Programmation Avancée

Sous-typage sémantique

David Baelde

ENS Paris-Saclay, L3 2020–2021

Sous-typage

Rappel

Un système de type peut être étendu au moyen d'une relation de sous-typage sur divers types :

- produits, enregistrements, variants, polymorphisme...

Le type τ est un sous-type de τ' (i.e. $\tau \leq \tau'$) quand :

- Les valeurs de type τ sont aussi des valeurs de type τ'.
- Remplacer une valeur de type τ' par une valeur de type τ ne peut pas provoquer d'erreur à l'exécution.

Aujourd'hui, nous allons (entre)voir que

il est possible et profitable de prendre la première intuition au sérieux.

Programmer avec des types ensemblistes

Dans le langage Flow :

```
function toStringPrimitives(val: number | boolean | string) {
  return String(val);
}
type One = { foo: number };
type Two = { bar: boolean };
var value: One & Two = {foo: 1,bar: true};
```

En Typed Racket :

```
(let ([a-number 37]) (if (even? a-number) 'yes 'no))
- : Symbol [more precisely: (U 'no 'yes)]
'no

(: f : (case-> (-> True Integer Integer) (-> False Boolean Boolean)))
(define (f condition x) (if condition (add1 x) (not x)))
```

Programmer avec des types ensemblistes

Dans une extension imaginable d'OCaml :

```
let balance = function
  | `Black, z, `Node (`Red, y, `Node (`Red, x, a, b), c), d
  | `Black, z, `Node (`Red, x, a, `Node (`Red, y, b, c)), d
  | `Black, x, a, `Node (`Red, z, `Node (`Red, y, b, c), d)
  | `Black, x, a, `Node (`Red, y, b, `Node (`Red, z, c, d)) ->
      `Node (`Red, y, `Node (`Black, x, a, b), `Node (`Black, z, c, d))
  | a, b, c, d ->
      `Node (a, b, c, d)

val balance :
    (`Black * 'a * 'a dirty * 'a tree) -> 'a tree
  & (`Black * 'a * 'a tree * 'a dirty) -> 'a tree
  & (`Red * 'a * 'a rbtree * 'a tree) -> 'a dirty
  & (`Red * 'a * 'a tree * 'a rbtree) -> 'a dirty
```

Exercice

Considérer la fonction suivante :

```
let foo = function (`A,`B) -> true | (`B,`A) -> false
```

Questions :

1. Quel type sera inféré par OCaml ?
2. Peut-on exprimer un type plus précis avec des types ensemblistes ?
3. Que dit OCaml sur cette définition ?

Typage précis du filtrage

Dans notre extension ensembliste d'OCaml, comment typer l'expression suivante ?

```
match u with p_1 as x_1 -> v_1 | p_2 as x_2 -> v_2
```

Premier essai :

$$\frac{\Gamma \vdash u : T \quad \Gamma, x_1 : T \vdash v_1 : T_1 \quad \Gamma, x_2 : T \vdash v_2 : T_2}{\Gamma \vdash e : T_1 \cup T_2}$$

Typage précis du filtrage

Dans notre extension ensembliste d'OCaml, comment typer l'expression suivante ?

```
match u with p_1 as x_1 -> v_1 | p_2 as x_2 -> v_2
```

Premier essai :

$$\frac{\Gamma \vdash u : T \quad \Gamma, x_1 : T \vdash v_1 : T_1 \quad \Gamma, x_2 : T \vdash v_2 : T_2}{\Gamma \vdash e : T_1 \cup T_2}$$

Intégration de la contrainte d'exhaustivité, avec P_i le type des expressions capturées par p_i :

$$\frac{\Gamma \vdash u : P_1 \cup P_2 \quad \Gamma, x_1 : P_1 \vdash v_1 : T_1 \quad \Gamma, x_2 : P_2 \vdash v_2 : T_2}{\Gamma \vdash e : T_1 \cup T_2}$$

Typage précis du filtrage

Dans notre extension ensembliste d'OCaml, comment typer l'expression suivante ?

```
match u with p_1 as x_1 -> v_1 | p_2 as x_2 -> v_2
```

Premier essai :

$$\frac{\Gamma \vdash u : T \quad \Gamma, x_1 : T \vdash v_1 : T_1 \quad \Gamma, x_2 : T \vdash v_2 : T_2}{\Gamma \vdash e : T_1 \cup T_2}$$

Intégration de la contrainte d'exhaustivité, avec P_i le type des expressions capturées par p_i :

$$\frac{\Gamma \vdash u : P_1 \cup P_2 \quad \Gamma, x_1 : P_1 \vdash v_1 : T_1 \quad \Gamma, x_2 : P_2 \vdash v_2 : T_2}{\Gamma \vdash e : T_1 \cup T_2}$$

Encore mieux, et plus algorithmique :

$$\frac{\Gamma \vdash u : T \quad \Gamma, x_1 : T \cap P_1 \vdash v_1 : T_1 \quad \Gamma, x_2 : (T \setminus P_1) \cap P_2 \vdash v_2 : T_2 \quad T \setminus (P_1 \cup P_2) \leq \emptyset}{\Gamma \vdash e : T_1 \cup T_2}$$

Conception d'un système de types ensemblistes

Tentatives de règles de typage :

$$\frac{\Gamma \vdash u : T_1 \quad \Gamma \vdash u : T_2}{\Gamma \vdash u : T_1 \cap T_2} \qquad \frac{\Gamma \vdash u : T_1 \cap T_2}{\Gamma \vdash u : T_i} \qquad \frac{\Gamma \vdash u : T_i}{\Gamma \vdash u : T_1 \cup T_2} \qquad \cdots$$

Et de sous-typage :

$$\frac{}{T_1 \cap T_2 \leq T_i} \qquad \frac{}{T_i \leq T_1 \cup T_2} \qquad \frac{T_1 \leq T_1' \quad T_2 \leq T_2'}{T_1 \cup T_2 \leq T_1' \cup T_2'} \qquad \cdots$$

Exemple

Les types $(T_1 \cup T_2) \to T'$ et $(T_1 \to T') \cap (T_2 \to T')$ devraient être équivalents.

Difficile de déterminer un jeu de règles complet... par rapport à quoi ?
On veut des algorithmes de typage aux résultats compréhensibles par un programmeur λ.

Sous-typage sémantique

A. Frisch, G. Castagna, V. Benzaken. *Semantic subtyping : Dealing set-theoretically with function, union, intersection, and negation types*. Journal of the ACM, 2008. [PDF]

Prendre au sérieux la première intuition :

$$T \leq T' \text{ ssi } [\![T]\!] \subseteq [\![T']\!] \qquad \text{où} \qquad [\![T]\!] \text{ est l'ensemble des valeurs de type } T$$

Problème

- Le typage s'appuie sur le sous-typage,
  qui s'appuie sur l'interprétation ensembliste $[\![\_]\!]$,
  qui s'appuie sur la notion de valeur bien typée,
  etc.

Vers une solution sémantique

Essayons d'interpréter les types *from scratch*[1] :

- $[\![\mathsf{Bool}]\!] = \{\mathsf{true}, \mathsf{false}\}$, $[\![\mathsf{Nat}]\!] = \mathbb{Z}$, etc.
- $[\![T \cup T']\!] = [\![T]\!] \cup [\![T']\!]$, $[\![T \cap T']\!] = [\![T]\!] \cap [\![T']\!]$, $[\![\mathsf{Any}]\!] = \mathcal{V}$, etc.
- $[\![T_1 \times T_2]\!] = [\![T_1]\!] \times [\![T_2]\!]$, $[\![[\text{`}\ell_i : T_i]_i]\!] = \bigcup_i \{\text{`}\ell_i(v) \mid v \in [\![T_i]\!]\}$, etc.
- $[\![T \to T']\!]$ est l'ensemble des fonctions qui,
  pour toute entrée dans $[\![T]\!]$, renvoient une valeur de $[\![T']\!]$.

1. Hypothèse : on ne peut/veut pas s'appuyer sur la notion de programme.

Vers une solution sémantique

Essayons d'interpréter les types *from scratch*[1] :

- $[\![\mathsf{Bool}]\!] = \{\mathsf{true}, \mathsf{false}\}$, $[\![\mathsf{Nat}]\!] = \mathbb{Z}$, etc.
- $[\![T \cup T']\!] = [\![T]\!] \cup [\![T']\!]$, $[\![T \cap T']\!] = [\![T]\!] \cap [\![T']\!]$, $[\![\mathsf{Any}]\!] = \mathcal{V}$, etc.
- $[\![T_1 \times T_2]\!] = [\![T_1]\!] \times [\![T_2]\!]$, $[\![[\text{`}\ell_i : T_i]_i]\!] = \bigcup_i \{\text{`}\ell_i(v) \mid v \in [\![T_i]\!]\}$, etc.
- $[\![T \to T']\!]$ est l'ensemble des fonctions qui,
  pour toute entrée dans $[\![T]\!]$, renvoient une valeur de $[\![T']\!]$.

L'ensemble $\mathcal{V}$ des valeurs devrait donc contenir les constantes, $\mathcal{V}^2$ et $\mathcal{V}^{\mathcal{V}}$!

1. Hypothèse : on ne peut/veut pas s'appuyer sur la notion de programme.

Bootstrap

Une solution de hackers :

Bootstrap

Une solution de hackers :

- Définir $[\![T]\!]_0$ avec un espace de valeurs $\mathcal{V}_0$ et une définition de $[\![T \to T']\!]_0$ ad-hoc.

Bootstrap

Une solution de hackers :

- Définir $[\![T]\!]_0$ avec un espace de valeurs $\mathcal{V}_0$ et une définition de $[\![T \to T']\!]_0$ ad-hoc.
- Définir le sous-typage à partir de cette sémantique : $T \leq_0 T'$ quand $[\![T]\!]_0 \subseteq [\![T']\!]_0$.

Bootstrap

Une solution de hackers :

- Définir $[\![T]\!]_0$ avec un espace de valeurs $\mathcal{V}_0$ et une définition de $[\![T \to T']\!]_0$ ad-hoc.
- Définir le sous-typage à partir de cette sémantique : $T \leq_0 T'$ quand $[\![T]\!]_0 \subseteq [\![T']\!]_0$.
- Une notion de typage $\Gamma \vdash_0 e : T$ en découle.

Bootstrap

Une solution de hackers :

- Définir $[\![T]\!]_0$ avec un espace de valeurs $\mathcal{V}_0$ et une définition de $[\![T \to T']\!]_0$ ad-hoc.
- Définir le sous-typage à partir de cette sémantique : $T \leq_0 T'$ quand $[\![T]\!]_0 \subseteq [\![T']\!]_0$.
- Une notion de typage $\Gamma \vdash_0 e : T$ en découle.
- Construire la sémantique naturelle qui en découle : $[\![T]\!]_1 = \{\ v \in \mathcal{V}\ \mid \emptyset \vdash_0 v : T\}$.

Bootstrap

Une solution de hackers :

- Définir $[\![T]\!]_0$ avec un espace de valeurs $\mathcal{V}_0$ et une définition de $[\![T \to T']\!]_0$ ad-hoc.
- Définir le sous-typage à partir de cette sémantique : $T \leq_0 T'$ quand $[\![T]\!]_0 \subseteq [\![T']\!]_0$.
- Une notion de typage $\Gamma \vdash_0 e : T$ en découle.
- Construire la sémantique naturelle qui en découle : $[\![T]\!]_1 = \{\ v \in \mathcal{V}\ \mid \emptyset \vdash_0 v : T\}$.
- Définir $\leq_1$, $\vdash_1$ et $[\![\_]\!]_2 \ldots$ jusqu'à où ?

Bootstrap

Une solution de hackers :

- Définir $[\![T]\!]_0$ avec un espace de valeurs $\mathcal{V}_0$ et une définition de $[\![T \to T']\!]_0$ ad-hoc.
- Définir le sous-typage à partir de cette sémantique : $T \leq_0 T'$ quand $[\![T]\!]_0 \subseteq [\![T']\!]_0$.
- Une notion de typage $\Gamma \vdash_0 e : T$ en découle.
- Construire la sémantique naturelle qui en découle : $[\![T]\!]_1 = \{\ v \in \mathcal{V}\ \mid \emptyset \vdash_0 v : T\}$.
- Définir $\leq_1$, $\vdash_1$ et $[\![\_]\!]_2 \ldots$ jusqu'à où ?

Le modèle $[\![T]\!]_0$ est dit universel quand

$$[\![T]\!]_0 \subseteq [\![T']\!]_0 \quad \Leftrightarrow \quad [\![T]\!]_1 \subseteq [\![T']\!]_1$$

i.e. les relations $\leq_0$ et $\leq_1$ coïncident, on a atteint un point fixe.

Un modèle universel

Prendre $\mathcal{V}$ contenant les constantes, clos par formation de paires et de parties finies.

$$[\![T \to T']\!] = \{f \subseteq_{\text{fin}} \mathcal{V}^2 \mid (v, v') \in f \text{ et } v \in [\![T]\!] \text{ implique } v' \in [\![T']\!]\}$$

Théorème (Frisch, Castagna & Benzaken)

Ce modèle est universel.

Un modèle universel

Prendre $\mathcal{V}$ contenant les constantes, clos par formation de paires et de parties finies.

$$[\![T \to T']\!] = \{f \subseteq_{\text{fin}} \mathcal{V}^2 \mid (v, v') \in f \text{ et } v \in [\![T]\!] \text{ implique } v' \in [\![T']\!]\}$$

Théorème (Frisch, Castagna & Benzaken)

Ce modèle est universel.

Exemple

- $\text{Nat} \to \text{Nat} \not\leq \text{Even} \to \text{Odd}$:
  $\{(0,0)\} \in [\![\text{Nat} \to \text{Nat}]\!]$ mais $\{(0,0)\} \notin [\![\text{Even} \to \text{Odd}]\!]$.
- $\text{Even} \to \text{Odd} \not\leq \text{Nat} \to \text{Nat}$:
  $\{(1, \text{`error'})\} \in [\![\text{Even} \to \text{Odd}]\!]$ mais $\{(1, \text{`error'})\} \notin [\![\text{Nat} \to \text{Nat}]\!]$.
- $\text{Unit} \to \text{Bool} \not\leq (\text{Unit} \to \text{True}) \cup (\text{Unit} \to \text{False})$:
  $\{((), \text{true}), ((), \text{false})\}$ n'appartient qu'à l'interprétation du premier type.

Décider le sous-typage

Cette définition sémantique n'exclut pas de décider $T \leq T'$.
C'est même plus élémentaire qu'avec une définition syntaxique du sous-typage ensembliste !

- On peut se ramener au test de vacuité : $T \leq T'$ ssi $T \setminus T' = T \cap \neg T' \leq \bot$.
- Par les lois de de Morgan et la distributivité,
  on se ramène à $\cup_i \cap_j T_{i,j} \leq \bot$, puis $\cap_k T'_k \leq \bot$.

Décider le sous-typage

Cette définition sémantique n'exclut pas de décider $T \leq T'$.
C'est même plus élémentaire qu'avec une définition syntaxique du sous-typage ensembliste !

- On peut se ramener au test de vacuité : $T \leq T'$ ssi $T \setminus T' = T \cap \neg T' \leq \bot$.
- Par les lois de de Morgan et la distributivité,
  on se ramène à $\cup_i \cap_j T_{i,j} \leq \bot$, puis $\cap_k T'_k \leq \bot$.
- Simplifier les cas mixtes :
  - $(T_1 \times T_2) \cap (T'_1 \to T'_2) \leq \bot$ toujours vrai.
  - $(T_1 \times T_2) \cap \neg(T'_1 \to T'_2) \leq \bot$ ssi $T_1 \times T_2 \leq \bot$.
  - etc.

Décider le sous-typage

Cette définition sémantique n'exclut pas de décider $T \leq T'$.
C'est même plus élémentaire qu'avec une définition syntaxique du sous-typage ensembliste !

- On peut se ramener au test de vacuité : $T \leq T'$ ssi $T \setminus T' = T \cap \neg T' \leq \bot$.
- Par les lois de de Morgan et la distributivité,
  on se ramène à $\cup_i \cap_j T_{i,j} \leq \bot$, puis $\cap_k T'_k \leq \bot$.
- Simplifier les cas mixtes :
  - $(T_1 \times T_2) \cap (T'_1 \to T'_2) \leq \bot$ toujours vrai.
  - $(T_1 \times T_2) \cap \neg(T'_1 \to T'_2) \leq \bot$ ssi $T_1 \times T_2 \leq \bot$.
  - etc.
- En exercice : traiter le cas $\bigcap_i T_i \times T'_i \leq \bigcup_j S_j \times S'_j$ et de même pour les flèches.

Décider le sous-typage

Cette définition sémantique n'exclut pas de décider $T \leq T'$.
C'est même plus élémentaire qu'avec une définition syntaxique du sous-typage ensembliste !

- On peut se ramener au test de vacuité : $T \leq T'$ ssi $T \setminus T' = T \cap \neg T' \leq \bot$.
- Par les lois de de Morgan et la distributivité,
  on se ramène à $\cup_i \cap_j T_{i,j} \leq \bot$, puis $\cap_k T'_k \leq \bot$.
- Simplifier les cas mixtes :
  - $(T_1 \times T_2) \cap (T'_1 \to T'_2) \leq \bot$ toujours vrai.
  - $(T_1 \times T_2) \cap \neg(T'_1 \to T'_2) \leq \bot$ ssi $T_1 \times T_2 \leq \bot$.
  - etc.
- En exercice : traiter le cas $\bigcap_i T_i \times T'_i \leq \bigcup_j S_j \times S'_j$ et de même pour les flèches.
- Et pour les types récursifs ?

Décider le sous-typage

Cette définition sémantique n'exclut pas de décider $T \leq T'$.
C'est même plus élémentaire qu'avec une définition syntaxique du sous-typage ensembliste !

- On peut se ramener au test de vacuité : $T \leq T'$ ssi $T \setminus T' = T \cap \neg T' \leq \bot$.
- Par les lois de de Morgan et la distributivité,
  on se ramène à $\cup_i \cap_j T_{i,j} \leq \bot$, puis $\cap_k T'_k \leq \bot$.
- Simplifier les cas mixtes :
  - $(T_1 \times T_2) \cap (T'_1 \to T'_2) \leq \bot$ toujours vrai.
  - $(T_1 \times T_2) \cap \neg(T'_1 \to T'_2) \leq \bot$ ssi $T_1 \times T_2 \leq \bot$.
  - etc.
- En exercice : traiter le cas $\bigcap_i T_i \times T'_i \leq \bigcup_j S_j \times S'_j$ et de même pour les flèches.
- Et pour les types récursifs ? On déroule et on mémoïse.

Cet algorithme peut être adapté pour produire un témoin en cas de non-inclusion.

Programmation XML

Le XML est un format de données “semi-structurées” utilisé pour structurer les pages web, des fichiers de configuration, des bases de données, etc.

Les *schémas* XML (e.g. DTD, Relax NG) permettent de spécifier des sous-ensembles de documents XML se conformant à certaines attentes (e.g. XHTML, Docbook).

List of types of XML schemas, Wikipedia. [WWW]

Hubert Comon et al., *Tree Automata Techniques and Applications*, 2007. [WWW]

Programmation XML

Le XML est un format de données “semi-structurées” utilisé pour structurer les pages web, des fichiers de configuration, des bases de données, etc.

Les *schémas* XML (e.g. DTD, Relax NG) permettent de spécifier des sous-ensembles de documents XML se conformant à certaines attentes (e.g. XHTML, Docbook).

List of types of XML schemas, Wikipedia. [WWW]

Hubert Comon et al., *Tree Automata Techniques and Applications*, 2007. [WWW]

Des langages de requêtes spécifiques permettent de sélectionner des noeuds dans un document XML (e.g. CSS, XPath) ou de transformer ces documents (e.g. XSLT, XQuery).

Programmation XML

Le XML est un format de données “semi-structurées” utilisé pour structurer les pages web, des fichiers de configuration, des bases de données, etc.

Les *schémas* XML (e.g. DTD, Relax NG) permettent de spécifier des sous-ensembles de documents XML se conformant à certaines attentes (e.g. XHTML, Docbook).

List of types of XML schemas, Wikipedia. [WWW]

Hubert Comon et al., *Tree Automata Techniques and Applications*, 2007. [WWW]

Des langages de requêtes spécifiques permettent de sélectionner des noeuds dans un document XML (e.g. CSS, XPath) ou de transformer ces documents (e.g. XSLT, XQuery).

Mais. . . pas de vérification statique des schémas.

Programmation XML statiquement typée

XDuce, un langage de programmation XML statiquement typé [PDF, WWW] :

- Opérations ensemblistes sur les types de base.

CDuce généralise cette approche :

- Sous-typage sémantique complet, opérations ensemblistes sur tous les types.
- Interopérabilité avec OCaml et frameworks web, extensions e.g. polymorphisme.

```
type Program = <program>[ Day* ]
type Day = <day date=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
let patch_program
  (p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program] =
  xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

- Site web www.cduce.org pour en lire plus, voir des exemples (dont arbres rouge & noir) et jouer avec un interprète en ligne.