Type Inference of Polymorphic and Overloaded Functions Towards Static Typing of Dynamic Languages
Mickaël Laurent

To cite this version:
Mickaël Laurent. Type Inference of Polymorphic and Overloaded Functions Towards Static Typing of Dynamic Languages. 35es Journées Francophones des Langages Applicatifs (JFLA 2024), Jan 2024, Saint-Jacut-de-la-Mer, France. hal-04406364

HAL Id: hal-04406364
https://inria.hal.science/hal-04406364
Submitted on 19 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
We present a type-checker prototype that aims to be used with dynamic languages featuring type-cases, such as JavaScript or Elixir. This type system is able to refine the environment when typing the branches of a type-case (occurrence typing), supports both parametric polymorphism (essential for modularity) and ad-hoc polymorphism through intersection types (essential for capturing overloaded behaviors), and features type inference.

1 Introduction

Typing dynamic languages is a challenging endeavour even for very simple pieces of code. For instance, JavaScript’s logical or operator "||" behaves like the following function 1:

```javascript
function lOr(x, y) {
  if (toBoolean(x)) { return x; } else { return y; }
}
```

where `toBoolean` is a function which simply checks whether its argument is one of the 8 values that are considered as falsy in JavaScript (i.e., `false`, `''`, `0`, `−0`, `0n`, `undefined`, `null`, and `NaN`) and in this case returns `false`, otherwise it returns `true`.

A naive type for the `lOr` function would be the type `(Any, Any) → Any` (where `Any` is the type of all values), which is a rather useless type since it essentially states that `lOr` is a binary function. To give `lOr` a more informative type, we need union and intersection types: we define the type `Falsy` as the union type `false ∨ '' ∨ 0 ∨ −0 ∨ 0n ∨ undefined ∨ null ∨ NaN`, where each value denotes here the singleton type containing that value, and the type `Truthy` to be its complement, that is, the type of all values that are not of type `Falsy`. This allows `toBoolean` to be typed `(Truthy → True) ∧ (Falsy → False)`, where `∧` is a type combinator denoting intersection: if the argument of a function of this type is a `Truthy`, then the function returns `true`, while if the argument is a `Falsy`, then the result is `false`. Then, we can deduce for `lOr` the following more precise type:

\[(Truthy, Any) → Truthy) ∧ ((Falsy, Truthy) → Truthy) ∧ ((Falsy, Falsy) → Falsy)\]

Notice how the use of an intersection of arrow types corresponds to the typing of an "overloading" behavior (also known as ad-hoc polymorphism), insofar as the result of an application depends on the type of the input. In order to derive such a type, the type system must deduce that whenever the condition tested by the `if` holds, then `x` is of type `Truthy` 1.

1This definition does not capture the short-circuit evaluation of "||".
and, therefore, (i) that all occurrences of x in the “then” branch (here just one) have type \text{Truthy} and (ii) that all the occurrences of the same variable x in the branch “else” (here none) have thus type \text{Falsy}. This kind of deduction is usually referred as occurrence typing \cite{THF10} since it requires to “narrow” the type of a variable x differently for its different occurrences. Our prototype is able to perform such type narrowing even when the tested expression is an application. We can be even more precise by inferring intersections of polymorphic function types. Indeed, for the \text{lOr} function defined above, the prototype infers the following first order polymorphic type (where \alpha and \beta are type variables):

\[ \forall \alpha, \beta . ((\alpha \land \text{Truthy}, \text{Any}) \rightarrow \alpha \land \text{Truthy}) \land ((\text{Falsy}, \beta) \rightarrow \beta) \]  

This type completely specifies the semantics of the function \text{lOr}: it states that if the first argument is a \text{Truthy}, then the application of the function returns the first argument, otherwise it returns the second argument. This type is more precise than the one in (1), since it allows the system to deduce that, say, if the first argument of \text{lOr} is an object, then the result will be an object of the same type (rather than just a truthy value).

## 2 Features

For any constant of the language, our prototype features the corresponding singleton type (\text{Nil}, \text{True}, \text{False}, etc.). Types can be combined using union \|, intersection \&, difference \\, and negation \~. Recursive and parametric types are also supported. A built-in syntax for the types of lists is also available, supporting regular expressions. For instance, [\text{Int} \mid \text{Bool}^*] denotes the lists starting with an integer followed by any number of Boolean values.

\begin{verbatim}
4 atom null (* Defines a new value null and a singleton type Null *)
5 type Falsy = False | "" | 0 | Null
6 type Truthy = ~Falsy
7 type Tree 'a = ('a[\text{Any}^*]) | [(Tree 'a)*] (* Recursive & parametric *)
\end{verbatim}

The example from the introduction can be written as follows (the types inferred are written as comments):

\begin{verbatim}
8 (* (Falsy -> False) & (Truthy -> True) *)
9 let toBoolean x = if x is Truthy then true else false
10 (* ((Falsy,'b) -> 'b) & ('a\Falsy, Any) -> 'a\Falsy) *)
11 let lOr (x,y) = if toBoolean x then x else y
12 (* 'a -> 'a *)
13 let (id : 'a -> 'a) x = lOr (x,x)
\end{verbatim}

Recursive functions such as \text{map} can be implemented (and typed) by first defining Curry’s fixpoint combinator:

\begin{verbatim}
14 (* (\'a -> \'b) -> X1) -> X1 where X1 = ('a -> 'b) & 'c *)
15 let fixpoint = fun f ->
16   let delta = fun x ->
17     f ( fun v -> ( x x v ))
18   in delta delta
19 (* ... (type omitted for concision) *)
20 let map_stub map f lst =
21   if lst is Nil then nil else (f (fst lst), map f (snd lst))
22 (* (Any -> [ ] -> [ ]) &
23   ([\'a -> \'b] -> ([\'a] -> ['b]) & ([\'a+] -> ['b+]) & ([ ] -> [ ])) *)
24 let map = fixpoint map_stub
\end{verbatim}
For convenience, and to allow for some optimisations, a syntax for recursive functions is available. Pattern matching is also implemented:

```
(* Tree 'a -> ['a*] *)

let rec flatten (x:Tree 'a) = match x with
  | [] -> []
  | (h, t) & :List -> concat (flatten h) (flatten t)
  | _ -> [x]
end
```

Note the presence of a user type annotation for the parameter `x`. Still, the type of `flatten` can be inferred without type annotation, in which case the prototype infers the type `(Tree 'a -> ['a*]) & ('b List -> ['b List])` in about 300ms (the second arrow captures the fact that, when given a value that is not a list, `flatten` returns a singleton list of this value).

This prototype is implemented in OCaml and uses the CDuce library for deciding subtyping. The source code is available in the supplemental material and on Zenodo: [CLN23]. An online version, compiled using Js_of_ocaml, can be tested at the following address: https://www.cduce.org/dynlang/ (it is about 8 times slower than the native version).

### 3 Limitations and future work

This prototype is an implementation of a type system formalized in an unpublished manuscript [CLN24] currently under submission, based on prior work on set-theoretic types [Fri04, CNXA15, CLNL22]. The formal type system is safe (well-typed programs cannot go wrong). However, as we could expect in a type system featuring unrestricted intersections, the type inference is not complete: our inference algorithm is terminating but may fail to type an expression typeable in the formal system.

In particular, the two main weaknesses of this prototype are: (i) the inference time, in particular for recursive functions, and (ii) the lack of principal types.

The first point is due to the backtracking nature of the inference algorithm and the subtyping constraints it has to solve. The inference time can be significantly reduced by providing type annotations, thus narrowing the search space and the amount of backtracking performed by the inference algorithm. Note that the type system is modular: top-level definitions are only typed once (backtracking only occurs inside of a top-level definition).

The second point can be illustrated with the function `map` presented in the previous section. The type inferred for `map` has several intersections. On the one hand, we might prefer to give it its simple ML type, `('a -> 'b) -> ['a*] -> ['b*]`, even though it is less precise. On the other hand, in some specific contexts, we might need an even more precise type, for instance we might need to capture the fact that lists of length 2 are transformed into lists of length 2 (`('a -> 'b) -> ['a;'a] -> ['b;'b]`), and so on. In the limit, we might want to capture the fact that any list of length `n` is transformed into a list of length `n`, but this would need either infinite intersections or dependent types, which are not supported by our type system. As the type system is modular, the lack of principal types makes it impossible to systematically guess a suitable type (when typing the top-level definition `map`, we do not know what uses will be made of it in later top-level definitions). Thus, we do not expect the type inference to be completely autonomous, but rather to assist the user by proposing relevant types that they may or may not choose to use.

Lastly, some future work is necessary before considering application to industrial languages such as JavaScript. In particular, the current prototype (and the formal type system it is based on) is only sound for pure expressions. Work is being made for supporting side-effects, as well as row-polymorphism for records (which is essential for typing oriented object languages, as records can be used to encode objects). In parallel, set-theoretic type systems are starting to be adopted outside of the academic sphere, for instance for the language Elixir [CDV23] or LUAU [Jef22].
References


