Vers un langage typé pour la programmation modulaire

UMONS

23 juin 2017

Introduction - OCaml

OCaml

Commençons par parler d'OCaml...

- langage fonctionnel : fonctions sont des citoyens de première classe et ce « nativement ».
- statiquement typé : les types sont assignés à la compilation, non à l'exécution. $\neq$ dynamiquement typé (Python).
- dispose d'algorithmes d'inférence de type : pas obligé d'assigner manuellement un type à un terme (contrairement à C ou Java).
- divisé en deux « langages » :
  - langage de base : fonctions, enregistrements, tuples, ...
  - langage de modules : modules (type = signature), foncteurs.
- les deux langages possèdent des algorithmes d'inférence de types différents.

But du mémoire

- Définir un calcul théorique où les deux langages (et en particulier les règles de typage et de sous-typage) sont unifiés. Calcul choisi : DOT (2016).
- Unification de la notion de module et d'enregistrement. Ces deux termes et leurs types sont représentés de la même manière ($\neq$ en OCaml).
- (Apport principal de ce mémoire). Implémenter un langage de surface, proche d'OCaml, un algorithme de typage et un algorithme de sous-typage pour écrire des programmes DOT. Langage : RML

Programmation fonctionnelle et typage

λ-calcul non typé

Calcul = syntaxe (termes) + sémantique (règles d'évaluation, non discutée).
Soit V un ensemble infini dénombrable de variables. La syntaxe du λ-calcul est définie comme le plus petit ensemble Λ tel que

1. $V \subseteq \Lambda$
2. $\forall u, v \in \Lambda, uv \in \Lambda$
3. $\forall x \in V, \forall u \in \Lambda, \lambda x.\, u \in \Lambda$

$\Leftrightarrow$

$t ::=$		terme
	x	var
	$t\ t$	app
	$\lambda x.\, t$	abs

- $uv \rightarrow$ application d'une fonction u à un paramètre v.
- $\lambda x.\, u \rightarrow$ définition d'une fonction (anonyme) qui prend un paramètre x et renvoie u.

λ-calcul simplement typé

- Classer les termes en fonction de leur nature $\rightarrow$ notion de type.
- Exemple pour le terme uv : u doit être moralement une fonction qui prend un paramètre v d'un certain type T_1. On note $T_1 \rightarrow T_2$ le type de u (type fléche) où T_2 est le type de retour de u.
- Formellement, on se donne τ, un ensemble de type, les éléments étant notés T et on définit une relation de typage entre Λ et τ. On note $t : T$ pour dire que t est de type T. On dit aussi que t est bien typé.

Syntaxe des types du λ-calcul simplement typé

On se donne $\mathcal{B}$ un ensemble de types de base, ses éléments étant notés B et on définit la syntaxe des types du λ-calcul simplement typé par

$T ::=$		types
	B	base
	$T \to T$	type des fonctions

Contexte et jugement de typage, différence de syntaxe

- $\lambda x. t$: type de x ? $\Rightarrow$ on annote la variable avec son type. $\lambda x. t$ devient $\lambda x : T. t$.
- un terme t peut contenir des variables $\Rightarrow$ besoin de connaitre le type de celles-ci pour typer t $\Rightarrow$ contexte de typage Γ.
- Un **contexte (ou environnement) de typage** est une suite (x_i, T_i) où x_i est une variable et T_i est le type de x_i, noté Γ. L'union d'un contexte Γ avec un couple (x, T) est noté $\Gamma, x : T$.
- $t : T$ devient $\Gamma \vdash t : T$ (appelé **jugement de typage**).

Règles de typage

Pour typer un terme t, on utilise des **règles de typage**. Une règle de typage permet de dériver des jugements de typage à partir d'autres jugements de typage ou d'axiome. Une succession d'utilisation de règles de typage mène à **un arbre de dérivation**.

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 .\, t : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash u : T_1 \to T_2 \qquad \Gamma \vdash v : T_1}{\Gamma \vdash u\, v : T_2} \quad \text{(T-APP)}$$

λ-calcul simplement typé

Théorèmes importants qui permettent d'affirmer « un terme bien typé ne bloque pas ».

1. Préservation : si un terme t est de type T et t s'évalue en t', alors t' est de type T.
2. Progression : si un terme t est de type T, alors soit t est une valeur, soit t s'évalue en t'.

Ces théorèmes sont vrais pour tous les calculs définis dans ces transparents. Les preuves se font le plus souvent sur l'arbre de dérivation.

Sous-typage et enregistrement

- But : affiner notre relation de typage en définissant une relation binaire (dite de **sous-typage**) sur les types.
- Notation : $S <: T$ (S est sous-type de T).
- Règle de typage qui crée un lien entre la relation de typage et la relation de sous-typage : (T-SUB).

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

DOT

$t ::=$	terme
x, y	var
$\lambda x : T.\, t$	abs
$x\, y$	app
$let\ x\ =\ t\ in\ t$	let
$\nu(x : T^x)d$	rec
$x.a$	champ proj

$d ::=$	decl
$\{a = t\}$	champ
$\{A = T\}$	type
$d \wedge d$	aggregation

$S, T ::=$		type
	Top	top
	$Bottom$	bottom
	$\forall(x : S)T^x$	fonction
	$\{A : S..T\}$	type decl
	$\{a : T\}$	champ decl
	$x.A$	type proj
	$\mu(x : T^x)$	rec
	$S \wedge T$	inter

DOT : règles typage (1/3)

$$\Gamma, x : T, \Gamma' \vdash x : T \quad \text{(T-VAR)} \qquad \frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

$$\frac{\Gamma, x : T \vdash t : U \qquad x \notin FV(T)}{\Gamma \vdash \lambda x : T.\, t : \forall(x : T) U^x} \quad \text{(ALL-I)}$$

$$\frac{\Gamma \vdash x : \forall(z : S) T^z \qquad \Gamma \vdash y : S}{\Gamma \vdash x\ y : [z := y] T^z} \quad \text{(ALL-E)}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma, x : T \vdash u : U \qquad x \notin FV(U)}{\Gamma \vdash let\ x\ =\ t\ in\ u : U} \quad \text{(LET)}$$

DOT : règles de typage (2/3)

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U} \quad \text{(AND-I)}$$

$$\frac{\Gamma, x : T \vdash d : T}{\Gamma \vdash \nu(x : T^x)d : \mu(x : T^x)} \quad (\{\ \}\text{-I})$$

$$\frac{\Gamma \vdash x : T^x}{\Gamma \vdash x : \mu(z : T^z)} \quad \text{(VAR-PACK)}$$

$$\frac{\Gamma \vdash x : \mu(z : T^z)}{\Gamma \vdash x : T^x} \quad \text{(VAR-UNPACK)}$$

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T} \quad \text{(FLD-E)}$$

DOT : règles de typage (3/3)

$$\Gamma \vdash \{A = T\} : \{A : T..T\} \quad \text{(TYP-I)}$$

$$\frac{\Gamma \vdash d_1 : T_1 \qquad \Gamma \vdash d_2 : T_2 \qquad dom(d_1) \cap dom(d_2) = \emptyset}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \quad \text{(ANDDEF-I)}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a : t\} : \{a : T\}} \quad \text{(FLD-I)}$$

DOT : règles de sous-typage (1/2)

$$\Gamma \vdash T <: \mathit{Top} \quad \text{(S-TOP)} \qquad \Gamma \vdash \mathit{Bottom} <: T \quad \text{(S-Bottom)}$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \quad \text{(S-TRANS)} \qquad \Gamma \vdash T <: T \quad \text{(S-REFL)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : S_1)\, T_1 <: \forall(x : S_2)\, T_2} \quad \text{(ALL< :ALL)}$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \quad \text{(< : SEL)}$$

DOT : règles de sous-typage (2/2)

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \quad (\text{SEL} <:) \qquad \Gamma \vdash T \wedge U <: T \quad (\text{AND-1-}<:)$$

$$\Gamma \vdash T \wedge U <: U \quad (\text{AND-2-}<:)$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \quad (<: \text{AND})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}} \quad (\text{TYP}<:\text{TYP})$$

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a : T\} <: \{a : U\}} \quad (\text{FLD} <: \text{FLD})$$

Implémentation de DOT en OCaml : RML

Implémentation : remarques

1. Aucun papier ne décrit l'implémentation d'algorithmes de typage et de sous-typage pour DOT.
2. Besoin de définir un langage de surface (= un langage plus haut niveau et plus facile à lire que le calcul théorique).
3. RML implémente des sucres syntaxiques pour écrire plus facilement des programmes.
4. RML implémente un algorithme d'inférence de type pour les modules. Par défaut, la définition d'un module est obligatoirement liée à une signature.
5. Plusieurs problèmes détectés lors de l'implémentation.

Implémentation : pas si facile...

Passer de la théorie à l'implémentation n'est pas facile !

1. Gestion des variables libres et liées (utilisation d'AlphaLib).
2. Un même jugement de typage peut être dérivé de plusieurs manières.
3. Une même question $S <: T$ peut être posée plusieurs fois (ex : liste avec tail).
4. Indécidabilité du sous-typage.
5. Le sous-typage dépend du typage à travers (SEL < :) et (< : SEL). **Pas usuel.**

Implémentation : problèmes

Problèmes rencontrés lors de l'implémentation et non décrits dans les différents papiers :

1. DOT n'est pas stable par insertion de binding locaux variable-variable.
2. Il est nécessaire d'implémenter un algorithme intermédiaire (best_bound) dans le cas où une question $x.t <: y.t$ est posée. Ce dernier n'est pas évident !