Traduction d'OCaml vers une variante de Système F

Jonathan Protzenko
sous la direction de François Pottier

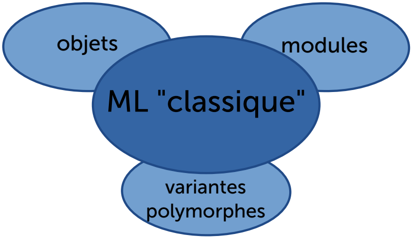
June 23, 2010

Plan

Plan

Pourquoi traduire?

- On veut augmenter la confiance dans la chaîne de compilation
- "Well-typed programs can't go wrong" (Milner)
- Le système de types d'OCaml est trop complexe

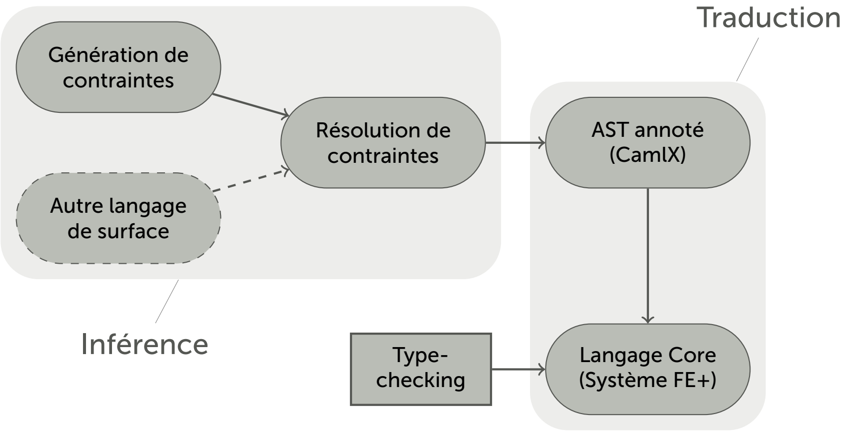


- Traduire le programme dans un langage de base
- Vérifier le typage *a posteriori*

Objectifs à long terme

- Fournir un langage intermédiaire pour *effectuer des analyses* et compiler plus en avant: expressions *simples*, informations de type *riches*.
- Augmenter la confiance dans la chaîne de compilation: à défaut de prouver la correction du typeur, prouver la cohérence de ses résultats.
- Clarifier la sémantique du langage original: quelles sont les constructions qui s'expriment bien dans FE+?

Dans les grandes lignes...

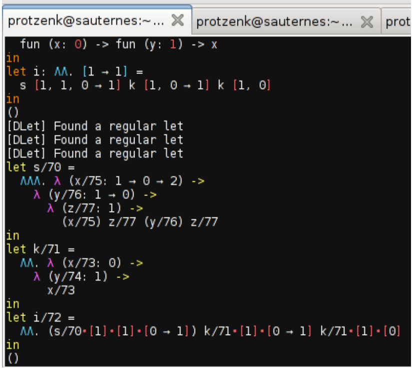


Le processus se découpe en deux parties : génération/résolution de contraintes, et traductions jusqu'à Système FE+.

Plan

Trois grands axes de travail

- Récrire un système d'inférence par contraintes (générateur de contraintes *et* solveur), et l'adapter pour donner un *AST annoté*.
- Élaborer un processus de traduction d'un fragment d'OCaml vers un langage minimaliste



  - Concevoir le système de types qui permet de justifier le comportement d'OCaml

Trois langages

Les langages suivants sont utilisés:

- CamlX « à trous » (OCaml annoté)
- CamlX « tout court » (OCaml annoté en De Bruijn)
- Système FE+

Système FE+ est Système F_η augmenté avec des coercions explicites, des expressions supplémentaires (tuples), des patterns...

Plan

Plan

1. Introduction
2. Décoration d'ASTs
   - Le cœur du problème
   - Garder les annotations à portée de main
3. Traduction(s)
4. Système F plus coercions
5. Conclusion

L'inférence par contraintes

- Présentation revue d'un algorithme « classique » (Pottier, Rémy, 2005)
- Séparation claire et élégante entre génération et résolution
- Préférable à l'implémentation OCaml, performante mais difficile d'accès

Exemple:

$$[\![\lambda z.t : T]\!] \quad = \quad \exists X_1 X_2.\,(\text{let } z : X_1 \text{ in } [\![t : X_2]\!] \wedge X_1 \rightarrow X_2 = T)$$

Que fait l'inférence par contraintes?

L'inférence par contraintes répond oui ou non.
Au mieux, affiche les types inférés des définitions top-level.

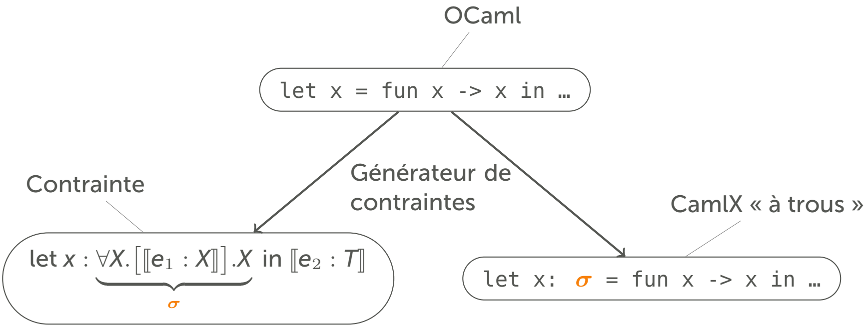
```
let (x, y) = (fun x -> x) (1, fun x -> x)

val x: int
val y: ∀ α. α → α
```

Comment l'adapter pour afficher un AST annoté?

(Pas de value restriction dans les exemples)

Plan

Une méthode ad-hoc



Idée: le générateur de contraintes renvoie *deux* arbres qui *partagent* des structures σ décrivant les schémas de type.

Fonctionnement de cette méthode

Le générateur de contraintes et le solveur ont dû être récrits.

- Le générateur de contraintes *pré-alloue* des « boîtes vides » correspondant aux futurs résultats du solveur de contraintes.
- Le solveur résout la contrainte, et remplit au passage les boîtes.
- Les boîtes sont partagées: après la résolution des contraintes, les trous sont remplis, et l'AST « CamlX » est désormais annoté.

Remplissage des boîtes

Système F annote les variables introduites par un λ. Il faut aussi conserver les types utilisés pour instancier les schémas.

Les schémas sont résolus au niveau des contraintes `let`.

`function`, `fun`, `let`, `match` génèrent une contrainte `let`. Procédure :

- créer une boîte ;
- l'attacher à la contrainte `let` ;
- l'attacher au nœud CamlX correspondant.

(idem pour les instances de schémas)

Avec un peu de recul...

- Simple et efficace : il s'agit de « faire suivre » les informations nécessaires.
- Facile à implémenter : solution d'une remarquable flexibilité.
- Peu élégant : le contenu des boîtes expose les structures internes du solveur.
- Pas de scope: les schémas de type sont extrudés, sortis de leur contexte.
- Formalisation difficile (la formalisation correcte est connue, mais n'est pas bonne pour une implémentation).

Il faudrait arriver à une forme plus propre et plus propice aux transformations...

Plan

Plan

De CamlX « à trous »...

... vers CamlX « tout court »

- Se passer des champs mutables et des classes d'équivalence
- Types en indices de De Bruijn
- Ne pas changer les expressions, simplement les types

```
let (x, y): ∀. [int * (0 → 0)] = Λ.
  (fun (x: int * (0 → 0)) -> x)
  (1, (fun (x: 0) -> x))
in
()
```

Nettoyage des boîtes

Les structures union-find sont mutables et contiennent du partage. On utilise des types avec des indices de De Bruijn.

- La généralisation se fait au niveau des `let` : pas de nœud Λ dans la syntaxe des expressions ;
- L'application de type se fait au niveau des instanciations : pas de nœud « application de type » ;
- les patterns sont présents dans les `let` et `function` ;
- les `let` sont multiples.

CamlX

Une représentation avec des types clairs mais des expressions complexes.

Plan

Désucrer CamlX vers...

... Système FE+

```
match
  Λ. (λ (x: int * (0 → 0)) -> x)
     (1, (λ (x: 0) -> x))
with
  | (x, y) ▸ ∀×; ×0[•[bottom]] ->
      ()
```

Liste des modifications

- Les patterns sont utilisés à de nombreux endroits en OCaml: on les restreint aux `match` uniquement.
- Les `let and` peuvent définir simultanément plusieurs motifs: on utilise des identifiants uniques pour s'en passer.
- Les Λ et les applications de types deviennent des nœuds normaux de la syntaxe des expressions.

... et surtout, sont ajoutées des coercions.

Plan

Plan

Aperçu du problème

Reprenons l'exemple initial.

```
let (x, y) = (fun x -> x) (1, fun x -> x)
```

OCaml répond:

```
val x: int
val y: 'a -> 'a
```

Comprendre:

```
val x: int
val y: ∀ α. α → α
```

Comment justifier ce jugement de typage ?

Où le type-checking échoue

À gauche	À droite
(x, y)	$\forall\alpha.\ (\mathsf{int} \times \alpha \to \alpha)$
`PCons(Tuple, [x; y])`	`Forall (TCons (Tuple, [int; …]))`

Head symbol mismatch: type $\forall\alpha.\ (\mathsf{int}, \alpha \to \alpha)$ does not match pattern: Tuple…

(c'est une erreur de type)

Du sous-typage

Solution: Système F_η, ou « Système F avec sous-typage ».

- Faire rentrer le quantificateur :
  $\forall\alpha.\ (\mathsf{int} \times \alpha \to \alpha) \leq (\forall\alpha.\ \mathsf{int} \times \forall\alpha.\ \alpha \to \alpha)$.
- Éliminer dans la première branche :
  $(\forall\alpha.\ \mathsf{int} \times \forall\alpha.\ \alpha \to \alpha) \leq (\mathsf{int} \times \forall\alpha.\ \alpha \to \alpha)$.

On attache des coercions aux motifs :

$\boxed{p \blacktriangleright c\text{: avant de filtrer sur } p\text{, appliquer la coercion } c}$

Qu'est-ce qu'une coercion?

Une coercion est un témoin de sous-typage.

$$\frac{\Gamma \vdash c : \tau_i \leq \tau_i'}{\Gamma \vdash (\times_i[c]) : (\tau_1, \ldots, \tau_i, \ldots, \tau_n) \leq (\tau_1, \ldots, \tau_i', \ldots, \tau_n)} \quad \text{(projection-tuple)}$$

$$\frac{\Gamma \vdash c : \tau \leq \tau'}{\Gamma \vdash (\forall[c]) : \forall\tau \leq \forall\tau'} \quad \text{(}\forall\text{-covariance)}$$

$$\frac{\Gamma \vdash c_1 : \tau \leq \tau' \qquad \Gamma \vdash c_2 : \tau' \leq \tau''}{\Gamma \vdash c_1; c_2 : \tau \leq \tau''} \quad \text{(transitivité)}$$

$$\frac{}{\Gamma \vdash \bullet[\sigma] : \forall\tau \leq [\sigma/0]\tau} \quad \text{(}\forall\text{-élimination)}$$

$$\frac{}{\Gamma \vdash (\forall\times) : \forall(\tau_1, \ldots, \tau_n) \leq (\forall\tau_1, \ldots, \forall\tau_n)} \quad \text{(}\forall\text{-distributivité)}$$

Plan

1. Introduction
2. Décoration d'ASTs
3. Traduction(s)
4. Système F plus coercions
   - Présentation des coercions
   - Règles de génération de coercions
5. Conclusion

Vu de loin

La relation de sous-typage dans F_η est indécidable mais...

- on ne demande jamais: $\tau \leq \tau'$?,
- on génère des coercions *au préalable*,
- on sait toujours *comment* coercer et vers quel type,
- on opère sur un sous-ensemble des situations de sous-typage.

Comment générer les coercions ?

Un ensemble de règles statiques permet de générer les coercions adaptées à chaque situation, *par filtrage sur le type et le pattern*.

- Rentrer récursivement les $\forall$ dans les tuples, grâce à $(\forall\times)$ et $(\forall[\bullet])$.
- Éliminer les quantifications $\forall$ inutiles pour les identifiants grâce à $(\bullet[\bot])$ et $(\forall[\bullet])$.

Au moment du type-checking, on a une fonction `apply_coercion: typ -> coercion -> typ`.

(exemple au tableau)

L'exemple initial

```
let (x, y) = (fun x -> x) (1, fun x -> x)
```

... devient

```
match
  Λ. (λ (x: int * (0 → 0)) -> x)
     (1, (λ (x: 0) -> x))
with
  | (x, y) ▸ ∀×; ×0[•[bottom]] ->
      ()
```

Phase finale

On vérifie le langage « core »...

... et on répond toujours oui (ouf !).

Le typeur est court (~200 lignes de code), et a été terminé entre le rapport et la présentation.

Plan

Plan

Une belle implémentation

199	`chaml/chaml.ml`
184	`chaml/algebra.ml`
601	`chaml/oCamlConstraintGenerator.ml`
183	`chaml/constraint.ml`
332	`chaml/solver.ml`
402	`chaml/unify.ml`
346	`chaml/translator.ml`
495	`chaml/desugar.ml`
37	`chaml/atom.ml`
74	`chaml/deBruijn.ml`
224	`chaml/typeCheck.ml`
156	`chaml/typePrinter.ml`
473	`stdlib/*.ml`
500	`tests/run_tests.ml`
...	...
6000+	total (avec les `.mli`)

Exemple de sortie de ChaML

Batterie de tests : une trentaine d'expressions ML mettant à l'épreuve les constructions reconnues par le programme. Comparaison de la sortie du solveur avec OCaml + type-checking du programme Core.

```
[Driver] 427 nodes in the OCaml ast
[Driver] 957 nodes in the constraint
[Driver] 915 nodes in the CamlX AST
[Driver] 666 nodes in the Core AST
```

Beaucoup d'annotations explicites de type en CamlX (pour construire les coercions). Le langage Core est plus léger.

Par rapport aux objectifs originaux...

Sont actuellement traduits :

- ML de base,
- patterns généralisants,
- patterns en profondeur.

Trouver le bon langage cible et gérer les coercions a occupé une bonne partie du temps !

Plan

Une expérience intéressante

- Poser les bases a été le plus difficile
- Dans le pipeline : value restriction, types algébriques (sans modifications du langage cible !)
- Plus tard : variantes polymorphes (qu'ajouter dans le langage de base ?), autres...

Peut-être beaucoup d'implémentation à venir.

Un *framework* pour l'inférence de types

- Adapter l'inférence par contraintes a été un gros travail
- ... en faire une bibliothèque d'inférence à la ML?
- Offrir une interface claire pour examiner les résultats du solveur.

Questions ?