Optimisation de la vitesse de compilation par la fusion entre inférence de types et analyse syntaxique

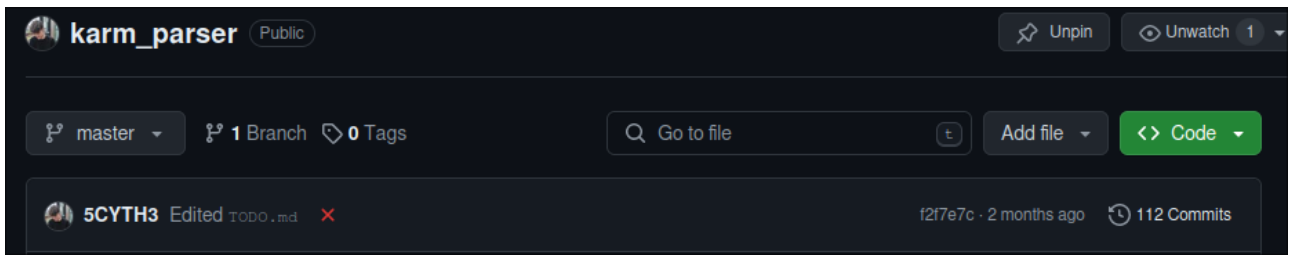
Enogad Le Biavant–Frederic

Alain René Lesage MPI

2025

L'idée

Karm, 2022

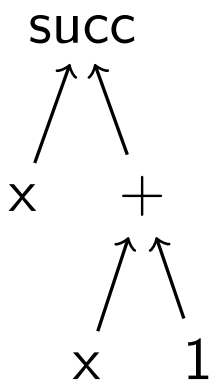


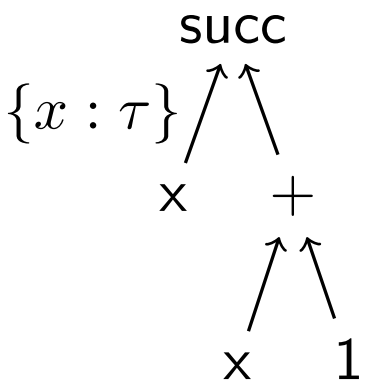
Comment optimiser la vitesse de compilation en fusionnant analyse syntaxique et sémantique ?

L'idée

$\Gamma = \{\texttt{+} : int \to int \to int\}$

$\texttt{let}\ succ = \lambda x.(+\ x\ 1)$

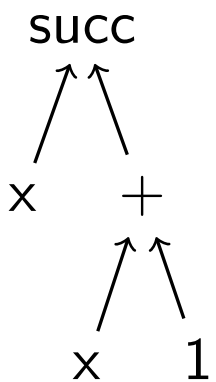


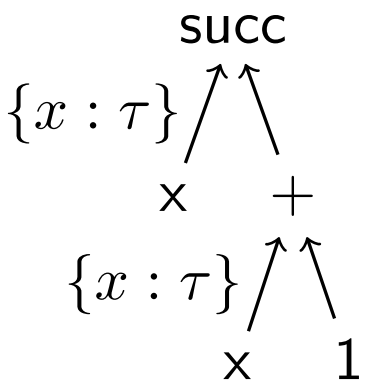


L'idée

$\Gamma = \{\texttt{+} : int \to int \to int,\ \texttt{x} : \tau\}$

$\texttt{let}\ succ = \lambda x.(+\ x\ 1)$

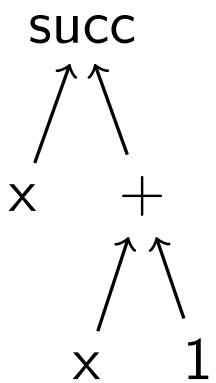


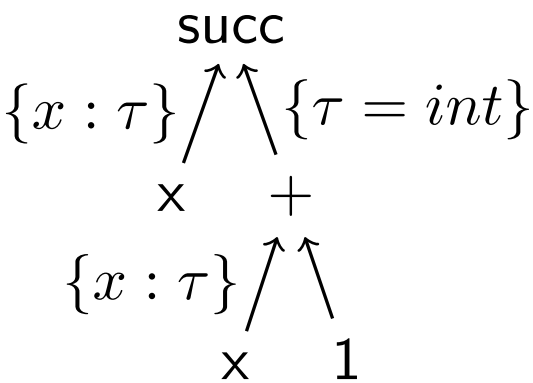


L'idée

$\Gamma = \{\texttt{+} : int \to int \to int,\ \texttt{x} : \tau\}$

$\texttt{let}\ succ = \lambda x.(+\ x\ 1)$

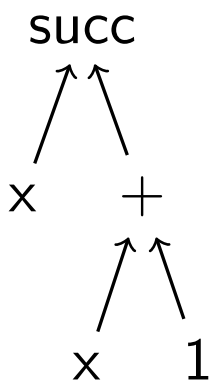


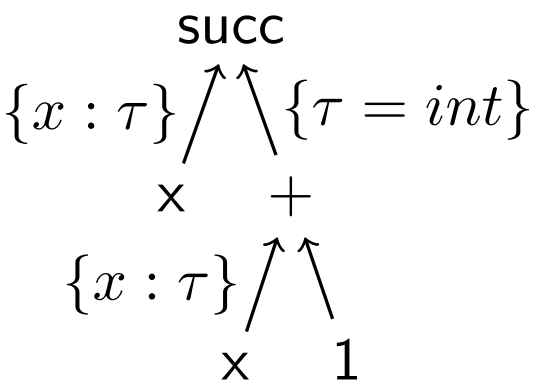


L'idée

$\Gamma = \{\texttt{+} : int \to int \to int,\ \texttt{x} : \tau\}$

$\texttt{let}\ succ = \lambda x.(+\ x\ 1)$





Parsing récursif descendant

Grammaire

$$
\begin{array}{lcl}
program & ::= & expr \\
expr & ::= & abs \\
 & | & app \\
 & | & letbinding \\
app & ::= & term\ [\{\ term\ \}] \\
abs & ::= & \text{"}\backslash\text{"}\ id\ \text{"."}\ expr \\
letbinding & ::= & \text{"let"}\ id\ \text{"="}\ expr\ \text{"in"}\ expr \\
term & ::= & \texttt{string} \\
 & | & \texttt{int} \\
 & | & \texttt{bool} \\
 & | & id \\
 & | & \text{"("}\ expr\ \text{")"}
\end{array}
$$

TT - Définitions

Définition de Type : classification de termes (Church).
Théorie de travail : Lambda calcul simplement typé (LCST), polymorphique.

$$\texttt{let}\ id = \lambda x.x : \forall\sigma \to \sigma$$

Hindley-Milner

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \text{var}$$

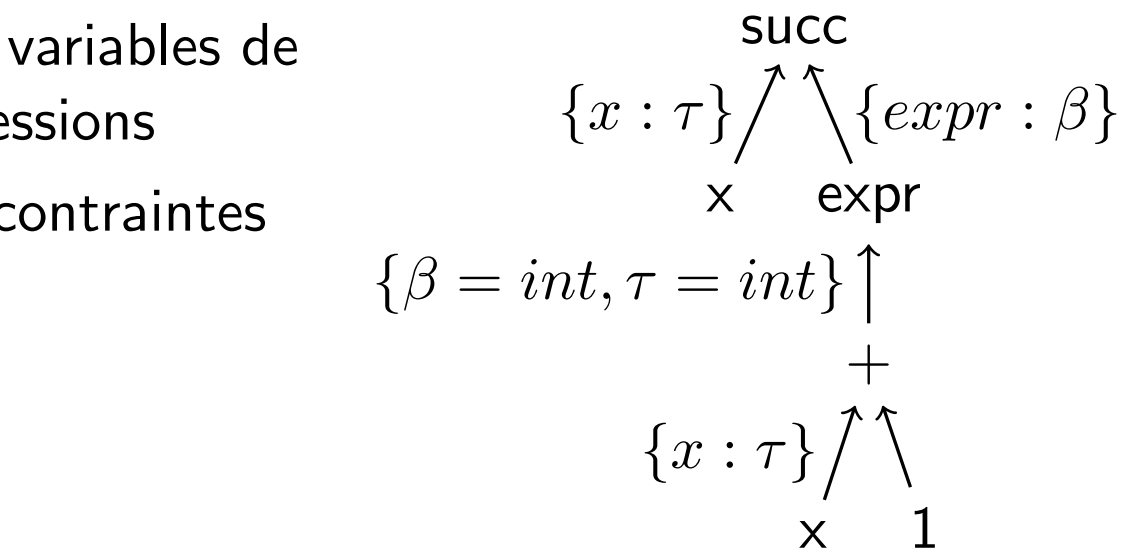
$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\ \text{abs}$$

$$\frac{\Gamma \vdash f : \tau \to \tau' \qquad \Gamma \vdash e : \tau}{\Gamma \vdash f\ e : \tau'}\ \text{app}$$

Hindley-Milner

Algorithme W

1. Assignation de variables de types aux expressions
2. Génération de contraintes
3. Substitutions
4. Unification
5. Instantiation, généralisation



Résultats

$$\mathcal{W} : \tilde{\Gamma} \times Expr \to Subst \times Type$$
$$\mathcal{W}^* : \tilde{\Gamma} \times L \to Subst \times \Gamma \times Expr \times L$$

Machine : i7 5th gen 3.00Ghz

Fichier de test : 1000 premiers nombres de church
Version non optimisée : $\approx 26.37s$
Version optimisée : $\approx 3.88s$

Fichier de test : 10K application de fonction successeur
Version non optimisée : $\approx 1.48s$
Version optimisée : $\approx 1.51s$

Résultats

Conclusion

Bénéfices

Durée inférieure lorsque beaucoup d'unifications

Plus permissif envers les grammaires très imbriquées que les systèmes traditionnels

Formalisation

Automates d'arbres : $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$

- $\mathcal{F}$: Alphabet gradué (fonction d'arité ar)

On peut décrire la grammaire comme une NRTG

[1] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375.

[2] Harry G. Mairson. "Deciding ML typability is complete for deterministic exponential time". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. San Francisco, California, USA: Association for Computing Machinery, 1989, pp. 382–401.

[3] J.B. Wells. "Typability and type checking in the second-order /spl lambda/-calculus are equivalent and undecidable". In: *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*. 1994, pp. 176–185.

[4] Joseph Gil and Ori Roth. *Ties between Parametrically Polymorphic Type Systems and Finite Control Automata*. 2020. arXiv: 2009.04437 [cs.PL].

[5] Donald E. Knuth. "On the translation of languages from left to right". In: *Information and Control* 8.6 (1965), pp. 607–639.

Annexe - Lexemes


```
type literal =
    | Str of string
    | Int of int
    | Bool of bool
    [@@deriving show]

type t =
    | Let
    | Lambda
    | Dot
    | Assign
    | In
    | LParen
    | RParen
    | Id of string
    | Literal of literal
    [@@deriving show]

type program = t list
[@@deriving show]
```

Annexe - Expressions


```
type literal =
    | Str of string
    | Int of int
    | Bool of bool
[@@deriving show]

type t =
    | Var of string
    | Abs of string * t
    | App of t * t
    | Let of string * t * t
    | Literal of literal
[@@deriving show]
```

Annexe - Types


```
type t =
    (* Primitives *)
    | Bool
    | Str
    | Int
    | TVar of string
    (* Function *)
    | Abs of t * t (* @-> *)
    (* Scheme *)
    | Forall of string * t
    [@@deriving show];;

let ( @-> ) t1 t2 = Abs (t1, t2)

let counter = ref 0;;

let fresh_tv () =
    let v = "t" ^ string_of_int !counter in
    incr counter;
    TVar v
let reset_tv_counter () = counter := 0;;
```

Annexe - Types


```
module TypeMap = Map.Make(String)

type subst = t TypeMap.t (* // *)

let rec apply_subst (s: subst) (t: t) =
    match t with
    | TVar v -> begin
        match TypeMap.find_opt v s with (* If type is not found,
            works like Id *)
        | Some x -> x
        | None -> t
    end
    | Abs (arg, ret) -> Abs (apply_subst s arg, apply_subst s
        ret)
    | Forall (v, t) -> Forall (v, apply_subst (TypeMap.remove v
        s) t)
    | t -> t

let ( *&* ) s1 s2 =
    TypeMap.union (fun _ t _ -> Some t) (TypeMap.map
        (apply_subst s1) s2) s1
```

Annexe - Types


```
type env = t TypeMap.t;; (* String : Type *)

let extend_env env k t = TypeMap.add k t env

let apply_env (env: env) k: t =
    try TypeMap.find k env
    with Not_found -> failwith ("Var not in scope : " ^ k)

let apply_subst_to_env subst env = TypeMap.map (fun t ->
     apply_subst subst t) env;;
```

Annexe - Types


```
module Ftv = Set.Make (String);;

type ftvs = Ftv.t

let rec free_tvs t: ftvs =
    match t with
    | Int | Bool | Str -> Ftv.empty
    | TVar v -> Ftv.singleton v
    | Abs (t1, t2) -> Ftv.union (free_tvs t1) (free_tvs t2)
    | Forall (v, t) -> Ftv.remove v (free_tvs t)

let free_tvs_env env =
    TypeMap.fold (fun _ t acc -> Ftv.union (free_tvs t) acc) env
        Ftv.empty
```

Annexe - Types


```
exception TypeError of string

let rec unify t1 t2: subst =
    match t1, t2 with
    | Bool, Bool | Int, Int | Str, Str -> TypeMap.empty
    | TVar v1, TVar v2 when v1 = v2 -> TypeMap.empty
    | TVar v, t | t, TVar v ->
        if Ftv.mem v (free_tvs t) then
            raise (TypeError ("Occurs check failed for variable "
                ^ v))
        else
            TypeMap.singleton v t
    | Abs (t1a, t1b), Abs (t2a, t2b) ->
        let s1 = unify t1a t2a in
        let s2 = unify (apply_subst s1 t1b) (apply_subst s1 t2b)
            in
        s2 *&* s1
    | Forall (v1, t1), Forall (v2, t2) ->
        let t2' = apply_subst (TypeMap.singleton v2 (TVar v1)) t2
            in
        unify t1 t2'
    | _ -> raise (TypeError "Cannot unify types")
```

Annexe - Types


```
let generalize env t =
    let env_fv = free_tvs_env env in
    let ftvs_of_t = free_tvs t in
    let vars_to_generalize = Ftv.filter (fun v -> not (Ftv.mem v
        env_fv)) ftvs_of_t in
    match Ftv.elements vars_to_generalize with
    | [] -> t
    | vars -> List.fold_right (fun v acc -> Forall (v, acc))
        vars t

let rec instantiate t =
    match t with
    | Forall (v, t) -> let fresh_var = fresh_tv () in
        instantiate (apply_subst (TypeMap.singleton v fresh_var)
        t)
    | _ -> t
```

Annexe - Parsing classique


```
type traversal = {
    expr: Expr.t;
    rest: Token.program;
}

let match_literals = function
    | Token.Str s -> Expr.Literal (Expr.Str s)
    | Token.Int i -> Expr.Literal (Expr.Int i)
    | Token.Bool b -> Expr.Literal(Expr.Bool b)

let rec parse_expr (program: program): traversal =
    match program with
    | Lambda :: Id arg :: Dot :: rest ->
        let { expr = body; rest = rest'; } = parse_expr rest in

        {
            expr = Abs (arg, body);
            rest = rest';
        }
    | Let :: _ -> parse_let program
    | _ -> parse_app program
```

Annexe - Parsing classique


```
and parse_let (program: program): traversal =
    match program with
    | Let :: Id i :: Assign :: t ->
        let { expr = body; rest; } = parse_expr t in

        let { expr = in_expr; rest = rest'; } = parse_ins rest in

        {
            expr = Let (i, body, in_expr);
            rest = rest';
        }
    | _ -> failwith "Expected `Let`."
```

Annexe - Parsing classique


```
and parse_ins (program: program): traversal =
    match program with
    | In :: t -> parse_expr t
    | _ -> failwith "No 'in' clause given. "

and parse_parenthesized (program: program): traversal =
    match program with
    | LParen :: t -> begin
        let e = parse_expr t in
        match e.rest with
        | RParen :: t' -> { expr = e.expr; rest = t'; }
        | _ -> failwith "Unclosed parenthesis"
    end
    | _ -> failwith "Expected LParen."
```

Annexe - Parsing classique


```
and parse_app (program: program): traversal =
    let rec aux (lhs: traversal) =
        let { expr = lhs_expr; rest; } = lhs in

        match rest with
        | [] | In :: _ | RParen :: _ -> lhs
        | _ -> begin
            let { expr = rhs_expr; rest = rest'; } = parse_term
                 rest in

            let res = {
                expr = App (lhs_expr, rhs_expr);
                rest = rest';
            } in
            aux res
        end
    in
    let initial = parse_term program in
    aux initial
```

Annexe - Parsing classique


```
and parse_term (program: program): traversal =
    match program with
    | Id x :: rest ->
        {
            expr = Var x;
            rest;
        }
    | Token.Literal l :: t ->
        {
            expr = match_literals l;
            rest = t;
        }
    | LParen :: _ -> parse_parenthesized program
    | RParen :: _ -> failwith "Alone RParen. ?"
    | _ -> failwith "Unexpected token"
```

Annexe - Parsing classique


```
let parse (program: program) =
        let { expr; rest = _; } = parse_expr program in
    expr
;;

let type_of_literal = function
    | Expr.Str _ -> Types.Str
    | Expr.Bool _ -> Types.Bool
    | Expr.Int _ -> Types.Int
;;
```

Annexe - Parsing classique


```
let rec infer_types_aux env e cont =
    let open Types in
    match e with
    | Expr.Literal l ->
        let t = type_of_literal l in
        cont (t, TypeMap.empty)
    | Var x ->
        let t = apply_env env x in
        cont (t, TypeMap.empty)
    | Abs (x, e) ->
        let t = fresh_tv () in
        let env' = extend_env env x t in
        infer_types_aux env' e (fun (t', s) ->
            let t_res = (apply_subst s t) @-> t' in
            cont (t_res, s)
        )
```

Annexe - Parsing classique


```
| App (e1, e2) ->
    infer_types_aux env e1 (fun (t1, s1) ->
        let env' = apply_subst_to_env s1 env in
        infer_types_aux env' e2 (fun (t2, s2) ->
            let tv = fresh_tv () in
            let s3 = unify (apply_subst s2 t1) (t2 @-> tv) in
            let f_subst = s3 *&* s2 *&* s1 in
            cont (apply_subst f_subst tv, f_subst)
        )
    )
| Let (x, e1, e2) ->
    infer_types_aux env e1 (fun (t1, s1) ->
        let t' = generalize env (apply_subst s1 t1) in
        let env' = extend_env env x t' in
        infer_types_aux env' e2 (fun (t2, s2) ->
            cont (t2, s2 *&* s1)
        )
    )
```

Annexe - Parsing classique


```
let infer_types env e =
       infer_types_aux env e (fun res -> res)

let run (p: program) (env: Types.env) =
    let ast = parse p in
    let t, _ = infer_types env ast in
    (ast, t)
;;
```

Annexe - Parsing fusionné


```
type traversal = {
    expr: Expr.t;
    t: Types.t;
    subst: Types.subst;
    rest: Token.t list;
}

let match_literals = function
    | Token.Str s -> Literal (Str s)
    | Token.Int i -> Literal (Int i)
    | Token.Bool b -> Literal(Bool b)

let type_literals = function
    | Token.Str _ -> Types.Str
    | Token.Int _ -> Types.Int
    | Token.Bool _ -> Types.Bool

let fresh_id (id: string) (env: Types.env): Types.t * Types.env =
    let tv = Types.fresh_tv () in
    let env' = Types.extend_env env id tv in
    (tv, env')
```

Annexe - Parsing fusionné


```
let rec parse_expr (program: Token.t list) (env: Types.env):
     traversal =
    match program with
    | Lambda :: Id arg :: Dot :: rest ->
        let (tv, env') = fresh_id arg env in
        let { expr = body; t; subst; rest = rest'; } = parse_expr
             rest env' in

        {
            expr = Abs (arg, body);
            t = Types.Abs (Types.apply_subst subst tv, t);
            subst;
            rest = rest';
        }
    | Let :: _ -> parse_let program env
    | _ -> parse_app program env
```

Annexe - Parsing fusionné


```
and parse_let (program: Token.t list) (env: Types.env):
     traversal =
    match program with
    | Let :: Id i :: Assign :: t ->
        let open Types in
        let { expr = body; t = t1; subst = subst1; rest; } =
             parse_expr t env in
        let t' = generalize env (apply_subst subst1 t1) in (* Get
             the type of the let def *)
        let env' = extend_env env i t' in (* Add the typesig of
             the let-expr to the ctx *)

        let { expr = in_expr; t = t2; subst = subst2; rest =
             rest'; } = parse_ins rest env' in

        {
            expr = Let (i, body, in_expr);
            t = t2;
            subst = subst2 *&* subst1;
            rest = rest';
        }
    | _ -> failwith "Expected `Let`."
```

Annexe - Parsing fusionné


```
and parse_ins (program: Token.t list) (env: Types.env):
     traversal =
    match program with
    | In :: t -> parse_expr t env
    | _ -> failwith "No 'in' clause given. "

and parse_parenthesized (program: Token.t list) (env:
     Types.env): traversal =
    match program with
    | LParen :: t -> begin
        let e = parse_expr t env in
        match e.rest with
        | RParen :: t' -> { expr = e.expr; t = e.t; subst =
             e.subst; rest = t'; }
        | _ -> failwith "Unclosed parenthesis"
    end
    | _ -> failwith "Expected LParen."
```

Annexe - Parsing fusionné


```
and parse_app (program: Token.t list) (env: Types.env):
     traversal =
    let open Types in

    let rec aux (lhs: traversal) =
        let { expr = lhs_expr; t = t1; rest; subst = s1; } = lhs
             in
        match rest with
        | [] | In :: _ | RParen :: _ -> lhs
        | _ -> begin
            let { expr = rhs_expr; t = t2; subst = s2; rest =
                 rest'; } = parse_term rest env in
            let tv = fresh_tv () in
            let s3 = unify (apply_subst s2 t1) (Abs (t2, tv)) in
            let f_subst = s3 *&* s2 *&* s1 in
```

Annexe - Parsing fusionné


```
            let res = {
                expr = App (lhs_expr, rhs_expr);
                t = apply_subst f_subst tv;
                subst = f_subst;
                rest = rest';
             } in
             aux res
        end
    in
    let initial = parse_term program env in
    aux initial
```

Annexe - Parsing fusionné


```
and parse_term (program: Token.t list) (env: Types.env):
     traversal =
    match program with
    | Id x :: rest ->
       let t = Types.apply_env env x in
       {
           expr = Var x;
           t;
           subst = Types.TypeMap.empty;
           rest;
       }
    | Token.Literal l :: t ->
       {
           expr = match_literals l;
           t = type_literals l;
           subst = Types.TypeMap.empty;
           rest = t;
       }
    | LParen :: _ -> parse_parenthesized program env
    | RParen :: _ -> failwith "Alone RParen. ?"
    | _ -> failwith "Unexpected token"
```

Annexe - Parsing fusionné


```
let parse (program: Token.program) (env: Types.env) =
    Types.reset_tv_counter ();
    let { expr; t; subst = _; rest = _; } = parse_expr program
        env in
    (expr, t)
```