Un prototype de système de types graduels ensemblistes pour Elixir

Guillaume Duboc

Un prototype de système de types graduels ensemblistes pour Elixir

Guillaume Duboc

IRIF, Remote Technology

Elixir est un langage de programmation fonctionnel exécuté sur la machine virtuelle Erlang. Il a récemment fait ses preuves dans divers domaines, notamment les applications Web à large traffic (Discord, Pinterest). La communauté Elixir est en plein essor, et cette recherche répond à une demande croissante d'intégration d'un système de types statiques dans Elixir, similaire à l'approche de Typescript [BAT14] qui a prouvé son efficacité et sa popularité. Sous la direction de Giuseppe Castagna et José Valim, le créateur d'Elixir, ce projet de thèse vise à développer et implémenter un système de types statiques pour le langage. Un prototype d'implémentation de ce système en Elixir est actuellement accessible via `https://typex.fly.dev/` (et sur Zenodo [DCV23]). Cette présentation illustre les difficultés spécifiques au typage d'Elixir, ainsi que les solutions apportées, tout en offrant une perspective sur la conception d'un système de types pratique dont la théorie n'est pas encore complètement établie.

1 Introduction : Elixir et les types ensemblistes

Les fonctions Elixir sont définies par des clauses de filtrage sur les arguments. Une manière typique de définir une fonction en Elixir est donc la suivante :

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

La première clause définit la fonction `negate` pour les entiers, et la seconde pour les booléens. Introduire un système de typage statique dans Elixir signifie, dans notre cas, de permettre aux programmeurs d'apposer sur la fonction `negate` une annotation de type, par exemple

```
$ (integer() or boolean()) -> (integer() or boolean())
```

qui indique que la fonction prend en argument un entier ou un booléen, et renvoie un entier ou un booléen. Le connecteur de type `or` est *ensembliste*, c'est-à-dire que si deux types sont vus comme représentant l'ensemble des valeurs qu'ils typent (e.g. 0, 1, 2 ... pour `integer()`), alors le type (`integer() or boolean()`) est l'union de ces deux ensembles. Cet opérateur ensembliste n'est pas le seul : en effet, on peut se rendre compte que si `negate` est appelée par une fonction annotée comme `subtract`

```
$ (integer(), integer()) -> integer()
def subtract(x, y) when is_integer(x) and is_integer(y) do
    x + negate(y)
end
```

alors une erreur de type serait levée, déclarant que l'opérateur `+` est appelé avec un terme de type `integer() or boolean()`. Ici, la notion de type *intersection* est utilisée pour donner un type fonctionnel plus précis, polymorphe ad-hoc, à la fonction `negate` :

```
$ (integer() -> integer()) and (boolean() -> boolean())
```

Ce type capture le fait que, lorsque `negate` reçoit un entier en argument, elle renvoie un entier, et lorsque c'est un booléen, un booléen. Ces deux connecteurs font partie de la syntaxe du langage et peuvent être utlisés librement dans les annotations de types, avec également la négation de type, les types singletons, et les variables de types, tous trois utilisés par l'exemple suivant :

```
$ ((false or nil, a) -> a) and
  ((b, term()) -> b) when a: term(), b: not(false or nil)
def logical_or(x, y) when x == false or x == nil, do: y
def logical_or(x, _), do: x
```

La fonction `logical_or` prend deux arguments en entrée ; si le premier est un atome `false` ou `nil` (ceux-ci étant représentés par les types singletons du même nom), alors le second argument est renvoyé, dont le type est capturé par la variable `a` (quantifiée après le type, avec une borne supérieure). Cette variable n'a pas de contrainte particulière puisque sa borne supérieur est `term()` soit le type de toutes les valeurs. Sinon, c'est le premier argument qui est renvoyé, dont le type est capturé par la variable `b` ; celle-ci a pour borne supérieure le type négation `not(false or nil)`, qui est le type de toutes les valeurs qui ne sont ni `false` ni `nil`.

2 Spécificités du typage d'Elixir

Si le cœur de ce typeur réside dans son adoption d'un système de types basé sur le sous-typage sémantique [CF05], une approche enrichie au fil des années avec du polymorphisme [CX11] et du typage graduel [CLPS19], l'extension de ce système à Elixir [CDV23] nécessite des efforts conséquents que nous présentons ici.

2.1 Analyse typée du filtrage avec gardes

Typer Elixir nécessite d'offrir une analyse typée précise des gardes, la difficulté venant du fait que les gardes sont des expressions complexes : ce sont des combinaisons booléennes (avec `and`, `or`, `not`), des tests d'égalité (`==`, `!=`), des comparaisons (`<`, `<=`, `>`, `>=`), des tests de type (`is_atom`, `is_integer`, etc.), des tests de présence dans une liste (`in`), des sélections dans des tuples, listes ou dictionnaires (`elem`, `hd`, `tl`, `map.key`), des opérateurs sur les structures de données (`tuple_size`, `map_size`, etc.). Cette analyse permet à notre typeur de détecter des erreurs de non exhaustivité (ou des branches redondantes) dans le filtrage par motifs, comme dans cet exemple de code qui manipule un type de résultat encodé dans un dictionnaire avec des champs `:output`, `:socket` ou bien `:output`, `:message`.

```
$ type result() =
    %{output: :ok, socket: socket()} or
    %{output: :error, message: :timeout or {:delay, integer()}}

$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.message == :timeout, do: "Timeout"
#=> Type Warning: non-exhaustive pattern matching

$ result() -> string()
def hand(r) when r.output == :ok, do: "Msg received"
def hand(r) when r.output == :error, do: "Error raised"
def hand(%{socket: _}), do: "Socket found"
#=> Type Warning: unused branch
```

Comme les fonctions sont définies par clauses de filtrage successives, cela nous aussi permet d'inférer des types intersections précis qui capturent la relation exacte entre types d'entrées et de sorties de chaque clause. Par exemple, dans cet exemple

```
def handle(r) when r.output == :ok, do: {:accepted, r.socket}
def handle(r) when is_atom(r.message), do: r.message
def handle(r), do: {:retry, elem(r.message, 1)}

$ %{output: :ok, socket: socket()} -> {:accept,socket()} and
  %{output: :error, message: :timeout} -> :timeout and
  %{output: :error, message: {:delay,integer()}} -> {:retry,integer()}
```

Ces différents exemples sont traités dans l'interface Typex sous le nom (`result/handle`).

2.2 Typage graduel : une approche hybride via les types fonctionnels forts

Le typage graduel [ST06] (c.à.d. introduire un type `dynamic()` qui peut devenir n'importe quel type à l'exécution) en pratique donne lieu à des compromis entre sûreté du typage et quantité d'information fournie par le typeur. Typiquement, le typeur de Typescript est non-sûr car, par défaut Javascript ne vérifie pas les types, et le type dynamique, ne peut ainsi pas être contrôlé même lorsqu'il est utilisée par des programmes entièrement annotés. Récupérer cette sûreté nécessite d'activer un mode de compilation spécifique [RSF+15].

Notre approche ici est de proposer un système de types sûr par défaut, sans modifier l'exécution. Cet objectif est atteint en prenant en compte les vérifications de types effectuées par la machine virtuelle Erlang. Celle-ci vérifie de manière systématiques les arguments d'opérateurs comme l'addition ; nous parlons d'opérations fortes, car elles sont sûres par défaut. Les gardes écrites par les programmeurs font également l'objet de telles vérifications, ce qui nous m à ajouter au système de types des types fonctionnels forts, qui désignent une fonction qui vérifie concrètement le type de ses arguments. Ainsi l'identité avec ou sans garde explicite est typée avec le même type, mais si ces fonctions sont appelées avec une expression de type dynamique, elles renverront des types différents :

```
$ integer() -> integer()
def id_weak(x), do: x

id_weak(x_dyn)
#=> dynamic()
```

```
$ integer() -> integer()
def id_strong(x) when is_integer(x), do: x

id_strong(x_dyn)
#=> integer()
```

3 Intérêt du logiciel

Le prototype Typex est une implémentation en Elixir d'un futur système de types permettant d'ajouter des annotations de types aux programmes Elixir, et de les vérifier. Il a été présenté lors de conférences majeures comme ElixirConf 2023 et BEAM Code 2023, recevant un accueil très positif. La question du typage dans les communautés Erlang et Elixir est très présente, avec divers efforts et théories [LS06, eqw, Jos19, CTPV20, VH18, SWB23], dont l'adoption n'est pas complète, en premier lieu car résoudre les problèmes spécifiques à Elixir de manière assez satisfaisante nécessite des techniques en partie développées dans ce projet. José Valim, le créateur d'Elixir, joue un rôle actif et soutient fortement ce projet de recherche, ce qui augmente considérablement sa visibilité et son acceptabilité au sein de la communauté Elixir. Le typeur développé dans le cadre de ce projet, dont Typex représente une façade de démonstration, est prévu pour être intégré progressivement au compilateur, ce qui ferait à terme d'Elixir un langage statiquement typé par défaut.

Références

[BAT14] G. Bierman, M. Abadi et M. Torgersen : Understanding TypeScript. *In European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[CDV23] Giuseppe Castagna, Guillaume Duboc et José Valim : The Design Principles of the Elixir Type System, 2023.

[CF05] Giuseppe Castagna et Alain Frisch : A gentle introduction to semantic subtyping. *In Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 198–208, 2005.

[CLPS19] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani et Jeremy G Siek : Gradual typing : a new perspective. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019.

[CTPV20] Mauricio Cassola, Agustín Talagorria, Alberto Pardo et Marcos Viera : A gradual type system for Elixir. *In Proceedings of the 24th Brazilian Symposium on Context-oriented Programming and Advanced Modularity*, pages 17–24, 2020.

[CX11] Giuseppe Castagna et Zhiwu Xu : Set-theoretic foundation of parametric polymorphism and subtyping. *In Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, pages 94–106, 2011.

[DCV23] Guillaume Duboc, Giuseppe Castagna et Jose Valim : Accepted artifact for 'the design principles of the elixir type system'. `https://doi.org/10.5281/zenodo.8425534`, 2023.

[eqw] eqWAlizer. `https://github.com/WhatsApp/eqwalizer`.

[Jos19] Svenningsson Josef : Gradualizer. `https://github.com/josefs/Gradualizer`, 2019.

[LS06] Tobias Lindahl et Konstantinos Sagonas : Practical type inference based on success typings. *In ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2006.

[RSF+15] A. Rastogi, N. Swamy, C. Fournet, G. Bierman et P. Vekris : Safe & efficient gradual typing for TypeScript. *In POPL '15*, pages 167–180. ACM, 2015.

[ST06] J. G. Siek et W. Taha : Gradual typing for functional languages. *In Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[SWB23] Albert Schimpf, Stefan Wehr et Annette Bieniusa : Set-theoretic types for erlang. *arXiv preprint arXiv :2302.12783*, 2023.

[VH18] Nachiappan Valliappan et John Hughes : Typing the wild in Erlang. *In Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*, pages 49–60, 2018.