Typage avancé de langages dynamiques

Mickaël LAURENT (Université Paris Cité) *

1 Introduction

Avec l'essor du Web, la quantité de code écrit en JavaScript a considérablement augmenté ces dernières années. Afin d'augmenter la sûreté de tels programmes, on veut pouvoir typer avec précision des expressions utilisant des *typecases*, i.e. des branchements conditionnés par le résultat d'un test de type (à l'éxecution) d'une variable ou expression. Ce motif de programmation est omniprésent en JavaScript car il permet d'implémenter la surcharge des fonctions. Des solutions au problème existent, avec des langages comme TypeScript [12] ou Flow[4] qui étendent JavaScript avec un système de types statique. Nous souhaitons cependant aller plus loin que ces langages, sur plusieurs points. Considérons par exemple le code TypeScript suivant

```
function foo(x : number | string) {
    return (typeof(x) === "number")? x+1 : x.trim();
}
```

On reconnait ici une fonction JavaScript classique à la différence près que le paramètre `x` est annoté par un type. Ce code utilise la fonction prédéfinie `typeof` du langage, qui renvoie une chaine de caractères décrivant le type dynamique de son argument. Afin de pouvoir typer statiquement ce programme, il faut déduire, lors du typage de la première branche `x+1`, que `x` est un nombre. De plus, il faut déduire, lors du typage de la seconde branche `x.trim()`, que `x` est une chaine de caractères. Ainsi, les différentes occurrences de la variable `x` devront être typées différemment selon le contexte dans lequel elles se trouvent : on appelle cette discipline de typage *l'occurrence typing*[11].

L'objectif est de créer un système de types respectant les critères suivants :

— intégrer de *l'occurrence typing* afin de pouvoir typer l'exemple ci-dessus.

— fournir des types précis. Par exemple, on veut pouvoir dire que le type de la fonction ci-dessus est $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{String} \to \texttt{String})$, désignant les fonctions pouvant prendre en entrée à la fois des entiers et des chaines de caractères, et renvoyant dans le premier cas un entier et dans le deuxième cas une chaine de caractères. Ces intersections de types fonctionnels permettent de typer précisemment les fonctions surchargées, c'est-à-dire les fonctions réalisant des opérations différentes selon le type de leurs paramètres.

— fournir une inférence de types, afin que l'utilisateur n'ai pas besoin d'écrire lui-même des annotations de type. Par exemple, dans le code ci-dessus, le domaine de la fonction doit pouvoir être inféré automatiquement sans l'annotation de type (ce qui est impossible en TypeScript ou Flow, ces derniers ne proposant pas d'inférence de types).

— être suffisamment général pour pouvoir typer avec précision des expressions plus complexes, notamment contenant des tests de type sur une expression arbitraire. Par exemple, lors du typage d'une expression de la forme `typeof(f(x)) == "number")? ... : ...`, on veut pouvoir déduire pour chaque branche des informations de type à la fois sur `x`, sur `f` et sur `f(x)`.

Pour parvenir à ce résultat, nous utilisons des *types ensemblistes* avec du sous-typage sémantique (remplissant de ce fait le deuxième critère présenté ci-dessus).

*Thèse commencée en Septembre 2020 et encadrée par Giuseppe CASTAGNA (CNRS, Université Paris Cité) et Kim NGUYEN (Université Paris-Saclay)

2 Types ensemblistes

Les types ensemblistes que nous utilisons sont ceux définis par FRISCH, CASTAGNA et BENZAKEN [6] et FRISCH, CASTAGNA et BENZAKEN [7]. Leur syntaxe est la suivante :

$$\textbf{Types} \quad t \quad ::= \quad b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

Chaque type dénote un ensemble de valeurs de notre langage. Ainsi, le type `Empty` ne contient aucune valeur, tandis que le type `Any` dénote l'ensemble de toutes les valeurs. On note $[\![t]\!]$ l'ensemble de valeurs dénoté par un type t : on a par exemple $[\![\texttt{Empty}]\!] = \{\}$, $[\![\texttt{True}]\!] = \{\texttt{true}\}$, $[\![\texttt{Bool}]\!] = \{\texttt{true}, \texttt{false}\}$, etc.

Grâce à cette inteprétation ensembliste des types, on peut définir entre eux une relation de sous-typage dite *sémantique* :

$$t_1 \leq t_2 \overset{def}{\Leftrightarrow} [\![t_1]\!] \subseteq [\![t_2]\!]$$

Parmi les travaux utilisant ces types ensemblistes, on compte notamment :

— Le système du langage CDuce[9], décrit par FRISCH [5], qui propose un système puissant mais fonctionnant sur un lambda-calcul explicitement typé. Il propose un pattern matching très général, dont les *typecases* sont un cas particulier, mais ne fait pas d'occurrence typing dessus. Ce système nous servira néanmoins de base.

— Le système décrit par PETRUCCIANI [8] qui permet d'inférer le type des fonctions, mais sans intersection de types fonctionnels. Il ne permet donc pas de typer les fonctions surchargées avec précision. Il ne propose pas d'occurrence typing non plus.

3 Système de types

Le langage que nous utilisons pour notre système de types est le lambda-calcul usuel, muni d'une construction additionnelle pour les *typecases* :

$$\begin{array}{lll} \textbf{Expressions} & e & ::= \quad c \mid x \mid \lambda x.e \mid ee \mid (e,e) \mid \pi_i e \mid \ (e \in t)\ ?\, e : e \\ \textbf{Values} & v & ::= \quad c \mid \lambda x.e \mid (v,v) \end{array} \tag{1}$$

Nous utilisons comme base les règles de typage usuelles du lambda-calcul simplement typé. Comme nous utilisons des types ensemblistes avec sous-typage, nous avons également besoin d'une règle de subsomption [1] :

$$[\to\text{I}]\ \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad [\to\text{E}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\leq]\ \frac{\Gamma \vdash e : t \quad t \leq t'}{\Gamma \vdash e : t'}$$

Ce qui nous intéresse maintenant, c'est le typage des *typecases*. Pour cela, nous utilisons deux règles simples :

$$[\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash\ (e \in t)\ ?\, e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash\ (e \in t)\ ?\, e_1 : e_2 : t_2}$$

La première concerne uniquement le cas où le test est toujours vrai (et donc la première branche sera toujours choisie), et la deuxième le cas où le test est toujours faux (et donc la deuxième branche sera toujours choisie). Malheureusement, ces deux règles seules ne suffisent pas dans le cas général, où l'expression testée e peut avoir un type qui intersecte à la fois t et $\neg t$.

Afin de pouvoir typer le cas général, nous avons besoin d'une règle capable de décomposer le type s de l'expression testée en deux types disjoints, $s \wedge t$ et $s \wedge \neg t$, afin de pouvoir par la suite appliquer les règles $[\in_1]$ et $[\in_2]$ dans chaque cas séparemment. Nous utilisons pour cela la règle d'élimination de l'union :

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \quad \Gamma, x : t_1 \vdash e : t \quad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

Cette règle peut être appliquée sur les expressions testées, mais pas seulement : elle peut être appliquée sur n'importe quelle sous-expression, permettant ainsi de mettre en corrélation le type de cette sous-expression avec le résultat d'un test.

1. Il s'agit de la règle $[\leq]$. Les règles pour les paires, variables et constantes, quant à elles, ont été omises.

Par exemple, reprenons la fonction `foo` de l'introduction. Pour rappel, elle a le type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{String} \to \texttt{String})$. Essayons maintenant de typer l'expression suivante (pour x ayant le type $\texttt{Int} \vee \texttt{String}$) :

$$((\texttt{foo}\ x) \in \texttt{Int})\ ?\ (\texttt{foo}\ x) + x + 1 : 42$$

Afin de typer la première branche, on doit déduire du résultat du test non seulement que $\texttt{foo}\ x$ est un entier, mais également que x est un entier (car si ce n'était pas le cas, $\texttt{foo}\ x$ aurait renvoyé une valeur de type `String`).

Pour cela, on peut simplement appliquer la règle [$\vee$] sur x afin d'étudier séparemment le cas où x a le type `Int` du cas où il a le type `String`. Lorsque x a le type `Int`, alors $(\texttt{foo}\ x)$ a également le type `Int` et ainsi le test est toujours vrai : nous pouvons alors appliquer la règle [$\in_1$] et typer la première branche. Lorsque x a le type `String`, le test est cette fois toujours faux et nous pouvons alors conclure avec la règle [$\in_2$].

L'avantage de ce système est qu'il est très simple et cependant très puissant : la règle d'élimination de l'union permet, en séparant les types de sous-expressions choisies, de faire tous les raffinements possibles de notre environnement lors du typage des branches d'un *typecase*. Nous avons montré que ce système est *sûr*, dans le sens où une expression typable avec ce système se réduit toujours à une valeur du même type ou alors diverge.

En revanche, l'inconvénient est que ce système n'est pas algorithmique :

— La règle [$\vee$] rend le système non dirigé par la syntaxe, car elle peut s'appliquer dans n'importe quel contexte et substituer n'importe quelle(s) sous-expression(s).

— Les règles [$\to$I] et [$\vee$] sont non-analytiques, dans le sens où l'on doit deviner, respectivement, le type t_1 de la variable abstraite et la décomposition $t_1 \vee t_2$ du type de la sous-expression choisie.

Pour résoudre le premier point, nous avons introduit une forme normale pour les expressions, que nous avons appelé *Maximal Sharing Canonical form* (MSC). Le principe de la forme MSC est de déplacer toutes les sous-expressions de notre expression initiale dans des définitions séparées, tout en factorisant le code le plus possible (i.e., deux sous-expressions syntaxiquement équivalentes vont partager la même définition). Par exemple, l'expression $((\texttt{foo}\ x) \in \texttt{Int})\ ?\ (\texttt{foo}\ x) + x + 1 : 42$ devient :

$$\begin{array}{l}
\texttt{bind}\ u = x + 1\ \texttt{in} \\
\texttt{bind}\ v = \texttt{foo}\ x\ \texttt{in} \\
\texttt{bind}\ w = v + u\ \texttt{in} \\
\texttt{bind}\ z = (v \in \texttt{Int})\ ?\ w : 42\ \texttt{in} \\
z
\end{array}$$

L'intéret est que désormais, toutes les sous-expressions sont définies les unes après les autres, et on peut donc appliquer la règle [$\vee$] une fois sur chacun de ces `bind` sans perdre de généralité. Ainsi, notre nouvelle règle [$\vee$] devient :

$$[\vee\text{-MSC}]\ \frac{\Gamma \vdash_{\mathcal{I}} e_1 : \bigvee_{j \in J} t_j \quad (\forall j \in J)\ \Gamma, x{:}t_j \vdash_{\mathcal{I}} e_2 : s_j}{\Gamma \vdash_{\mathcal{I}} \texttt{bind}\, x = e_1\, \texttt{in}\, e_2 : \bigvee_{j \in J} s_j}\ J \neq \varnothing$$

Pour résoudre le second point, qui est de choisir la bonne décomposition lors de l'application de [$\vee$] et le bon type pour la variable abstraite lors de l'application de [$\to$I], nous ajoutons des annotations dans nos expressions en forme MSC : chaque lambda-abstraction et chaque `bind` se voit enrichi d'annotations indiquant, en fonction de l'environnement actuel, les types à considérer.

Avec ces expressions en forme MSC annotées, nous pouvons définir un système de types algorithmique équivalent au système de types initial : toute expression typable dans le système initial, une fois mise en forme MSC, peut être annotée de telle sorte à devenir typable dans le système algorithmique. Inversement, si une expression en forme MSC est typable par le système algorithmique, alors l'expression d'origine est typable par le système initial.

La dernière étape pour pouvoir typer une expression du langage source est donc de fournir un algorithme permettant d'inférer ces annotations. Nous avons proposé un tel algorithme qui, bien que non complet, donne en général de meilleurs résultats que les approches actuelles de la littérature (notamment [10]). Cet algorithme fonctionne itérativement en raffinant au fur et à mesure les annotations de notre expression en forme MSC : il va pour cela regarder les tests de type qui sont faits ainsi que les applications de fonctions surchargées, et en déduire des décompositions à faire jusqu'à ce que l'expression devienne typable, ou au contraire, que l'algorithme détecte qu'il ne pourra pas la rendre typable.

Nous avons réalisé un prototype implémentant le système de type présenté dans cette section (ainsi que l'algorithme d'inférence). Il peut être testé en ligne à cette adresse : `https://typecaseunion.github.io/`. Ce prototype a été implémenté en OCaml (environ 4000 lignes à l'heure actuelle), en utilisant la bibliothèque de types ensemblistes du langage CDuce[9].

4 Extensions en cours et futures

Le travail présenté jusqu'ici a donné lieu à une publication à la conférence POPL [1]. Cependant, le système de types et le système d'inférence des annotations que nous y avons présenté a quelques limitations. Nous en avons déjà dépassé certaines, mais il reste plusieurs points à améliorer.

4.1 Inférence des arguments fonctionnels

Le système d'inférence des annotations n'est pas capable d'inférer des types fonctionnels pour les paramètres d'une lambda-abstraction. Nous sommes actuellement en train de retravailler ce système d'inférence pour permettre cela. Par exemple, si on modifie notre fonction `foo` de la manière suivante :

```
function foo(f, x) {
    return (typeof(x) === "number")? f(x)+1 : x.trim();
}
```

alors on veut pouvoir inférer le type $((\texttt{Int} \to \texttt{Int}) \times \texttt{Int} \to \texttt{Int}) \wedge (\texttt{Any} \times \texttt{String} \to \texttt{String})$.

4.2 Polymorphisme

Notre système de types actuel ne supporte pas le polymorphisme. Ainsi, la fonction identité sera typée par notre système avec le type $\texttt{Any} \to \texttt{Any}$, ce qui est peu précis. En se basant sur les travaux [2, 3], l'ajout du polymorphisme à notre système devrait résoudre un certain nombre de problèmes de précision. Par exemple, si on modifie la seconde branche de `foo` de la manière suivante :

```
function foo(x) {
    return (typeof(x) === "number")? x+1 : x;
}
```

alors on veut pouvoir inférer le type $\forall\alpha.\ (\texttt{Int} \to \texttt{Int}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$.

4.3 Effets de bord

Nous travaillons actuellement avec un lambda-calcul pur. L'ajout d'effets de bord nécessite dans un premier temps de revoir la règle d'élimination de l'union : en effet, cette dernière ne devrait pas pouvoir s'appliquer sur des sous-expressions non pures, car deux expressions non pures syntaxiquement équivalentes peuvent se réduire à deux valeurs différentes. Par exemple, remplaçons dans notre fonction `foo` la variable x par une application $f(x)$, avec f de type $\texttt{Any} \to (\texttt{Int} \vee \texttt{String})$:

```
function foo(x) {
    return (typeof(f(x)) === "number")? f(x)+1 : f(x).trim();
}
```

Si on suppose que f est pure, alors cette expression peut être typée par $\texttt{Any} \to (\texttt{Int} \vee \texttt{String})$. En revanche, sans la garantie que f est pure, alors ce type n'est plus correct car l'appel à $f(x)$ de la première branche peut donner une valeur dans `String`, malgré le fait que ce même appel a donné une valeur dans `Int` lors du précédent test.

Au niveau du système algorithmique, c'est le partage réalisé lors de la mise en forme MSC qui doit être régulé : deux sous-expressions impures ne doivent pas être représentées par la même variable. Cela nécessite de détecter au préalable les expressions impures, et ceci le plus finement possible. Cette analyse étant préalable au typage, elle devra se faire directement sur une expression non-typée.

Références

[1] Giuseppe CASTAGNA et al. “On Type-Cases, Union Elimination, and Occurrence Typing”. In : *Proc. ACM Program. Lang.* 6.POPL (jan. 2022). DOI : 10.1145/3498674. URL : https://doi.org/10.1145/3498674.

[2] Giuseppe CASTAGNA et al. “Polymorphic Functions with Set-Theoretic Types. Part 1 : Syntax, Semantics, and Evaluation”. In : *Proceedings of the 41st Annual ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’14. Jan. 2014, p. 5-17. DOI : 10.1145/2676726.2676991.

[3] Giuseppe CASTAGNA et al. “Polymorphic functions with set-theoretic types. Part 2 : local type inference and type reconstruction”. In : *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’15. Jan. 2015, p. 289-302. DOI : 10.1145/2676726.2676991.

[4] *Flow*. Facebook URL : https://flow.org/.

[5] Alain FRISCH. “Théorie, conception et réalisation d’un langage de programmation adapté à XML”. Thèse de doct. Université Paris Diderot, déc. 2004. URL : http://www.cduce.org/papers/frisch_phd.pdf.

[6] Alain FRISCH, Giuseppe CASTAGNA et Véronique BENZAKEN. “Semantic Subtyping”. In : *LICS ’02, 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2002, p. 137-146. DOI : 10.1109/LICS.2002.1029823.

[7] Alain FRISCH, Giuseppe CASTAGNA et Véronique BENZAKEN. “Semantic subtyping : dealing set-theoretically with function, union, intersection, and negation types”. In : *Journal of the ACM* 55.4 (sept. 2008), 19 :1-19 :64. ISSN : 0004-5411. URL : http://doi.acm.org/10.1145/1391289.1391293.

[8] Tommaso PETRUCCIANI. “Polymorphic Set-Theoretic Types for Functional Languages”. Thèse de doct. Joint Ph.D. Thesis, Università di Genova et Université Paris Diderot, mar. 2019. URL : https://tel.archives-ouvertes.fr/tel-02119930.

[9] *The CDuce Compiler*. CDuce URL : https://www.cduce.org.

[10] Sam TOBIN-HOCHSTADT et Matthias FELLEISEN. “Logical types for untyped languages”. In : *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’10. Baltimore, Maryland, USA : ACM, 2010, p. 117-128. ISBN : 978-1-60558-794-3. DOI : 10.1145/1863543.1863561. URL : http://doi.acm.org/10.1145/1863543.1863561.

[11] Sam TOBIN-HOCHSTADT et Matthias FELLEISEN. “The Design and Implementation of Typed Scheme”. In : *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA : ACM, 2008, p. 395-406. ISBN : 978-1-59593-689-9. DOI : 10.1145/1328438.1328486.

[12] *TypeScript*. Microsoft URL : https://www.typescriptlang.org/.