Compilation

Typage

Adrien Guatto

Master 1 Informatique
2022–2023

Plan

Plan

Pourquoi le typage ? Retour au premier évaluateur L1

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

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

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

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

Pourquoi le typage ? Retour au premier évaluateur L1

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

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

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

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

Comment s'assurer que ce code ne lève pas `Failure "ill-typed"` ?

Les systèmes de types

Qu'est-ce qu'un système de types ?

- Un jeu de règles syntaxiques pour associer à une expression un type.
- Un type est une petite formule logique qui décrit une propriété.
- Un programme bien typé satisfait la propriété décrite par son type.

Les systèmes de types

Qu'est-ce qu'un système de types ?

- Un jeu de règles syntaxiques pour associer à une expression un type.
- Un type est une petite formule logique qui décrit une propriété.
- Un programme bien typé satisfait la propriété décrite par son type.

Quelques exemples de propriétés que l'on peut garantir par typage :

- OCaml : absence de corruption mémoire ;

Les systèmes de types

Qu'est-ce qu'un système de types ?

- Un jeu de règles syntaxiques pour associer à une expression un type.
- Un type est une petite formule logique qui décrit une propriété.
- Un programme bien typé satisfait la propriété décrite par son type.

Quelques exemples de propriétés que l'on peut garantir par typage :

- OCaml : absence de corruption mémoire ;
- C : données manipulées en accord avec leurs tailles (`sizeof`) ;

Les systèmes de types

Qu'est-ce qu'un système de types ?

- Un jeu de règles syntaxiques pour associer à une expression un type.
- Un type est une petite formule logique qui décrit une propriété.
- Un programme bien typé satisfait la propriété décrite par son type.

Quelques exemples de propriétés que l'on peut garantir par typage :

- OCaml : absence de corruption mémoire ;
- C : données manipulées en accord avec leurs tailles (`sizeof`) ;
- Coq : énoncé mathématique arbitraire (cf. *assistants de preuve*, S2) ;

Les systèmes de types

Qu'est-ce qu'un système de types ?

- Un jeu de règles syntaxiques pour associer à une expression un type.
- Un type est une petite formule logique qui décrit une propriété.
- Un programme bien typé satisfait la propriété décrite par son type.

Quelques exemples de propriétés que l'on peut garantir par typage :

- OCaml : absence de corruption mémoire ;
- C : données manipulées en accord avec leurs tailles (`sizeof`) ;
- Coq : énoncé mathématique arbitraire (cf. *assistants de preuve*, S2) ;

Terminologie douteuse et pop-culture informatique

- "typage dynamique" : on parlera plutôt de langage non-typé.
- "typage fort, typage faible" : à banir, n'a pas de sens précis.

Cette séance de cours

Contenu

- Introduction aux différents concepts des systèmes de types.
  - Sous la forme de code OCaml et de formalisme mathématique.
- Deux systèmes de types et les extensions linguistiques attenantes.
  - Types simples (monomorphes).
  - Types polymorphes.
- Une technique d'implémentation simple, le typage bidirectionnel.

Cette séance de cours

Contenu

- Introduction aux différents concepts des systèmes de types.
  - Sous la forme de code OCaml et de formalisme mathématique.
- Deux systèmes de types et les extensions linguistiques attenantes.
  - Types simples (monomorphes).
  - Types polymorphes.
- Une technique d'implémentation simple, le typage bidirectionnel.

Un point important qui ne sera pas traité

L'inférence de types polymorphes, essentielle aux langages à la ML.

Cette séance de cours

Contenu

- Introduction aux différents concepts des systèmes de types.
  - Sous la forme de code OCaml et de formalisme mathématique.
- Deux systèmes de types et les extensions linguistiques attenantes.
  - Types simples (monomorphes).
  - Types polymorphes.
- Une technique d'implémentation simple, le typage bidirectionnel.

Un point important qui ne sera pas traité

L'inférence de types polymorphes, essentielle aux langages à la ML.

Un objectif pratique : vous préparer à la réalisation du jalon 3.

Plan

Plan

Les types simples

- Une valeur de $\mathrm{L1}$ est un entier ou une fonction.
- Les types classifient les valeurs, donc deux constructeurs de types.

Les types simples

- Une valeur de L1 est un entier ou une fonction.
- Les types classifient les valeurs, donc deux constructeurs de types.

```
type ty = TInt | TFun of ty * ty

module Typechecker : sig
  exception Ill_typed of ...

  (** [synth_exp e] calcule le type de l'expression [e], ou bien
      lève l'exception {! Ill_typed} si [e] est mal typée. *)
  val synth_exp : Lang2.exp -> ty

  (** [check_exp expected e] lève l'exception {! Ill_typed} si
      l'expression [e] n'est pas de type [expected]. *)
  val check_exp : ty -> Lang2.exp -> unit
end
```

Le module `Typechecker`, premier jet

```
exception Ill_typed of ...

let rec synth_exp = function
| Int _ -> TInt
| Add (e1, e2) -> check_exp TInt e1; check_exp TInt e2; TInt
| App (e1, e2) -> app_types (synth_exp e1) e2
| Let (e, b) -> synth_bound1 (synth_exp e) b
| Fun b -> let dom = (* ??? *) in TFun (dom, synth_bound1 dom b)
| Var x -> (* ??? *)

and check_exp expected e =
  let given = synth_exp e in
  if given <> expected then type_mismatch ~expected ~given e

and app_types ty_f e2 = match ty_f with
  | TFun (dom, cod) -> check_exp dom e2; cod
  | given -> type_mismatch_expected_fun given

and synth_bound1 : ty -> bound1 -> ty = fun bound_ty b -> (* ??? *)
```

Gestion des variables

- Une expression est typée dans un contexte de typage.
- On suppose les paramètres de fonctions annotés avec leurs types.

Gestion des variables

- Une expression est typée dans un contexte de typage.
- On suppose les paramètres de fonctions annotés avec leurs types.

```
type exp = ... | Fun of ty * bound1

module Typechecker : sig
  type cx = (Id.t * ty) list

  (** [synth_exp cx e] calcule le type de l'expression [e] en
      supposant que ses variables libres ont les types
      prescrits par le contexte de typage [cx]. Elle lève
      l'exception {! Ill_typed} si [e] est mal typée dans [cx]. *)
  val synth_exp : cx -> Lang2.exp -> ty

  (** [check_exp cx expected e] lève l'exception {! Ill_typed}
      si [e] n'est pas de type [expected] dans [cx]. *)
  val check_exp : cx -> ty -> Lang2.exp -> unit
end
```

Le module `Typechecker`

```
let rec synth_exp cx = function
| Int _ -> TInt
| Add (e1, e2) -> check_exp cx TInt e1; check_exp cx TInt e2; TInt
| App (e1, e2) -> app_types cx (synth_exp e1) e2
| Let (e, b) -> synth_bound1 cx (synth_exp e) b
| Fun (dom, b) -> TFun (dom, synth_bound1 cx dom b)
| Var x -> (try List.assoc x cx with Not_found -> unbound_variable x)

and check_exp cx expected e =
  let given = synth_exp cx e in
  if given <> expected then type_mismatch ~expected ~given e

and app_types cx ty_f e2 = match ty_f with
  | TFun (dom, cod) -> check_exp cx dom e2; cod
  | given -> type_mismatch_expected_fun given

and synth_bound1 cx bty b = synth_exp ((b.bound, bty) :: cx) b.body
```

Sûreté du typage et remarques

Théorème (Sûreté du typage)

Si `check_exp [] e ty` *termine sans erreur, il existe une valeur* `v` *telle que :*

1. `eval_exp [] e = v`, *en particulier sans lever d'exception,*
2. `check_val v ty` *termine sans erreur.*

Sûreté du typage et remarques

Théorème (Sûreté du typage)

Si `check_exp [] e ty` *termine sans erreur, il existe une valeur* `v` *telle que :*

1. `eval_exp [] e = v`, *en particulier sans lever d'exception,*
2. `check_val v ty` *termine sans erreur.*

Remarques

- Requiert une annotation de type sur tous les paramètres de fonctions.
  - De même en C, Java, Rust, etc. OCaml est capable de l'inférer.
- Un appel à `synth_exp` parcourt chaque sous-terme exactement une fois, et en particulier termine toujours ; à contraster avec l'évaluation !

```
eval [] Build.(let f = fun_ (fun x -> app x [x]) in app f [f])
```

Sûreté du typage et remarques

Théorème (Sûreté du typage)

Si `check_exp [] e ty` *termine sans erreur, il existe une valeur* `v` *telle que :*

1. `eval_exp [] e = v`, *en particulier sans lever d'exception,*
2. `check_val v ty` *termine sans erreur.*

Remarques

- Requiert une annotation de type sur tous les paramètres de fonctions.
  - De même en C, Java, Rust, etc. OCaml est capable de l'inférer.
- Un appel à `synth_exp` parcourt chaque sous-terme exactement une fois, et en particulier termine toujours ; à contraster avec l'évaluation !

```
eval [] Build.(let f = fun_ (fun x -> app x [x]) in app f [f])
```

Pour démontrer le théorème, on va en général formuler le système de types dans un style mathématique plus abstrait que le code OCaml.

Plan

Les jugements de typage

On va définir un ensemble de triplets

$$\mathcal{T} \subseteq Cx \times Expr \times Type$$

tel que $(\Gamma, \mathsf{e}, \mathsf{t}) \in \mathcal{T}$ lorsque e est de type t dans le contexte Γ.

Les jugements de typage

On va définir un ensemble de triplets

$$\mathcal{T} \subseteq Cx \times Expr \times Type$$

tel que $(\Gamma, \mathsf{e}, \mathsf{t}) \in \mathcal{T}$ lorsque e est de type t dans le contexte Γ.

L'ensemble $\mathcal{T}$ est défini comme le plus petit sous-ensemble de

$$Cx \times Expr \times Type$$

qui satisfait un jeu propriétés correspondant aux règles de typage.

- $\{(\Gamma, \texttt{Int n}, \texttt{int}) \mid \Gamma \in Ctx\} \subseteq \mathcal{T}$
- $\{(\Gamma, \texttt{e1 + e2}, \texttt{int}) \mid \Gamma \in Ctx, (\Gamma, \texttt{e1}, \texttt{int}) \in \mathcal{T}, (\Gamma, \texttt{e2}, \texttt{int}) \in \mathcal{T}\} \subseteq \mathcal{T}$
- $\{(\Gamma, \mathsf{x}, \mathsf{t}) \mid \Gamma \ni \mathsf{x} : \mathsf{t}\} \subseteq \mathcal{T}$
- ...

Les jugements de typage

On va définir un ensemble de triplets

$$\mathcal{T} \subseteq Cx \times Expr \times Type$$

tel que $(\Gamma, \mathsf{e}, \mathsf{t}) \in \mathcal{T}$ lorsque e est de type t dans le contexte Γ.

L'ensemble $\mathcal{T}$ est défini comme le plus petit sous-ensemble de

$$Cx \times Expr \times Type$$

qui satisfait un jeu propriétés correspondant aux règles de typage.

- $\{(\Gamma, \texttt{Int n}, \texttt{int}) \mid \Gamma \in Ctx\} \subseteq \mathcal{T}$
- $\{(\Gamma, \texttt{e1 + e2}, \texttt{int}) \mid \Gamma \in Ctx, (\Gamma, \texttt{e1}, \texttt{int}) \in \mathcal{T}, (\Gamma, \texttt{e2}, \texttt{int}) \in \mathcal{T}\} \subseteq \mathcal{T}$
- $\{(\Gamma, \mathsf{x}, \mathsf{t}) \mid \Gamma \ni \mathsf{x} : \mathsf{t}\} \subseteq \mathcal{T}$
- ...

Classiquement, l'existence de $\mathcal{T}$ est assurée par le théorème de Tarski. On peut aussi l'expliquer par une définiton *inductive* (cf. parsing, jalon 2).

Une présentation inductive du jugement de typage

On écrit $\Gamma \vdash \texttt{e} : \texttt{t}$ pour $(\Gamma, \texttt{e}, \texttt{t}) \in \mathcal{T}$.

$$\textsc{Var}\ \frac{\Gamma \ni \texttt{x} : \texttt{t}}{\Gamma \vdash \texttt{x} : \texttt{t}} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash \texttt{n} : \texttt{int}} \qquad \textsc{Plus}\ \frac{\Gamma \vdash \texttt{e1} : \texttt{int} \qquad \Gamma \vdash \texttt{e2} : \texttt{int}}{\Gamma \vdash \texttt{e1 + e2} : \texttt{int}}$$

$$\textsc{Let}\ \frac{\Gamma \vdash \texttt{e} : \texttt{t} \qquad \Gamma, \texttt{x} : \texttt{t} \vdash \texttt{e'} : \texttt{t'}}{\Gamma \vdash \texttt{let x = e in e'} : \texttt{t'}} \qquad \textsc{Fun}\ \frac{\Gamma, \texttt{x} : \texttt{t} \vdash \texttt{e} : \texttt{t'}}{\Gamma \vdash (\texttt{fun (x : t) -> e}) : \texttt{t} \to \texttt{t'}}$$

$$\textsc{App}\ \frac{\Gamma \vdash \texttt{e} : \texttt{t} \to \texttt{t'} \qquad \Gamma \vdash \texttt{e'} : \texttt{t}}{\Gamma \vdash \texttt{e e'} : \texttt{t'}}$$

Une présentation inductive du jugement de typage

On écrit $\Gamma \vdash \texttt{e} : \texttt{t}$ pour $(\Gamma, \texttt{e}, \texttt{t}) \in \mathcal{T}$.

$$\textsc{Var}\ \frac{\Gamma \ni \texttt{x} : \texttt{t}}{\Gamma \vdash \texttt{x} : \texttt{t}} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash \texttt{n} : \texttt{int}} \qquad \textsc{Plus}\ \frac{\Gamma \vdash \texttt{e1} : \texttt{int} \qquad \Gamma \vdash \texttt{e2} : \texttt{int}}{\Gamma \vdash \texttt{e1 + e2} : \texttt{int}}$$

$$\textsc{Let}\ \frac{\Gamma \vdash \texttt{e} : \texttt{t} \qquad \Gamma, \texttt{x} : \texttt{t} \vdash \texttt{e'} : \texttt{t'}}{\Gamma \vdash \texttt{let x = e in e'} : \texttt{t'}} \qquad \textsc{Fun}\ \frac{\Gamma, \texttt{x} : \texttt{t} \vdash \texttt{e} : \texttt{t'}}{\Gamma \vdash (\texttt{fun (x : t) -> e}) : \texttt{t} \to \texttt{t'}}$$

$$\textsc{App}\ \frac{\Gamma \vdash \texttt{e} : \texttt{t} \to \texttt{t'} \qquad \Gamma \vdash \texttt{e'} : \texttt{t}}{\Gamma \vdash \texttt{e e'} : \texttt{t'}}$$

Définition (Dérivabilité)

Le jugement $\Gamma \vdash \texttt{e} : \texttt{t}$ est *dérivable* si on peut construire un arbre fini de racine $\Gamma \vdash \texttt{e} : \texttt{t}$ dont les nœuds sont étiquetés par les règles ci-dessus.

Présentation déclarative et présentation algorithmique

La présentation mathématique sous forme d'un jugement inductif :

- $+$ est plus abstraite que le code (ne mentionne que les cas bien typés) ;
- $+$ se prête au raisonnement par induction sur les dérivations ;
- $\pm$ ne décrit pas *a priori* un algorithme ! C'est une formulation *déclarative*.

Présentation déclarative et présentation algorithmique

La présentation mathématique sous forme d'un jugement inductif :

- $+$ est plus abstraite que le code (ne mentionne que les cas bien typés) ;
- $+$ se prête au raisonnement par induction sur les dérivations ;
- $\pm$ ne décrit pas *a priori* un algorithme ! C'est une formulation *déclarative*.

Proposition (Déterminisme du typage)

Si $\Gamma \vdash \mathtt{e} : \mathtt{t1}$ *et* $\Gamma \vdash \mathtt{e} : \mathtt{t2}$ *alors* $\mathtt{t1} = \mathtt{t2}$.

Démonstration.

Par induction sur la dérivation de $\Gamma \vdash \mathtt{e} : \mathtt{t1}$. □

Présentation déclarative et présentation algorithmique

La présentation mathématique sous forme d'un jugement inductif :

- $+$ est plus abstraite que le code (ne mentionne que les cas bien typés) ;
- $+$ se prête au raisonnement par induction sur les dérivations ;
- $\pm$ ne décrit pas *a priori* un algorithme ! C'est une formulation *déclarative*.

Proposition (Déterminisme du typage)

Si $\Gamma \vdash \texttt{e} : \texttt{t1}$ *et* $\Gamma \vdash \texttt{e} : \texttt{t2}$ *alors* $\texttt{t1} = \texttt{t2}$.

Démonstration.

Par induction sur la dérivation de $\Gamma \vdash \texttt{e} : \texttt{t1}$. □

Pouvez-vous donner un exemple de règle sûre qui invalide la propriété ?

Présentation déclarative et présentation algorithmique

La présentation mathématique sous forme d'un jugement inductif :

- $+$ est plus abstraite que le code (ne mentionne que les cas bien typés) ;
- $+$ se prête au raisonnement par induction sur les dérivations ;
- $\pm$ ne décrit pas *a priori* un algorithme ! C'est une formulation *déclarative*.

Proposition (Déterminisme du typage)

Si $\Gamma \vdash \texttt{e} : \texttt{t1}$ *et* $\Gamma \vdash \texttt{e} : \texttt{t2}$ *alors* $\texttt{t1} = \texttt{t2}$.

Démonstration.

Par induction sur la dérivation de $\Gamma \vdash \texttt{e} : \texttt{t1}$. □

Pouvez-vous donner un exemple de règle sûre qui invalide la propriété ?

$$\frac{\Gamma, \texttt{x} : \texttt{t} \vdash \texttt{e} : \texttt{t'}}{\Gamma \vdash (\texttt{fun x -> e}) : \texttt{t} \to \texttt{t'}}$$

Exemple : la gestion des booléens

Exercice

Ajoutez l'équivalent des règles ci-dessous à `synth_exp`.

$$\textsc{Bool}\ \frac{}{\Gamma \vdash \texttt{b} : \texttt{bool}} \qquad \textsc{If}\ \frac{\Gamma \vdash \texttt{e1} : \texttt{bool} \qquad \Gamma \vdash \texttt{e2} : \texttt{t} \qquad \Gamma \vdash \texttt{e3} : \texttt{t}}{\Gamma \vdash \texttt{if e1 then e2 else e3} : \texttt{t}}$$

Exemple : la gestion des booléens

Exercice

Ajoutez l'équivalent des règles ci-dessous à `synth_exp`.

$$\textsc{Bool}\ \frac{}{\Gamma \vdash \texttt{b} : \texttt{bool}} \qquad \textsc{If}\ \frac{\Gamma \vdash \texttt{e1} : \texttt{bool} \qquad \Gamma \vdash \texttt{e2} : \texttt{t} \qquad \Gamma \vdash \texttt{e3} : \texttt{t}}{\Gamma \vdash \texttt{if e1 then e2 else e3} : \texttt{t}}$$

```
let rec synth_exp cx = function
...
| Bool _ -> TBool
| If (e1, e2, e3) ->
  check_exp cx TBool e1;
  let t = synth_exp cx e2 in (* choix arbitraire, comme pour Add *)
  check_exp cx t e3;
  t
```

De la spécification à l'implémentation

Comment implémenter un jugement

$$\mathcal{J} \subseteq A_1 \times \cdots \times A_n$$

défini inductivement sous la forme d'une fonction récursive partielle ?

Une recette informelle

1. Identifier le *mode* de chaque composante A_i : entrée ou sortie ?
2. Déterminer une entrée A_i qui décroît à chaque appel récursif.
3. Regrouper les règles selon la forme de l'entrée A_i.

De la spécification à l'implémentation

Comment implémenter un jugement

$$\mathcal{J} \subseteq A_1 \times \cdots \times A_n$$

défini inductivement sous la forme d'une fonction récursive partielle ?

Une recette informelle

1. Identifier le *mode* de chaque composante A_i : entrée ou sortie ?
2. Déterminer une entrée A_i qui décroît à chaque appel récursif.
3. Regrouper les règles selon la forme de l'entrée A_i.

Cette recette fonctionne pour les jugements d'évaluation (jalon 2) et de typage (jalon 3), pas pour le jugement d'acceptation d'un mot par une grammaire (cours d'analyse syntaxique).

Plan

Plan

Des limites du typage monodirectionnel

Notre vérificateur de types requiert beaucoup d'annotations. Par exemple :

```
let g = fun (f : int -> int) -> fun (x : int) -> f (x + 1) in
g (fun (y : int) -> y + 1)
```

L'annotation sur `y` est clairement redondante, au vu du type de `g`.

Des limites du typage monodirectionnel

Notre vérificateur de types requiert beaucoup d'annotations. Par exemple :

```
let g = fun (f : int -> int) -> fun (x : int) -> f (x + 1) in
g (fun (y : int) -> y + 1)
```

L'annotation sur `y` est clairement redondante, au vu du type de `g`.

Deux approches parmi d'autres

1. L'inférence de type permet d'éviter toute annotation dans le code ci-dessus via un processus global de résolution de contrainte.
2. Le typage bidirectionnel permet d'éviter certaines annotations par un processus local de propagation d'information.

Des limites du typage monodirectionnel

Notre vérificateur de types requiert beaucoup d'annotations. Par exemple :

```
let g = fun (f : int -> int) -> fun (x : int) -> f (x + 1) in
g (fun (y : int) -> y + 1)
```

L'annotation sur `y` est clairement redondante, au vu du type de `g`.

Deux approches parmi d'autres

1. L'inférence de type permet d'éviter toute annotation dans le code ci-dessus via un processus global de résolution de contrainte.
2. Le typage bidirectionnel permet d'éviter certaines annotations par un processus local de propagation d'information.

Étudions la deuxième approche, à implémenter pour Hopix dans le jalon 3.

Retour au vérificateur de types

```
let rec synth_exp cx = function
| Var x -> lookup x cx | Int _ -> TInt
| Add (e1, e2) -> check_exp cx TInt e1; check_exp cx TInt e2; TInt
| App (e1, e2) -> app_types cx (synth_exp cx e1) e2
| Let (e, b) -> synth_bound1 cx (synth_exp cx e) b
| Fun (dom, b) -> TFun (dom, synth_bound1 cx dom b)

and check_exp cx expected e =
  let given = synth_exp cx e in
  if given <> expected then type_mismatch ~expected ~given e
```

Retour au vérificateur de types

```
let rec synth_exp cx = function
| Var x -> lookup x cx | Int _ -> TInt
| Add (e1, e2) -> check_exp cx TInt e1; check_exp cx TInt e2; TInt
| App (e1, e2) -> app_types cx (synth_exp cx e1) e2
| Let (e, b) -> synth_bound1 cx (synth_exp cx e) b
| Fun (dom, b) -> TFun (dom, synth_bound1 cx dom b)

and check_exp cx expected e =
  let given = synth_exp cx e in
  if given <> expected then type_mismatch ~expected ~given e
```

Observations

- La fonction `check_exp` n'exploite pas le type `expected`.
- La notion d'annotation de typage a du sens au delà des paramètres.

```
type exp = ... | Fun of bound1 | Annot of exp * ty
```

Un algorithme de typage bidirectionnel

```
let rec synth_exp cx = function
| Var x -> lookup x cx | Int _ -> TInt
| Add (e1, e2) -> check_exp cx TInt e1; check_exp cx TInt e2; TInt
| App (e1, e2) -> app_types cx (synth_exp cx e1) e2
| Let (e, b) -> synth_bound1 (synth_exp cx e) b
| Annot (e, expected) -> check_exp cx expected e; expected
| (Fun _) as e -> cannot_synthesize e

and check_exp cx expected e = match e with
| Var _ | Int _ | Add _ | App _ | Annot _ ->
  let given = synth_exp cx e in
  if given <> expected then type_mismatch ~expected ~given e
| Let (e, b) ->
  check_bound1 cx (synth_exp cx e) expected b
| Fun b ->
  begin match expected with
  | TFun (dom, cod) -> check_bound1 cx dom cod b
  | _ -> type_mismatch_expected_fun expected
  end
```

Plan

Deux jugements inductifs pour le typage bidirectionnel

$$\textsc{SynthVar}\ \frac{\Gamma \ni \mathsf{x} : \mathsf{t}}{\Gamma \vdash \mathsf{x} \Rightarrow \mathsf{t}} \qquad \textsc{SynthInt}\ \frac{}{\Gamma \vdash \mathsf{n} \Rightarrow \mathbf{int}} \qquad \textsc{SynthPlus}\ \frac{\Gamma \vdash \mathsf{e1} \Leftarrow \mathbf{int} \qquad \Gamma \vdash \mathsf{e2} \Leftarrow \mathbf{int}}{\Gamma \vdash \mathsf{e1} + \mathsf{e2} \Rightarrow \mathbf{int}}$$

$$\textsc{SynthLet}\ \frac{\Gamma \vdash \mathsf{e} \Rightarrow \mathsf{t} \qquad \Gamma, \mathsf{x} : \mathsf{t} \vdash \mathsf{e'} \Rightarrow \mathsf{t'}}{\Gamma \vdash \mathbf{let}\ \mathsf{x} = \mathsf{e}\ \mathbf{in}\ \mathsf{e'} \Rightarrow \mathsf{t'}} \qquad \textsc{SynthApp}\ \frac{\Gamma \vdash \mathsf{e} \Rightarrow \mathsf{t} \to \mathsf{t'} \qquad \Gamma \vdash \mathsf{e'} \Leftarrow \mathsf{t}}{\Gamma \vdash \mathsf{e}\ \mathsf{e'} \Rightarrow \mathsf{t'}}$$

$$\textsc{SynthCheck}\ \frac{\Gamma \vdash \mathsf{e} \Leftarrow \mathsf{t}}{\Gamma \vdash (\mathsf{e} : \mathsf{t}) \Rightarrow \mathsf{t}} \qquad \textsc{CheckSynth}\ \frac{\Gamma \vdash \mathsf{e} \Rightarrow \mathsf{t}}{\Gamma \vdash \mathsf{e} \Leftarrow \mathsf{t}} \qquad \textsc{CheckFun}\ \frac{\Gamma, \mathsf{x} : \mathsf{t} \vdash \mathsf{e} \Leftarrow \mathsf{t'}}{\Gamma \vdash (\mathbf{fun}\ \mathsf{x} \to \mathsf{e}) \Leftarrow \mathsf{t} \to \mathsf{t'}}$$

$$\textsc{CheckLet}\ \frac{\Gamma \vdash \mathsf{e} \Rightarrow \mathsf{t} \qquad \Gamma, \mathsf{x} : \mathsf{t} \vdash \mathsf{e'} \Leftarrow \mathsf{t'}}{\Gamma \vdash \mathbf{let}\ \mathsf{x} = \mathsf{e}\ \mathbf{in}\ \mathsf{e'} \Leftarrow \mathsf{t'}}$$

Quels résultats ?

On note $\Gamma \vdash \texttt{e} : \texttt{t}$ pour le jugement *déclaratif* muni de la règle ci-dessous.

$$\frac{\Gamma, \texttt{x} : \texttt{t} \vdash \texttt{e} : \texttt{t'}}{\Gamma \vdash (\texttt{fun x -> e}) : \texttt{t} \to \texttt{t'}}$$

On écrit $\lfloor \texttt{e} \rfloor$ pour l'expression e dont on a effacé toutes les annotations.

Quels résultats ?

On note $\Gamma \vdash \texttt{e} : \texttt{t}$ pour le jugement *déclaratif* muni de la règle ci-dessous.

$$\frac{\Gamma, \texttt{x} : \texttt{t} \vdash \texttt{e} : \texttt{t'}}{\Gamma \vdash (\texttt{fun x -> e}) : \texttt{t} \to \texttt{t'}}$$

On écrit $\lfloor \texttt{e} \rfloor$ pour l'expression e dont on a effacé toutes les annotations.

Théorème (Correction du typage bidirectionnel)

Si $\Gamma \vdash \texttt{e} \Rightarrow \texttt{t}$ *ou* $\Gamma \vdash \texttt{e} \Leftarrow \texttt{t}$ *alors* $\Gamma \vdash \lfloor \texttt{e} \rfloor : \texttt{t}$.

Quels résultats ?

On note $\Gamma \vdash \texttt{e} : \texttt{t}$ pour le jugement *déclaratif* muni de la règle ci-dessous.

$$\frac{\Gamma, \texttt{x} : \texttt{t} \vdash \texttt{e} : \texttt{t'}}{\Gamma \vdash (\texttt{fun x -> e}) : \texttt{t} \to \texttt{t'}}$$

On écrit $\lfloor \texttt{e} \rfloor$ pour l'expression e dont on a effacé toutes les annotations.

Théorème (Correction du typage bidirectionnel)

Si $\Gamma \vdash \texttt{e} \Rightarrow \texttt{t}$ *ou* $\Gamma \vdash \texttt{e} \Leftarrow \texttt{t}$ *alors* $\Gamma \vdash \lfloor \texttt{e} \rfloor : \texttt{t}$.

Théorème (Complétude partielle du typage bidirectionnel)

Si $\Gamma \vdash \texttt{e} : \texttt{t}$ *alors :*

1. *il existe* $\texttt{e1} \in \lfloor \texttt{e} \rfloor^{-1}$ *tel que* $\Gamma \vdash \texttt{e1} \Rightarrow \texttt{t}$,
2. *il existe* $\texttt{e2} \in \lfloor \texttt{e} \rfloor^{-1}$ *tel que* $\Gamma \vdash \texttt{e2} \Leftarrow \texttt{t}$.

Gestion des expressions booléennes

Exercice

Étendre le typage bidirectionnel aux conditionnelles.

Gestion des expressions booléennes

Exercice

Étendre le typage bidirectionnel aux conditionnelles.

$$\textsc{CheckIf}\ ?\quad \frac{\Gamma \vdash \texttt{e1} \Leftarrow \texttt{bool} \qquad \Gamma \vdash \texttt{e2} \Leftarrow \texttt{t} \qquad \Gamma \vdash \texttt{e3} \Leftarrow \texttt{t}}{\Gamma \vdash \texttt{if e1 then e2 else e3} \Leftarrow \texttt{t}}$$

Gestion des expressions booléennes

Exercice

Étendre le typage bidirectionnel aux conditionnelles.

$$\textsc{CheckIf}\ ?\quad \frac{\Gamma \vdash \texttt{e1} \Leftarrow \texttt{bool} \qquad \Gamma \vdash \texttt{e2} \Leftarrow \texttt{t} \qquad \Gamma \vdash \texttt{e3} \Leftarrow \texttt{t}}{\Gamma \vdash \texttt{if e1 then e2 else e3} \Leftarrow \texttt{t}}$$

$$\textsc{SynthIf1}\ ?\quad \frac{\Gamma \vdash \texttt{e1} \Leftarrow \texttt{bool} \qquad \Gamma \vdash \texttt{e2} \Rightarrow \texttt{t} \qquad \Gamma \vdash \texttt{e3} \Leftarrow \texttt{t}}{\Gamma \vdash \texttt{if e1 then e2 else e3} \Rightarrow \texttt{t}}$$

Gestion des expressions booléennes

Exercice

Étendre le typage bidirectionnel aux conditionnelles.

$$\textsc{CheckIf}\ ?\quad \frac{\Gamma \vdash \texttt{e1} \Leftarrow \texttt{bool} \qquad \Gamma \vdash \texttt{e2} \Leftarrow \texttt{t} \qquad \Gamma \vdash \texttt{e3} \Leftarrow \texttt{t}}{\Gamma \vdash \texttt{if e1 then e2 else e3} \Leftarrow \texttt{t}}$$

$$\textsc{SynthIf1}\ ?\quad \frac{\Gamma \vdash \texttt{e1} \Leftarrow \texttt{bool} \qquad \Gamma \vdash \texttt{e2} \Rightarrow \texttt{t} \qquad \Gamma \vdash \texttt{e3} \Leftarrow \texttt{t}}{\Gamma \vdash \texttt{if e1 then e2 else e3} \Rightarrow \texttt{t}}$$

$$\textsc{SynthIf2}\ ?\quad \frac{\Gamma \vdash \texttt{e1} \Leftarrow \texttt{bool} \qquad \Gamma \vdash \texttt{e2} \Leftarrow \texttt{t} \qquad \Gamma \vdash \texttt{e3} \Rightarrow \texttt{t}}{\Gamma \vdash \texttt{if e1 then e2 else e3} \Rightarrow \texttt{t}}$$

Plan

Bilan au sujet du typage bidirectionnel

L'expression ci-dessous est de type `int -> int` d'après `synth_exp`.

```
let g = (fun f -> fun x -> f (x + 1)) : (int -> int) -> int -> int in
g (fun y -> y + 1)
```

On peut trouver cette quantité d'annotations raisonnable.

Bilan

- Une technique simple pour réduire la quantité d'annotations.
  - Pas aussi flexible que l'inférence de types à la ML. Complétude partielle.
  - Compatible avec des systèmes bien plus expressifs, e.g., sous-typage.
- Parfois délicat de choisir une direction pour une construction donnée.
  - Pour la conditionnelle, le choix de CheckIf est le plus naturel.

Bilan au sujet du typage bidirectionnel

L'expression ci-dessous est de type `int -> int` d'après `synth_exp`.

```
let g = (fun f -> fun x -> f (x + 1)) : (int -> int) -> int -> int in
g (fun y -> y + 1)
```

On peut trouver cette quantité d'annotations raisonnable.

Bilan

- Une technique simple pour réduire la quantité d'annotations.
  - Pas aussi flexible que l'inférence de types à la ML. Complétude partielle.
  - Compatible avec des systèmes bien plus expressifs, e.g., sous-typage.
- Parfois délicat de choisir une direction pour une construction donnée.
  - Pour la conditionnelle, le choix de CheckIf est le plus naturel.

Le typage bidirectionnel s'étend facilement pour traiter types algébriques, enregistrements, références, etc. Que manque-t-il pour Hopix ?

Plan

Plan

Pourquoi le polymorphisme ?

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `int -> int`

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `(int -> int) -> (int -> int)`

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `((int -> int) -> int) -> ((int -> int) -> int)`

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `t -> t` quel que soit `t`

Pourquoi le polymorphisme ?

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `int -> int`

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `(int -> int) -> (int -> int)`

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `((int -> int) -> int) -> ((int -> int) -> int)`

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `t -> t` quel que soit `t`

Un défaut d'expressivité

Aucun type ne représente la famille de types $\{\texttt{t -> t} \mid \texttt{t} \in Ty\}$.

Pourquoi le polymorphisme ?

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `int -> int`

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `(int -> int) -> (int -> int)`

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `((int -> int) -> int) -> ((int -> int) -> int)`

$\Gamma \vdash$ `fun x -> x` $\Leftarrow$ `t -> t` quel que soit `t`

Un défaut d'expressivité

Aucun type ne représente la famille de types $\{\texttt{t -> t} \mid \texttt{t} \in Ty\}$.

La quantification universelle et le polymorphisme

On peut représenter cette famille de types par le type $\forall\alpha.\,\alpha$ `->` α.

On aura donc, dans l'exemple, $\Gamma \vdash$ `fun x -> x` $: \forall\alpha.\,\alpha$ `->` α.

Le typage polymorphe

$$\frac{?}{\Gamma \vdash \mathsf{e} : \forall \alpha.\, \mathsf{t}} \qquad \frac{\Gamma \vdash \mathsf{e}' : \forall \alpha.\, \mathsf{t}}{?}$$

Quelles intuitions ?

Le typage polymorphe

$$\frac{?}{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}} \qquad \frac{\Gamma \vdash \mathtt{e'} : \forall \alpha.\, \mathtt{t}}{?}$$

Quelles intuitions ?

- L'expression e doit se comporter *uniformément* vis-à-vis de α pour être de type polymorphe. Exemple canonique : `List.length`.

Le typage polymorphe

$$\frac{?}{\Gamma \vdash \texttt{e} : \forall \alpha.\, \texttt{t}} \qquad \frac{\Gamma \vdash \texttt{e'} : \forall \alpha.\, \texttt{t}}{?}$$

Quelles intuitions ?

- L'expression `e` doit se comporter *uniformément* vis-à-vis de α pour être de type polymorphe. Exemple canonique : `List.length`.
- L'expression `e'` peut être utilisée avec n'importe quel type `t` où α a été remplacé par un `t'` quelconque... ou presque.

Le typage polymorphe

$$\frac{\textcolor{orange}{?}}{\Gamma \vdash \texttt{e} : \forall \alpha.\, \texttt{t}} \qquad\qquad \frac{\Gamma \vdash \texttt{e'} : \forall \alpha.\, \texttt{t}}{\textcolor{orange}{?}}$$

Quelles intuitions ?

- L'expression `e` doit se comporter *uniformément* vis-à-vis de α pour être de type polymorphe. Exemple canonique : `List.length`.
- L'expression `e'` peut être utilisée avec n'importe quel type `t` où α a été remplacé par un `t'` quelconque... ou presque.

On peut essayer capturer ces intuitions, déjà via un jugement déclaratif.

Règles déclaratives naïves pour le polymorphisme

$$\textsc{Generalize}\ \frac{\Gamma \vdash \mathsf{e} : \mathsf{t}}{\Gamma \vdash \mathsf{e} : \forall \alpha.\, \mathsf{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathsf{e} : \forall \alpha.\, \mathsf{t}}{\Gamma \vdash \mathsf{e} : \mathsf{t}[\alpha \backslash \mathsf{t}']}$$

Ces règles sont-elles satisfaisantes ?

Règles déclaratives naïves pour le polymorphisme

$$\textsc{Generalize}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{t}}{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}}{\Gamma \vdash \mathtt{e} : \mathtt{t}[\alpha \backslash \mathtt{t'}]}$$

Ces règles sont-elles satisfaisantes ?

$$\frac{}{\bullet \vdash \texttt{fun x -> x} : \forall \alpha.\, \forall \beta.\, \alpha \texttt{ -> } \beta}$$

Règles déclaratives naïves pour le polymorphisme

$$\textsc{Generalize}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{t}}{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}}{\Gamma \vdash \mathtt{e} : \mathtt{t}[\alpha \backslash \mathtt{t'}]}$$

Ces règles sont-elles satisfaisantes ?

$$\frac{\dfrac{}{\bullet \vdash \texttt{fun x -> x} : \alpha \texttt{ -> } \beta}}{\bullet \vdash \texttt{fun x -> x} : \forall \alpha.\, \forall \beta.\, \alpha \texttt{ -> } \beta}$$

Règles déclaratives naïves pour le polymorphisme

$$\textsc{Generalize}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{t}}{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}}{\Gamma \vdash \mathtt{e} : \mathtt{t}[\alpha \backslash \mathtt{t'}]}$$

Ces règles sont-elles satisfaisantes ?

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \beta}}{\bullet \vdash \texttt{fun x -> x} : \alpha \texttt{ -> } \beta}}{\bullet \vdash \texttt{fun x -> x} : \forall \alpha.\, \forall \beta.\, \alpha \texttt{ -> } \beta}$$

Règles déclaratives naïves pour le polymorphisme

$$\textsc{Generalize}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{t}}{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}}{\Gamma \vdash \mathtt{e} : \mathtt{t}[\alpha \backslash \mathtt{t'}]}$$

Ces règles sont-elles satisfaisantes ?

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \forall \alpha.\, \alpha}}{x : \alpha \vdash x : \beta}}{\bullet \vdash \texttt{fun x -> x} : \alpha \texttt{ -> } \beta}$$
$$\bullet \vdash \texttt{fun x -> x} : \forall \alpha.\, \forall \beta.\, \alpha \texttt{ -> } \beta$$

Règles déclaratives naïves pour le polymorphisme

$$\textsc{Generalize}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{t}}{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}}{\Gamma \vdash \mathtt{e} : \mathtt{t}[\alpha \backslash \mathtt{t'}]}$$

Ces règles sont-elles satisfaisantes ?

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : \alpha \ni x : \alpha}{x : \alpha \vdash x : \alpha}}{x : \alpha \vdash x : \forall \alpha.\, \alpha}}{x : \alpha \vdash x : \beta}}{\bullet \vdash \texttt{fun x -> x} : \alpha \texttt{ -> } \beta}}{\bullet \vdash \texttt{fun x -> x} : \forall \alpha.\, \forall \beta.\, \alpha \texttt{ -> } \beta}$$

Règles déclaratives naïves pour le polymorphisme

$$\textsc{Generalize}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{t}}{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}}{\Gamma \vdash \mathtt{e} : \mathtt{t}[\alpha \backslash \mathtt{t'}]}$$

Ces règles sont-elles satisfaisantes ?

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha \ni x : \alpha}}{x : \alpha \vdash x : \alpha}}{x : \alpha \vdash x : \forall \alpha.\, \alpha}}{x : \alpha \vdash x : \beta}}{\bullet \vdash \texttt{fun x -> x} : \alpha \texttt{ -> } \beta}$$

$$\bullet \vdash \texttt{fun x -> x} : \forall \alpha.\, \forall \beta.\, \alpha \texttt{ -> } \beta$$

Ces règles sont donc fausses, invalidant la sûreté du typage.

```
"toto" ^ (fun x -> x) 42
```

Analyse du problème

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : \alpha \ni x : \alpha}{x : \alpha \vdash x : \alpha}}{x : \alpha \vdash x : \forall \alpha.\, \alpha}}{x : \alpha \vdash x : \beta}}{\bullet \vdash \texttt{fun x -> x} : \alpha \texttt{ -> } \beta}}{\bullet \vdash \texttt{fun x -> x} : \forall \alpha.\, \forall \beta.\, \alpha \texttt{ -> } \beta}$$

Analyse du problème

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : \alpha \ni x : \alpha}{x : \alpha \vdash x : \alpha}}{x : \alpha \vdash x : \forall \alpha.\, \alpha}}{x : \alpha \vdash x : \beta}}{\bullet \vdash \texttt{fun x -> x} : \alpha \texttt{ -> } \beta}}{\bullet \vdash \texttt{fun x -> x} : \forall \alpha.\, \forall \beta.\, \alpha \texttt{ -> } \beta}$$

Il y a une confusion au moment de la généralisation. En particulier, cette dérivation est incompatible avec le renommage des variables de types liées.

Analyse du problème

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : \alpha \ni x : \alpha}{x : \alpha \vdash x : \alpha}}{x : \alpha \vdash x : \forall \alpha.\, \alpha}}{x : \alpha \vdash x : \beta}}{\bullet \vdash \texttt{fun x -> x} : \alpha \texttt{ -> } \beta}}{\bullet \vdash \texttt{fun x -> x} : \forall \alpha.\, \forall \beta.\, \alpha \texttt{ -> } \beta}$$

Il y a une confusion au moment de la généralisation. En particulier, cette dérivation est incompatible avec le renommage des variables de types liées.

$$\dfrac{\dfrac{\dfrac{\text{mal typé !}}{x : \alpha \vdash x : \gamma}}{x : \alpha \vdash x : \forall \gamma.\, \gamma}}{x : \alpha \vdash x : \beta}$$

Le polymorphisme en style déclaratif et implicite

On considère les types à renommage des variables de types liées près.

$$\mathtt{t} ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{t} \texttt{ -> } \mathtt{t'} \mid \alpha \mid \forall \alpha.\, \mathtt{t}$$

$$\textsc{Generalize}\ \frac{\Gamma, \alpha \vdash \mathtt{e} : \mathtt{t} \qquad \Gamma \not\ni \alpha}{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathtt{e} : \forall \alpha.\, \mathtt{t} \qquad \Gamma \vdash \mathtt{t'}\ \mathbf{type}}{\Gamma \vdash \mathtt{e} : \mathtt{t}[\alpha \backslash \mathtt{t'}]}$$

$$\frac{\Gamma \ni \alpha}{\Gamma \vdash \alpha\ \mathbf{type}} \qquad \frac{}{\Gamma \vdash \mathtt{int}\ \mathbf{type}} \qquad \ldots \qquad \frac{\Gamma \vdash \mathtt{t}\ \mathbf{type} \qquad \Gamma \vdash \mathtt{t'}\ \mathbf{type}}{\Gamma \vdash \mathtt{t} \texttt{ -> } \mathtt{t'}\ \mathbf{type}}$$

Bilan

$$\textsc{Generalize}\ \frac{\Gamma, \alpha \vdash \mathsf{e} : \mathsf{t} \qquad \Gamma \not\ni \alpha}{\Gamma \vdash \mathsf{e} : \forall \alpha.\, \mathsf{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathsf{e} : \forall \alpha.\, \mathsf{t} \qquad \Gamma \vdash \mathsf{t'}\ \mathbf{type}}{\Gamma \vdash \mathsf{e} : \mathsf{t}[\alpha \backslash \mathsf{t'}]}$$

- Ces règles sont simples et satisfaisantes du point de vue structurel.
  - Règle d'introduction vs. règle d'élimination.
- Elles garantissent la sûreté du typage, au sens précédent.
- Elles sont impossibles à implémenter tel quel ! (Wells 1994)

Bilan

$$\textsc{Generalize}\ \frac{\Gamma, \alpha \vdash \mathsf{e} : \mathsf{t} \qquad \Gamma \not\ni \alpha}{\Gamma \vdash \mathsf{e} : \forall \alpha.\, \mathsf{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathsf{e} : \forall \alpha.\, \mathsf{t} \qquad \Gamma \vdash \mathsf{t'}\ \mathbf{type}}{\Gamma \vdash \mathsf{e} : \mathsf{t}[\alpha \backslash \mathsf{t'}]}$$

- Ces règles sont simples et satisfaisantes du point de vue structurel.
  - Règle d'introduction vs. règle d'élimination.
- Elles garantissent la sûreté du typage, au sens précédent.
- Elles sont impossibles à implémenter tel quel ! (Wells 1994)

Pour la culture : le système F, un langage théorique

- Le langage restreint aux règles traitant fonctions et polymorphisme.
- Tous ses programmes terminent. C'est vraiment difficile à montrer.

Bilan

$$\textsc{Generalize}\ \frac{\Gamma, \alpha \vdash \mathsf{e} : \mathsf{t} \qquad \Gamma \not\ni \alpha}{\Gamma \vdash \mathsf{e} : \forall \alpha.\, \mathsf{t}} \qquad \textsc{Instantiate}\ \frac{\Gamma \vdash \mathsf{e} : \forall \alpha.\, \mathsf{t} \qquad \Gamma \vdash \mathsf{t'}\ \mathbf{type}}{\Gamma \vdash \mathsf{e} : \mathsf{t}[\alpha \backslash \mathsf{t'}]}$$

- Ces règles sont simples et satisfaisantes du point de vue structurel.
  - Règle d'introduction vs. règle d'élimination.
- Elles garantissent la sûreté du typage, au sens précédent.
- Elles sont impossibles à implémenter tel quel ! (Wells 1994)

Pour la culture : le système F, un langage théorique

- Le langage restreint aux règles traitant fonctions et polymorphisme.
- Tous ses programmes terminent. C'est vraiment difficile à montrer.

Plan

Deux solutions possibles

Solution 1 : le polymorphisme explicite

On **écrit** explicitement les points de généralisation et instantiation.

$$\textsc{Generalize}\ \frac{\Gamma, \alpha \vdash \texttt{e} : \texttt{t} \qquad \Gamma \not\ni \alpha}{\Gamma \vdash \texttt{fun (type } \alpha\texttt{) -> e} : \forall \alpha.\, \texttt{t}}$$

$$\textsc{Instantiate}\ \frac{\Gamma \vdash \texttt{e} : \forall \alpha.\, \texttt{t} \qquad \Gamma \vdash \texttt{t'}\ \mathbf{type}}{\Gamma \vdash \texttt{e}\langle \texttt{t'} \rangle : \texttt{t}[\alpha \backslash \texttt{t'}]}$$

Solution 2 : le polymorphisme implicite à la ML

On **restreint** quantification et points d'instanciation du polymorphisme.

$$\begin{aligned} \texttt{t} &::= \texttt{int} \mid \texttt{bool} \mid \texttt{t -> t'} \\ \texttt{s} &::= t \mid \forall \alpha.\, s \end{aligned}$$

Intuition : généraliser les définitions, instancier les occurrences de variables.

Le polymorphisme à la ML, formulé déclarativement

$$\textsc{Let}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{t} \qquad \Gamma, \mathtt{x} : \mathit{Gen}_\Gamma(\mathtt{t}) \vdash \mathtt{e'} : \mathtt{t'}}{\Gamma \vdash \mathtt{let\ x = e\ in\ e'} : \mathtt{t'}} \qquad \textsc{Var}\ \frac{(\Gamma \vdash t_i\ \mathbf{type})_{i \in [1,n]} \qquad \Gamma \ni x : \forall(\alpha_1, \ldots, \alpha_n).\, \mathtt{t}}{\Gamma \vdash \mathtt{x} : \mathtt{t}[\alpha_1 \backslash \mathtt{t}_1, \ldots, \alpha_n \backslash \mathtt{t}_n]}$$

$$\mathit{Gen}_\Gamma(\mathtt{t}) \triangleq \forall(\alpha_1, \ldots, \alpha_n).\, \mathtt{t} \text{ où } \{\alpha_1, \ldots, \alpha_n\} = FV(\mathtt{t}) \setminus FV(\Gamma)$$

Le polymorphisme à la ML, formulé déclarativement

$$\textsc{Let}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{t} \qquad \Gamma, \mathtt{x} : \mathit{Gen}_\Gamma(\mathtt{t}) \vdash \mathtt{e'} : \mathtt{t'}}{\Gamma \vdash \mathbf{let}\ \mathtt{x} = \mathtt{e}\ \mathbf{in}\ \mathtt{e'} : \mathtt{t'}} \qquad \textsc{Var}\ \frac{(\Gamma \vdash t_i\ \mathbf{type})_{i\in[1,n]} \qquad \Gamma \ni x : \forall(\alpha_1, \ldots, \alpha_n).\, \mathtt{t}}{\Gamma \vdash \mathtt{x} : \mathtt{t}[\alpha_1 \backslash \mathtt{t}_1, \ldots, \alpha_n \backslash \mathtt{t}_n]}$$

$$\mathit{Gen}_\Gamma(\mathtt{t}) \triangleq \forall(\alpha_1, \ldots, \alpha_n).\, \mathtt{t} \text{ où } \{\alpha_1, \ldots, \alpha_n\} = FV(\mathtt{t}) \setminus FV(\Gamma)$$

- Le jugement est déclaratif. Si la règle $\textsc{Let}$ est algorithmique, ce n'est pas du tout le cas pour règle $\textsc{Var}$ où il faut deviner les types $\mathtt{t}_i$.

Le polymorphisme à la ML, formulé déclarativement

$$\textsc{Let}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{t} \qquad \Gamma, \mathtt{x} : \mathit{Gen}_\Gamma(\mathtt{t}) \vdash \mathtt{e'} : \mathtt{t'}}{\Gamma \vdash \mathtt{let\ x = e\ in\ e'} : \mathtt{t'}} \qquad \textsc{Var}\ \frac{(\Gamma \vdash t_i\ \mathbf{type})_{i\in[1,n]} \qquad \Gamma \ni x : \forall(\alpha_1, \ldots, \alpha_n).\, \mathtt{t}}{\Gamma \vdash \mathtt{x} : \mathtt{t}[\alpha_1 \backslash \mathtt{t}_1, \ldots, \alpha_n \backslash \mathtt{t}_n]}$$

$$\mathit{Gen}_\Gamma(\mathtt{t}) \triangleq \forall(\alpha_1, \ldots, \alpha_n).\, \mathtt{t} \text{ où } \{\alpha_1, \ldots, \alpha_n\} = FV(\mathtt{t}) \setminus FV(\Gamma)$$

- Le jugement est déclaratif. Si la règle $\textsc{Let}$ est algorithmique, ce n'est pas du tout le cas pour règle $\textsc{Var}$ où il faut deviner les types $\mathtt{t}_i$.
- L'algorithme W de $\textsc{Damas}$ et $\textsc{Milner}$ (1982) est un formulation algorithmique qui sert de base de l'inférence de types à la OCaml.

Splendeurs et misères de l'inférence de types à la ML

Théorème (Correction de l'algorithme W)

Si $\Gamma \vdash \mathtt{e} \Rightarrow_{\mathbf{W}} \mathtt{t}$ *alors* $\Gamma \vdash \mathtt{e} : \mathtt{t}$.

Splendeurs et misères de l'inférence de types à la ML

Théorème (Correction de l'algorithme W)

Si $\Gamma \vdash \mathtt{e} \Rightarrow_{\mathbf{W}} \mathtt{t}$ *alors* $\Gamma \vdash \mathtt{e} : \mathtt{t}$.

Théorème (Complétude de l'algorithme W)

Si $\Gamma \vdash \mathtt{e} : \mathtt{t}$ *alors :*

- *il existe* `t'` *tel que* $\Gamma \vdash \mathtt{e} \Rightarrow_{\mathbf{W}} \mathtt{t'}$, *et*
- *le type* `t` *est une instance (spécialisation) de* `t'`.

Splendeurs et misères de l'inférence de types à la ML

Théorème (Correction de l'algorithme W)

Si $\Gamma \vdash \mathtt{e} \Rightarrow_{\mathbf{W}} \mathtt{t}$ *alors* $\Gamma \vdash \mathtt{e} : \mathtt{t}$.

Théorème (Complétude de l'algorithme W)

Si $\Gamma \vdash \mathtt{e} : \mathtt{t}$ *alors :*

- *il existe* `t'` *tel que* $\Gamma \vdash \mathtt{e} \Rightarrow_{\mathbf{W}} \mathtt{t'}$, *et*
- *le type* `t` *est une instance (spécialisation) de* `t'`.

Splendeurs

L'algorithme W calcule le type principal (le plus général) de l'expression.

Splendeurs et misères de l'inférence de types à la ML

Théorème (Correction de l'algorithme W)

Si $\Gamma \vdash \texttt{e} \Rightarrow_{\mathbf{W}} \texttt{t}$ *alors* $\Gamma \vdash \texttt{e} : \texttt{t}$.

Théorème (Complétude de l'algorithme W)

Si $\Gamma \vdash \texttt{e} : \texttt{t}$ *alors :*

- *il existe* `t'` *tel que* $\Gamma \vdash \texttt{e} \Rightarrow_{\mathbf{W}} \texttt{t'}$, *et*
- *le type* `t` *est une instance (spécialisation) de* `t'`.

Splendeurs

L'algorithme W calcule le type principal (le plus général) de l'expression.

Misères

L'expression `(fun f -> (f 1, f "toto"))` est mal typée en ML.

Plan

La semaine prochaine

Le jalon 3 : vérification bidirectionnelle pour Hopix avec des annotations explicites.

Bibliographie


Damas, Luis et Robin Milner (1982). “Principal type-schemes for functional programs”. In : *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '82*. DOI : 10.1145/582153.582176. URL : https://web.cs.wpi.edu/~cs4536/c12/milner-damas_principal_types.pdf.


Wells, J. B. (1994). “Typability and type checking in the second-order lambda-calculus are equivalent and undecidable”. In : *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*. DOI : 10.1109/lics.1994.316068. URL : http://dx.doi.org/10.1109/lics.1994.316068.