. Annexe

```
    | Ty_Abs { param; term } -> V_Forall { env; term }
      | Ty_App { func; arg } -> begin
          match interpret env func with
            | V_Forall { term; env } -> interpret env term
            | V_Int _ | V_Closure _ | V_Native _ -> failwith "[Type error] : not a function"
        end;;
end;;
```

8. Annexe

typechecker.ml :

```
#use "parser.ml"
#use "utils.ml"

module Typer = struct            (* For type inference *)
 open Type;;
 open Expr;;
 open Utils;;
 open Token;;
 open Parse;;
 open Dic;;

 (* https://en.wikipedia.org/wiki/Simply_typed_lambda_calculus#Typing_rules *)
 let rec infer_type (env : t_expr Env.t) (expr : expr) : t_expr =
  match expr with
   | Var x ->           (* cf. Sequent 1 *)
    begin
     match Env.find_opt x env with
      | Some t_expr -> t_expr
      | None -> failwith "[Type error] : variable not typed"
    end
   | Int _ -> T_Int          (* cf. Sequent 2 *)
   | Abs { param; t_param; term } ->      (* cf. Sequent 3 *)
    begin
     match t_param with
      | Some t_param -> let env = Env.add param t_param env in
            let t_term = infer_type env term in
            T_Arrow { t_param = t_param; t_term = t_term }
      | None -> let aux p =
           begin
            match Env.find_opt p env with
```

8. Annexe

```
          | Some t_param -> t_param
          | None -> failwith "[Type error] : variable not typed"
        end
        in let t_param = aux param in
        let t_term = infer_type env term in
        T_Arrow { t_param = t_param; t_term = t_term }
    end
   | App { func; arg } ->          (* cf. Sequent 4 *)
    begin let t_func = infer_type env func in
    let t_arg = infer_type env arg in
     match t_func with
      | T_Int -> failwith "[Type error] : T_Int is not a valid function type"
      | T_Arrow { t_param; t_term } when Type.equal t_param t_arg -> t_term
      | _ -> failwith "[Type error] : this application type is not a valid function type"
    end
   | Ty_Abs { param; term } -> let t_ret = infer_type env term in
    T_Forall { t_param=param; t_ret=t_ret }
   | Ty_App { func; arg } -> begin
    let t_func = infer_type env func in
     match t_func with
      | T_Forall { t_param; t_ret } -> subst t_ret t_param arg
      | _ -> failwith "[Type error] : this application type is not a valid function type"
    end;;

let infer (filename : string) (def : string) : t_expr =
 let file = Utils.read_file filename in
 let tokenized_defs = tokenize file in
 let defs_dic = parse tokenized_defs in
 let defs_keys = Dic.get_keys defs_dic in
```

8. Annexe

```
  let rec add_def_to_context (keys : string list) (context : Type.t_expr Env.t) : Type.t_expr Env.t =
   match keys with
    | [] -> context
    | k::_ when k=def -> context
    | k::t -> let t_expr = infer_type context (Dic.find defs_dic k) in
        Env.add k t_expr (add_def_to_context t context) in
  let context = Env.empty |> add_def_to_context defs_keys in
  infer_type context (Dic.find defs_dic def);;

 let infer_print (filename : string) (def : string) : unit =
  print_type (infer filename def);;

 let infer_with_context (filename : string) (def : string) (context : Type.t_expr Env.t) : t_expr =
  let file = Utils.read_file filename in
  let tokenized_defs = tokenize file in
  let defs_dic = parse tokenized_defs in
  let defs_keys = Dic.get_keys defs_dic in
  let rec add_def_to_context (keys : string list) (context : Type.t_expr Env.t) : Type.t_expr Env.t =
   match keys with
    | [] -> context
    | k::_ when k=def -> context
    | k::t -> let t_expr = infer_type context (Dic.find defs_dic k) in
        Env.add k t_expr (add_def_to_context t context) in
  let context = add_def_to_context defs_keys context in
  infer_type context (Dic.find defs_dic def);;

 let infer_print_with_context (filename : string) (def : string) (context : Type.t_expr Env.t) : unit =
  print_type (infer_with_context filename def context);;
end;;
```

8. Annexe

utils.ml :

```
module Utils = struct
 open Str;;

 let read_file (filename : string) : string =
  let ic = open_in filename in
  let n = in_channel_length ic in
  let s = Bytes.create n in
   really_input ic s 0 n;
   close_in ic;
   Bytes.to_string s;;
end;;
```

8. Annexe

examples.ml :

```
#use "interpreter.ml";;
#use "typechecker.ml";;

(* example *)
open Type;;
open Expr;;
open Value;;
open Interpreter;;
open Token;;
open Parse;;
open Dic;;
open Typer;;

let context = Env.empty;;
let context = Env.add "x" (T_Var "T") context;;

print_string "id : "; infer_print_with_context "definitions.sf" "id" context;;
print_string "perm : "; infer_print_with_context "definitions.sf" "perm" context;;
print_string "modus : "; infer_print_with_context "definitions.sf" "modus" context;;
print_string "trans : "; infer_print_with_context "definitions.sf" "trans" context;;

(* System F *)
print_string "id_abs : "; infer_print_with_context "definitions.sf" "id_abs" context;;
print_string "id_abs_eval : "; infer_print_with_context "definitions.sf" "id_abs_eval" context;;
print_string "true : "; infer_print_with_context "definitions.sf" "true" context;;
print_string "false : "; infer_print_with_context "definitions.sf" "false" context;;
print_string "empty_list : "; infer_print_with_context "definitions.sf" "empty_list" context;;
```

8. Annexe

definitions.sf :

```
let id = \x.x;

let perm = \x:A.\y:B.x;

let modus = \f:(A -> B).\x:A.(f x);

let trans = \f:(B -> C).\g:(A -> B).\x:A.(f (g x));


let id_abs = @T.\x:T.x;

let id_abs_eval = @T.\x:T.x [A];


let true = @T.\x:T.\y:T.x;

let false = @T.\x:T.\y:T.x;

let empty_list = @A.@T.\x:A.\f:(A -> T -> T).x;

EOF
```

8. Annexe

example.agda :

```
module Example where

open import Agda.Primitive

-- =====================================================================
-- Define == operator
-- =====================================================================

data Equal {a : Level} {X : Set a} : X -> X -> Set a where
  equal : {x : X} ->  Equal x x

_==_ : _
_==_ = Equal

-- =====================================================================
-- Define natural numbers and opperations
-- =====================================================================

data N : Set where
  zero : N
  suc : N -> N

_+_ : N -> N -> N
zero + n = n
suc m + n = suc (m + n)

_*_ : N -> N -> N
zero * n = zero
suc m * n = (m * n) + n
```

8. Annexe

```
one : N
one = suc zero

two : N
two = suc one

-- ==================================================================
-- Prove some properties on equality
-- ==================================================================

=-refl : (n : N) -> n == n
=-refl n = equal

=-cong : {A B : Set} -> (f : A -> B) -> {x y : A} -> x == y -> f x == f y
=-cong f equal = equal

=-stab : (a b : N) -> (c : N) -> a == b -> (a + c) == (b + c)
=-stab a b c = \ x -> (=-cong (\ x -> (x + c)) x)

=-rev-stab : (a b : N) -> (c : N) -> a == b -> (c + a) == (c +  b)
=-rev-stab a b c  = \x -> (=-cong (\x -> (c + x)) x)

=-sym : {a b : N} -> a == b -> b == a
=-sym equal = =-cong ( \ x -> x) equal

=-trans : (a b c : N) -> a == b -> b == c -> a == c
=-trans zero zero zero p1 p2 = equal
=-trans zero zero (suc c) equal ()
=-trans zero (suc b) c () p2
=-trans (suc a) .(suc a) .(suc a) equal equal = equal
```

8. Annexe

```
-- ==============================================================
-- Prove some properties on addition
-- ==============================================================

+-assoc : (a b c : N) -> ((a + b) + c) == (a + (b + c))
+-assoc zero b c = equal
+-assoc (suc a) b c = =-sym (
  =-trans (suc a + (b + c)) (suc (a + (b + c))) ((suc a + b) + c) equal (
    =-trans (suc (a + (b + c))) (suc ((a + b) + c)) ((suc a + b) + c) (
      =-cong suc (=-sym (+-assoc a b c))
    ) equal
  )
)

+-zero : (n : N) -> (n + zero) == n
+-zero zero = equal
+-zero (suc n) = =-cong suc (+-zero n)

+-suc-rev : (a b : N) -> (a + suc b) == suc (a + b)
+-suc-rev zero b = equal
+-suc-rev (suc a) b = =-cong suc (+-suc-rev a b)

+-comm : (a b : N) -> (a + b) == (b + a)
+-comm a zero  = +-zero a
+-comm a (suc b) = (
  =-trans (a + suc b) (suc (a + b)) ((suc b) + a) (+-suc-rev a b) (
    =-sym (=-cong suc (+-comm b a))
  )
)
```

8. Annexe

```
+-perm : (a b c d : N) -> ((a + b) + (c + d)) == ((a + c) + (b + d))
+-perm zero b c d = (
  =-trans ((zero + b) + (c + d)) (b + (c + d)) ((zero + c) + (b + d)) equal (
    =-trans (b + (c + d)) ((b + c) + d) ((zero + c) + (b + d)) (=-sym (+-assoc b c d)) (
      =-trans ((b + c) + d) ((c + b) + d) ((zero + c) + (b + d)) (
        =-stab (b + c) (c + b) d (+-comm b c)
      ) (
        =-trans ((c + b) + d) (c + (b + d)) ((zero + c) + (b + d)) (+-assoc c b d) equal
      )
    )
  )
)
+-perm (suc a) b c d = (
  =-trans ((suc a + b) + (c + d)) ((suc (a + b)) + (c + d)) ((suc a + c) + (b + d)) equal (
    =-trans (suc (a + b) + (c + d)) (suc ((a + b) + (c + d))) ((suc a + c) + (b + d)) equal (
      =-trans (suc ((a + b) + (c + d))) (suc ((a + c) + (b + d))) ((suc a + c) + (b + d))
      (=-cong suc (+-perm a b c d)) (
        =-sym (
          =-trans ((suc a + c) + (b + d)) ((suc (a + c)) + (b + d)) (suc ((a + c) + (b + d)))
          equal equal
        )
      )
    )
  )
)
```

8. Annexe

```
-- ======================================================================
-- Prove some properties on multiplication
-- ======================================================================

*+-dist : (k n m : N) -> (k * (n + m)) == ((k * n) + (k * m))

*+-dist zero n m = equal
*+-dist (suc k) n m = (
  =-trans (suc k * (n + m)) ((k * (n + m)) + (n + m)) (((suc k * n) + (suc k * m))) equal (=-sym (
    =-trans ((suc k * n) + (suc k * m)) (((k * n) + n) + ((k * m) + m)) ((k * (n + m)) + (n + m))
    equal (
      =-trans (((k * n) + n) + ((k * m) + m)) (((k * n) + (k * m)) + (n + m)) ((k * (n + m)) + (n + m))
      (+-perm (k * n) n (k * m) m) (=-sym (=-stab (k * (n + m)) ((k * n) + (k * m)) (n + m)
      (*+-dist k n m))))
    )
  )
)

*-zero : (n : N) -> (n * zero) == zero
*-zero zero = equal
*-zero (suc n) (
  = =-trans (suc n * zero) ((n * zero) + zero) zero equal (
    =-trans ((n * zero) + zero) (n * zero) zero (+-zero (n * zero)) (*-zero n)
  )
)
```

8. Annexe

```
*-one : (n : N) -> (n * one) == n
*-one zero = equal
*-one (suc n) = (
  =-trans (suc n * one) ((n * one) + one) (suc n) equal (
    =-trans ((n * one) + one) (n + one) (suc n) (
      =-stab (n * one) (n) one (*-one n)
    ) (
      =-trans (n + one) (one + n) (suc n) (+-comm n one) equal
    )
  )
)

*-comm : (a b : N) -> (a * b) == (b * a)

*-comm zero b = (
  =-trans (zero * b) zero (b * zero) equal (=-sym (*-zero b))
)
*-comm (suc a) b = (
  =-trans (suc a * b) ((a * b) + b) (b * suc a) equal (
    =-trans ((a * b) + b) (b + (a * b)) (b * suc a) (+-comm (a * b) b) (
      =-trans (b + (a * b)) (b + (b * a)) (b * suc a) (
        =-rev-stab (a * b) (b * a) b (*-comm  a b)) (
        =-trans (b + (b * a)) ((b * one) + (b * a)) (b * (suc a)) (=-sym (
          =-stab (b * one) (b) (b * a) (*-one b))) (
          =-trans ((b * one) + (b * a)) (b * (one + a)) (b * suc a) (=-sym (*+-dist b one a)) equal
        )
      )
    )
  )
)
```

8. Annexe

```
-- ==================================================================
-- Prove the correction of sum function defined below
-- ==================================================================


sum : N -> N
sum zero = zero
sum (suc n) = suc n + sum n

sum-proof : (n : N) -> (two * (sum n)) == (n * (suc n))
sum-proof zero = equal
sum-proof (suc n) = (
  =-trans (two * sum (suc n)) (two * ((suc n) + (sum n))) (suc n * suc (suc n)) equal (
    =-trans (two * (suc n + sum n)) ((two * (suc n)) + (two * (sum n))) (suc n * suc (suc n))
    (*+-dist two (suc n) (sum n)) (
      =-trans ((two * suc n) + (two * sum n)) ((two * suc n) + (n * (suc n)))
      ((suc n * suc (suc n))) (
        =-rev-stab (two * sum n) (n * suc n) (two * suc n) (sum-proof n)
      )(
        =-trans ((two * suc n) + (n * suc n)) (((suc n) * two) + (n * suc n)) (suc n * suc (suc n)) (
          =-stab (two * suc n) (suc n * two) (n * suc n) (*-comm two (suc n))
        )(
          =-trans (((suc n) * two) + (n * suc n)) (((suc n) * two) + ((suc n) * n))
          (suc n * suc (suc n)) (
            =-rev-stab (n * suc n) (suc n * n) (suc n * two) (*-comm n (suc n))
          ) (=-sym (*+-dist (suc n) two n))
        )
      )
    )
  )
)
```