Vérification des résultats de l'inférence de types du compilateur OCaml

Pierrick Couderc

OCamlPro

U2IS, École Nationale Supérieure des Techniques Avancées - ParisTech

23 octobre 2018

Evolution de l'automobile

Par Ramgeis — fotografiert von Ramgeis in Pebble Beach, Kalifornien im August 2004, CC BY-SA 3.0,

https ://commons.wikimedia.org/w/index.php ?curid=604329

Evolution de l'automobile : électronique

Les Numériques - *Automobile : on a vu le tableau de bord du futur*

Evolution de automobile : capteurs et diagnotistics

https ://www.carbon-cleaning.com/diagnostique-auto

Désassemblage et vérification

La voiture est en panne, mais tous les capteurs fonctionnent et n'indiquent pas de panne.

- On désassemble une pièce ;
- Si chacun des éléments qui la compose est correct : OK
- Sinon, on identifie l'élément qui est incorrect et le capteur qui fait défaut, et on analyse son comportement.

Evolution d'un compilateur

L'évolution de n'importe quel compilateur est similaire :

- Initial : Le compilateur supporte un langage simple
- Etape 1 : Ajout une nouvelle construction
- Etape 2 : Réusinage du code
- ...
- Un jour, un défaut apparaît $\Rightarrow$ interaction entre deux constructions qui n'était pas prévue

Evolution d'OCaml

OCaml : compilateur d'un dialecte de ML, multi-paradigme (fonctionnel, impératif, objet, avec modules de première classe et d'ordre supérieur).
Historique du système de types :

- *Caml Light* (1990) : réécriture complète de *CAML*.
- *Caml Special Light* (1995) : modules
- *Objective Caml* (1996) : objets avec sous-typage structurel
- *Objective Caml 3.00* : variants polymorphes, arguments labellisés
- *Objective Caml 3.05* : méthodes et champs polymorphes
- *Objective Caml 3.07* : restriction relâchée de valeurs
- *Objective Caml 3.12* : modules de première classe, type abstraits locaux
- *OCaml 4.00* : types algébriques gardés

OCaml : typage statique

OCaml : langage fortement et statiquement typé.

Typage : Robin Milner

> Well typed programs cannot “go wrong”.

OCaml : typage statique

OCaml : langage fortement et statiquement typé.

Typage : Robin Milner

Well typed programs cannot "go wrong".

Typage (plus simplement)

On ne mélange pas les torchons et les serviettes.

Types

Le calcul `1 + "truc"` est dit *mal typé*

- 1 : un nombre (un entier)
- `"truc"` : un mot (une chaîne de caractères)
- $+$: l'addition, entre deux nombres

L'addition se fait entre deux nombres, donc ce calcul n'a pas de sens.

Système de types

Ensemble des règles qui régissent les types d'un langage.

Inférence et annotations

Exemple d'un lambda-calcul simplement typé, avec des entiers (`int`) :

```
(λ x. x + 1)
```

Inférence et annotations

Exemple d'un lambda-calcul simplement typé, avec des entiers (`int`) :

$(\lambda\ \mathtt{x}.\ \mathtt{x} + 1)$

Annotation par l'inférence (étape par étape) :

$\langle\ \lambda\ \mathtt{x} : \mathtt{int}.\ \langle\ \langle\ \mathtt{x} : \mathtt{int}\ \rangle + 1 \rangle : \mathtt{int}\ \rangle : \mathtt{int} \to \mathtt{int}$

Vérification de l'inférence

But de la vérification :

- Tester la correction du système de types
- Repérer des régressions de l'inférence

Mise-en-œuvre (pour OCaml 4.02) :

- Formalisation d'un ensemble de règles de cohérence (spécification)
- Implémentation d'un vérificateur selon cette spécification formalisée

Plan

1. Arbre de syntaxe abstrait annoté
2. Spécification : MiniML
   - Équivalence de types
   - Filtrage de types
   - Instanciation de schémas
3. Spécification : OCaml
   - Bonne formation de types
   - Égalités locales de types
   - Généralisation
4. Implémentation de vérificateur
5. Conclusion

TypedTree : l'arbre de syntaxe annoté

On ne parlera pas exactement d'OCaml, mais du TypedTree avec la syntaxe d'OCaml.
Le TypedTree $\langle$ e : τ $\rangle$

- L'expression courante : e
- Son type inféré : τ
- ~~L'environnement issu de l'inférence : Γ_{inf}~~ $\Rightarrow$ reconstruit

Cohérence de TypedTree

Pour tout nœud du Typedtree, son expression e peut avoir le type τ dans le contexte Γ reconstruit.

Spécification explicite

Spécification de ML classique :

$$\text{App}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

Spécification explicite

Spécification de ML classique :

$$\text{App}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

Spécification explicite (sans partage de métavariables) :

$$\text{App}\ \frac{\Gamma \vdash \langle e_1 : \tau_1 \rangle \qquad \Gamma \vdash \langle e_2 : \tau_2 \rangle \qquad \Gamma \vdash \tau_1 \prec \tau_2' \to \tau' \qquad \Gamma \vdash \tau_2' \equiv \tau_2 \qquad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash \langle e_1\ e_2 : \tau \rangle}$$

Moteur de l'inférence : l'unification

MiniML

Lambda-calcul avec let polymorphe

Unification est fortement liée à l'inférence, et utilisée pour vérifier plusieurs propriétés (MiniML) :

- Equivalence de types
- Filtrage sur la forme des types
- Instanciation de schémas de types

Equivalence de types

Equivalence : égalité structurelle, modulo des expansions d'abbréviations.

Jugement d'équivalence

$$\Gamma \vdash_{\tau} \tau_1 \equiv \tau_2$$

Equivalence de types

Equivalence : égalité structurelle, modulo des expansions d'abbréviations.

Jugement d'équivalence

$$\Gamma \vdash_\tau \tau_1 \equiv \tau_2$$

Par exemple :

$$\vdash_\tau \texttt{int} \to \texttt{int} \equiv \texttt{int} \to \texttt{int}$$

Equivalence de types

Equivalence : égalité structurelle, modulo des expansions d'abbréviations.

Jugement d'équivalence

$$\Gamma \vdash_\tau \tau_1 \equiv \tau_2$$

Par exemple :

$$\vdash_\tau \texttt{int} \to \texttt{int} \equiv \texttt{int} \to \texttt{int}$$

ou encore :

$$\texttt{t} = \texttt{int} \vdash_\tau \texttt{t} \equiv \texttt{int}$$

Filtrage de types

Filtrage

- Teste la forme d'un type
- Extrait ses sous nœuds sous la forme de métavariables.

Jugement de filtrage (type fonctionnel)

$$\Gamma \vdash_{\tau} \tau \prec \tau_1 \to \tau_2$$

Filtrage de types

Filtrage

- Teste la forme d'un type
- Extrait ses sous nœuds sous la forme de métavariables.

Jugement de filtrage (type fonctionnel)

$$\Gamma \Big|\!\!\frac{}{\tau} \tau \prec \tau_1 \to \tau_2$$

Exemple d'utilisation pour la vérification d'abstraction :

$$\text{Abs}\ \frac{\Gamma \Big|\!\!\frac{}{\tau} \tau' \prec \tau_d \to \tau_{cd} \qquad \Gamma \Big|\!\!\frac{}{\tau} \tau \equiv \tau_d \qquad \Gamma \Big|\!\!\frac{}{\tau} \tau_{cd} \equiv \tau_e \qquad \Gamma \oplus (x : \tau) \Big|\!\!- \langle e : \tau_e \rangle}{\Gamma \Big|\!\!- \langle \lambda x : \tau.\ e : \tau' \rangle}$$

Instanciation de schémas

Instanciation : vérification de la monomorphisation des schémas de type.
i.e. il n'existe qu'une substitution pour les variables liées.

```
let ⟨ f : ∀α.α → α ⟩ = .. in
⟨ f : int → int ⟩ 2
```

`int` $\rightarrow$ `int` est une instance correcte de $\forall\alpha.\alpha \rightarrow \alpha$. On a substitué `int` à α.

Instanciation de schémas

Instanciation : vérification de la monomorphisation des schémas de type.
i.e. il n'existe qu'une substitution pour les variables liées.

$$\texttt{let } \langle\ \texttt{f} : \forall\alpha.\alpha \to \alpha\ \rangle = \texttt{..} \texttt{ in}$$
$$\langle\ \texttt{f} : \texttt{int} \to \texttt{int}\ \rangle\ \texttt{2}$$

$\texttt{int} \to \texttt{int}$ est une instance correcte de $\forall\alpha.\alpha \to \alpha$. On a substitué `int` à α.
$\texttt{int} \to \texttt{bool}$ n'est pas correct, car α a pris deux valeurs incompatibles.

Instanciation : définition inductive

Instanciation :

- Définition inductive, deux règles importantes
- Calcule une substitution θ : environnement qui associe un type à chaque variable de type de son domaine.

Instanciation : définition inductive

Instanciation :

- Définition inductive, deux règles importantes
- Calcule une substitution θ : environnement qui associe un type à chaque variable de type de son domaine.

Jugement d'instanciation

$$\Gamma, \theta \mid\!\!\!\vdash_{\tau} \tau \leq \tau' \Rightarrow \theta'$$

Instanciation : définition inductive

Instanciation :

- Définition inductive, deux règles importantes
- Calcule une substitution θ : environnement qui associe un type à chaque variable de type de son domaine.

Jugement d'instanciation

$$\Gamma, \theta \models_{\tau} \tau \leq \tau' \Rightarrow \theta'$$

$$\text{Inst-Var-Unbound} \quad \frac{\alpha \notin dom(\theta)}{\Gamma, \theta \models_{\tau} \tau \leq \alpha \Rightarrow \theta \oplus [\alpha \mapsto \tau]}$$

Instanciation : définition inductive

Instanciation :

- Définition inductive, deux règles importantes
- Calcule une substitution θ : environnement qui associe un type à chaque variable de type de son domaine.

Jugement d'instanciation

$$\Gamma, \theta \models_{\tau} \tau \leq \tau' \Rightarrow \theta'$$

$$\text{Inst-Var-Unbound}\ \frac{\alpha \notin \mathit{dom}(\theta)}{\Gamma, \theta \models_{\tau} \tau \leq \alpha \Rightarrow \theta \oplus [\alpha \mapsto \tau]} \qquad \text{Inst-Var-Bound}\ \frac{\begin{array}{c}\alpha \in \mathit{dom}(\theta) \\ \mathit{let}\ \tau_\alpha = \theta(\alpha) \\ \Gamma, \theta \models_{\tau} \tau_\alpha \equiv \tau\end{array}}{\Gamma, \theta \models_{\tau} \tau \leq \alpha \Rightarrow \theta}$$

Instanciation et substitution initiale

Avec quel environnement θ commencer l'instanciation ?

$$\texttt{let } \langle \texttt{ f : } \forall\alpha.\beta \to \alpha \texttt{ } \rangle \texttt{ = .. in}$$
$$\langle \texttt{ f : int} \to \texttt{int } \rangle \texttt{ 2}$$

$\texttt{int} \to \texttt{int}$ n'est pas une instance correcte : la variable β est libre dans le type.

Instanciation et substitution initiale

Avec quel environnement θ commencer l'instanciation ?

```
let ⟨ f : ∀α.β → α ⟩ = .. in
⟨ f : int → int ⟩ 2
```

`int` $\rightarrow$ `int` n'est pas une instance correcte : la variable β est libre dans le type.

Avant toute vérification, on génère une *substitution initiale* θ_{init}.

Substitution initiale

$$\theta_{init}(\forall \overline{\alpha}.\tau) = [\beta \mapsto \beta \mid \beta \in (fv(\tau) - \overline{\alpha})]$$

OCaml : vérification d'un TypedTree

L'unification présentée jusqu'ici s'applique à MiniML.
Unification spécifique à OCaml :

- Equivalence de types
- Instanciation de schémas de types
- Filtrage sur la forme des types
- Vérification de non échappement de types
- Génération et vérification des équations de types pour les GADTs

Extrêmement versatile, mais peut être particulièrement complexe ("ctype.ml", 3000 LoC).

Vérification de non-échappement

OCaml supporte les effets de bords, types existentiels et modules de première classe : peuvent être la source des échappements de types.

Vérification de non-échappement

OCaml supporte les effets de bords, types existentiels et modules de première classe : peuvent être la source des échappements de types.

```
let l = ref []

type t = A | B

(* Rejeté *)
let _ = l := [ A ]
```

Vérification de non-échappement

OCaml supporte les effets de bords, types existentiels et modules de première classe : peuvent être la source des échappements de types.

```
let l = ref []

type t = A | B

(* Rejeté *)
let _ = l := [ A ]
```

Jugement de bonne formation

$$\Gamma \left|\frac{}{\tau}\right. \tau \ wf$$

- Validité des constructeurs de types
- Application correcte des paramètres de constructeurs de types
- Cohérence des types de variants polymorphes

Types algébriques gardés

Types algébriques gardés (*GADT*) : généralisation des types algébrique, avec support d'équations de types locales et types existentiels.

```
type (_, _) eq = Eq : ('a, 'a) eq

type _ repr = Int : int t | Bool : bool t

type ex = Ex : 'a * ('a -> string) -> ex
```

Nécessité de supporter des équations de types localement valides.

Equations de types

```
type _ t = Int: int t | Bool: bool t

let f (type a) (x: a t) : a =
  match x with
    Int -> 0
  | Bool -> false
```

- Branche `Int`, `a` $\sim$ `int`
- Branche `Bool`, `a` $\sim$ `bool`

Equations de types

```
type _ t = Int: int t | Bool: bool t

let f (type a) (x: a t) : a =
  match x with
    Int -> 0
  | Bool -> false
```

Les équations sont gérées par des classes d'équivalence (Φ) :

- a est un type abstrait local : $\Phi(a) = \{a\}$.

Equations de types

```
type _ t = Int: int t | Bool: bool t

let f (type a) (x: a t) : a =
  match x with
    Int -> 0
  | Bool -> false
```

Les équations sont gérées par des classes d'équivalence (Φ) :

- a est un type abstrait local : $\Phi(\mathtt{a}) = \{\mathtt{a}\}$.
- Pour la branche `Int` : $\Phi(\mathtt{a}) = \Phi(\mathtt{int}) = \{\mathtt{int},\ \mathtt{a}\}$.

Equations de types

```
type _ t = Int: int t | Bool: bool t

let f (type a) (x: a t) : a =
  match x with
    Int -> 0
  | Bool -> false
```

Les équations sont gérées par des classes d'équivalence (Φ) :

- a est un type abstrait local : $\Phi(\texttt{a}) = \{\texttt{a}\}$.
- Pour la branche `Int` : $\Phi(\texttt{a}) = \Phi(\texttt{int}) = \{\texttt{int}, \texttt{a}\}$.
- Pour la branche `Bool` : $\Phi(\texttt{a}) = \Phi(\texttt{bool}) = \{\texttt{bool}, \texttt{a}\}$.

Généralisation

Généralisation (inférence) :

- Rendre universelle des variables de types jusqu'ici existentielles
- Critère de généralisation d'OCaml : restriction relâchée de valeurs.

Généralisation (vérification) :

- Vérifier que des variables universelles étaient généralisables
- Utilisation du critère de non expansivité d'OCaml et de la variance calculée

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\text{Let}\ \frac{}{\Gamma, \Phi \vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\forall i.\ \Gamma, \Phi, \emptyset \left|\!\!\vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i\right.$$

$$\text{Let}\ \frac{}{\Gamma, \Phi \left|\!\!\vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle\right.}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i$$
$$let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}})$$

$$\text{Let}\ \frac{}{\Gamma, \Phi \vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i$$

$$let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_\exists = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_n}})$$

$$\text{Let}\ \frac{}{\Gamma, \Phi \vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\frac{\begin{array}{c}\forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_{\exists} = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_n}}) \\ \forall i.\ \Gamma, \Phi \vdash \langle e_i : \tau_i \rangle\end{array}}{\Gamma, \Phi \vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle}\ \text{Let}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\forall i.\ \Gamma, \Phi, \emptyset \Big|_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i$$

$$let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_\exists = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_n}})$$

$$\forall i.\ \Gamma, \Phi \Big| \langle e_i : \tau_i \rangle \qquad \forall i.\ \Gamma, \Phi, \theta_{init}(\Gamma) \Big|_{\tau} \tau_i \leq \sigma_i \Rightarrow \theta_i$$

$$\text{Let}\ \frac{}{\Gamma, \Phi \Big| \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\frac{\begin{array}{c}\forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_{\exists} = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_n}}) \\ \forall i.\ \Gamma, \Phi \vdash \langle e_i : \tau_i \rangle \qquad \forall i.\ \Gamma, \Phi, \theta_{init}(\Gamma) \vdash_{\tau} \tau_i \leq \sigma_i \Rightarrow \theta_i \\ \forall i.\ check\_gen(\Gamma, \Phi, \sigma_i, e_i)\end{array}}{\Gamma, \Phi \vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle}\ \text{Let}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\text{Let}\ \frac{\begin{array}{c}\forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_\exists = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_n}}) \\ \forall i.\ \Gamma, \Phi \vdash \langle e_i : \tau_i \rangle \qquad \forall i.\ \Gamma, \Phi, \theta_{init}(\Gamma) \vdash_{\tau} \tau_i \leq \sigma_i \Rightarrow \theta_i \\ \forall i.\ check\_gen(\Gamma, \Phi, \sigma_i, e_i) \qquad let\ \Gamma' = \Gamma \oplus_{\mathcal{V}} \mathcal{V}_p \oplus_{\mathcal{T}} \mathcal{T}_\exists\end{array}}{\Gamma, \Phi \vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\text{Let}\ \frac{\begin{array}{c}\forall i.\ \Gamma, \Phi, \emptyset \left|\!\vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_\exists = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_n}}) \\ \forall i.\ \Gamma, \Phi \left|\!\vdash \langle e_i : \tau_i \rangle \right. \qquad \forall i.\ \Gamma, \Phi, \theta_{init}(\Gamma) \left|\!\vdash_{\tau} \tau_i \leq \sigma_i \Rightarrow \theta_i \right. \\ \forall i.\ check\_gen(\Gamma, \Phi, \sigma_i, e_i) \qquad let\ \Gamma' = \Gamma \oplus_{\mathcal{V}} \mathcal{V}_p \oplus_{\mathcal{T}} \mathcal{T}_\exists \\ \Gamma', \Phi_n \left|\!\vdash \langle e' : \tau' \rangle \right.\end{array}}{\Gamma, \Phi \left|\!\vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle \right.}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\text{Let}\ \frac{\begin{array}{c}\forall i.\ \Gamma, \Phi, \emptyset \left|\!\vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_\exists = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_n}}) \\ \forall i.\ \Gamma, \Phi \left|\!\vdash \langle e_i : \tau_i \rangle \right. \qquad \forall i.\ \Gamma, \Phi, \theta_{init}(\Gamma) \left|\!\vdash_{\tau} \tau_i \leq \sigma_i \Rightarrow \theta_i \right. \\ \forall i.\ check\_gen(\Gamma, \Phi, \sigma_i, e_i) \qquad let\ \Gamma' = \Gamma \oplus_{\mathcal{V}} \mathcal{V}_p \oplus_{\mathcal{T}} \mathcal{T}_\exists \\ \Gamma', \Phi_n \left|\!\vdash \langle e' : \tau' \rangle \right. \qquad \Gamma, \Phi \left|\!\vdash_{\tau} \tau'\ wf \right.\end{array}}{\Gamma, \Phi \left|\!\vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle \right.}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\text{Let}\ \frac{\begin{array}{c}\forall i.\ \Gamma, \Phi, \emptyset \vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ let\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad let\ \mathcal{T}_\exists = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_n}}) \\ \forall i.\ \Gamma, \Phi \vdash \langle e_i : \tau_i \rangle \qquad \forall i.\ \Gamma, \Phi, \theta_{init}(\Gamma) \vdash_{\tau} \tau_i \leq \sigma_i \Rightarrow \theta_i \\ \forall i.\ check\_gen(\Gamma, \Phi, \sigma_i, e_i) \qquad let\ \Gamma' = \Gamma \oplus_{\mathcal{V}} \mathcal{V}_p \oplus_{\mathcal{T}} \mathcal{T}_\exists \\ \Gamma', \Phi_n \vdash \langle e' : \tau' \rangle \qquad \Gamma, \Phi \vdash_{\tau} \tau'\ wf \qquad \Gamma', \Phi_n \vdash_{\tau} \tau \equiv \tau' \end{array}}{\Gamma, \Phi \vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle}$$

Vérification : système de types

Ces définitions permettent de vérifier le cœur d'OCaml. Exemple de vérification de `let` :

$$\text{Let}\ \frac{\begin{array}{c}\forall i.\ \Gamma, \Phi, \emptyset \left|\!\vdash_{\rho} \langle p_i : \sigma_i \rangle \Rightarrow (\overline{v_i : \sigma_{v_i}}), (\overline{\tau_{\exists_i}}), \Phi_i \\ \mathit{let}\ \mathcal{V}_p = (\overline{v_1 : \sigma_{v_1}}) \uplus .. \uplus (\overline{v_n : \sigma_{v_n}}) \qquad \mathit{let}\ \mathcal{T}_\exists = (\overline{\tau_{\exists_1}}) \uplus .. \uplus (\overline{\tau_{\exists_n}}) \\ \forall i.\ \Gamma, \Phi \left|\!\vdash \langle e_i : \tau_i \rangle \right. \qquad \forall i.\ \Gamma, \Phi, \theta_{init}(\Gamma) \left|\!\vdash_{\tau} \tau_i \leq \sigma_i \Rightarrow \theta_i \right. \\ \forall i.\ \mathit{check\_gen}(\Gamma, \Phi, \sigma_i, e_i) \qquad \mathit{let}\ \Gamma' = \Gamma \oplus_{\mathcal{V}} \mathcal{V}_p \oplus_{\mathcal{T}} \mathcal{T}_\exists \\ \Gamma', \Phi_n \left|\!\vdash \langle e' : \tau' \rangle \right. \qquad \Gamma, \Phi \left|\!\vdash_{\tau} \tau'\ \mathit{wf} \right. \qquad \Gamma', \Phi_n \left|\!\vdash_{\tau} \tau \equiv \tau' \right.\end{array}}{\Gamma, \Phi \left|\!\vdash \langle \texttt{let}\ \overline{p = e}\ \ \texttt{in}\ e' : \tau \rangle \right.}$$

Système de types exécutable

Spécification décrite de manière exécutable :
⇒ Implémentation en OCaml d'un vérificateur, pour OCaml 4.02.
Vérifie :

- Langage de base - *Formalisé*
  - noyau fonctionnel
  - extensions impératives
  - extensions fonctionnelles (dont Variants polymorphes et GADTs)
- Modules
- Objets

Système de types exécutable

Spécification décrite de manière exécutable :
⇒ Implémentation en OCaml d'un vérificateur, pour OCaml 4.02.
Vérifie :

- Langage de base - *Formalisé*
  - noyau fonctionnel
  - extensions impératives
  - extensions fonctionnelles (dont Variants polymorphes et GADTs)
- Modules
- Objets

mais pas :

- Abbréviations récursives (`type t = int * t`) (`-rectypes`)
- Module récursifs (`module rec M = .. M ..`)
- Variance (`type -'a = ..`)

Implémentation

Quelques données :

- Implémentation purement fonctionnelle

Implémentation

Quelques données :

- Implémentation purement fonctionnelle
- $\sim$ **5 kLoC**, contre $\sim$ **12.5 kLoC** pour typer le même sous-ensemble dans le compilateur
  $\rightarrow$ Mais, la vérification plus simple que l'inférence

Implémentation

Quelques données :

- Implémentation purement fonctionnelle
- $\sim$ **5 kLoC**, contre $\sim$ **12.5 kLoC** pour typer le même sous-ensemble dans le compilateur
  $\rightarrow$ Mais, la vérification plus simple que l'inférence
- Pas de partage avec le compilateur autre que les structures de données

Implémentation

Quelques données :

- Implémentation purement fonctionnelle
- $\sim$ **5 kLoC**, contre $\sim$ **12.5 kLoC** pour typer le même sous-ensemble dans le compilateur
  $\rightarrow$ Mais, la vérification plus simple que l'inférence
- Pas de partage avec le compilateur autre que les structures de données
- 4.02 : capable de vérifier certains programmes provoquant une erreur de l'inférence

Implémentation

Quelques données :

- Implémentation purement fonctionnelle
- $\sim$ **5 kLoC**, contre $\sim$ **12.5 kLoC** pour typer le même sous-ensemble dans le compilateur
  $\rightarrow$ Mais, la vérification plus simple que l'inférence
- Pas de partage avec le compilateur autre que les structures de données
- 4.02 : capable de vérifier certains programmes provoquant une erreur de l'inférence
  - **MPR 7222** : échappement de types existentiels

Implémentation

Quelques données :

- Implémentation purement fonctionnelle
- $\sim$ **5 kLoC**, contre $\sim$ **12.5 kLoC** pour typer le même sous-ensemble dans le compilateur
  $\rightarrow$ Mais, la vérification plus simple que l'inférence
- Pas de partage avec le compilateur autre que les structures de données
- 4.02 : capable de vérifier certains programmes provoquant une erreur de l'inférence
  - **MPR 7222** : échappement de types existentiels
  - **MPR 6992** : interaction entre foncteurs et GADTs permettant des coercions non sûres

Sémantique opérationnelle du TypedTree

Formalisation d'une sémantique à petits pas pour le TypedTree

- Permettre (à l'avenir) une preuve de sûreté
- Sûreté : progrès et préservation

En particulier, la préservation :

```
⟨ let ⟨ f ⟨ x : 'a ⟩ : 'a -> 'a ⟩ = ⟨ x : 'a ⟩ in
⟨ ⟨ f : int -> int ⟩ ⟨ 0: int ⟩ : int ⟩ : int ⟩
```

Sémantique opérationnelle du TypedTree

Formalisation d'une sémantique à petits pas pour le TypedTree

- Permettre (à l'avenir) une preuve de sûreté
- Sûreté : progrès et préservation

En particulier, la préservation :

```
⟨ let ⟨ f ⟨ x : 'a ⟩ : 'a -> 'a ⟩ = ⟨ x : 'a ⟩ in
⟨ ⟨ f : int -> int ⟩ ⟨ 0: int ⟩ : int ⟩ : int ⟩
```

$\rightsquigarrow$

```
⟨ ⟨ fun ⟨ x : int ⟩ -> ⟨ x : int ⟩ : int -> int ⟩
    ⟨ 0: int ⟩ : int ⟩
```

Nécessité d'une *instanciation sémantique*

Formalisation Coq

MiniML : formalisation des règles et implémentation d'un vérificateur.

Engineering Formal Metatheory

(Aydemir, Charguéraud, Pierce, Pollack et Weirich, POPL '08)

Représentation des variables liées par une quantification cofinie
LibLN : bibliothèque implémentant les structures de données et dédiée à la preuve de langages.

Travaux existants sur ce formalisme

- Charguéraud : ML + références + exceptions + types algébriques
- Garrigue : ML + types récursives + variants polymorphes

Vérification de l'inférence : travaux connexes

Deux autres axes principaux :

- Langages intermédiaires typés :
  - Haskell : FC
  - SML : TIL et TILT
- Compilateurs certifiés :
  - CakeML (SML)
  - CompCert (C)

Perspectives

Vérification de l'inférence

- Formalisation et sémantique des modules et objets
- Preuve de sûreté

Spécification du système de types

- Langage intermédiaire typé (*Lambda typé*)
- Moteur d'inférence :
  Documentation et possible réécriture

Conclusion

Contributions :

- Découpage et formalisation inductive des propriétés de l'unification
- Spécification du langage de base du système de types d'OCaml 4.02
- Implémentation d'un vérificateur suivant cette spécification
  github.com/OCamlPro-Couderc/ocp-typechecker

Bug report 6992

```
type (_, _) eq = Eq : ('a, 'a) eq
let cast : type a b . (a, b) eq -> a -> b = fun Eq x
    -> x

module Fix (F : sig type 'a f end) = struct
  type 'a fix = ('a, 'a F.f) eq
  let uniq (type a) (type b) (Eq : a fix) (Eq : b
      fix) : (a, b) eq = Eq
end

module FixId = Fix (struct type 'a f = 'a end)
let bad : (int, string) eq = FixId.uniq Eq Eq
let _ = Printf.printf "Oh dear: %s" (cast bad 42)
```

Bug report 7222

```
type 'a n = N
type nil = private Nil_type
type (_,_) elt = Elt: 'nat n -> ('l,'nat -> 'l) elt
type _ t = Nil : nil t | Cons : ('x, 'fx) elt * 'x t
     -> 'fx t

let undetected: ('a -> 'b -> nil) t -> 'a n -> 'b n
    -> unit = fun sh i j ->
  let Cons(Elt dim, _) = sh
  in ()

undetected (Cons(Elt N,Cons(Elt N,Nil))) N N
```