Towards an effect system for OCaml

Stephen Dolan, Matija Pretnar, Leo White, KC Sivaramakrishnan

May 13, 2016

With the introduction of algebraic effects to OCaml[1], extending OCaml's type system into a type & effect system is a natural next step. In such a system, programs receive a type $A!\mathcal{E}$, where A is the type of returned values, and $\mathcal{E}$ is the effect annotation, whose exact form is yet to be determined. Even though there is already an existing polymorphic effect system for handlers with an inference algorithm [3], it is not obvious how to include it in OCaml due to backwards compatibility.

There are a number of properties that a feasible effect system should satisfy:

Soundness If a program e receives a type $A!\mathcal{E}$, every potential effect `E` should be captured in $\mathcal{E}$.

Usefulness An effect system that annotates each program with every possible effect there is, is obviously sound, but not very useful. Thus, an effect information should not mention an effect that is guaranteed not to happen.

Backwards compatibility We want each program that was typable before introducing effect annotations, to remain typable. Furthermore, the effect system should play along nicely with OCaml's module system, thus whole-program analysis is out of the question.

To see what the above properties imply, take a program `if X then perform E1 else perform E2`. The effect information of `perform E1` must mention `E1` for the sake of soundness, but omit `E2` for the sake of usefulness. Conversely, the effect information of `perform E2` should mention `E2` but not `E1`. But the whole program must remain typable due to backwards compatibility, and its type should mention both `E1` and `E2` due to soundness. From this, it follows that the effect system needs to provide a way of enlarging effect information. There are two established ways of providing this flexibility: subtyping [4] or row polymorphism [1]. Both are difficult to apply directly to OCaml, due to already-existing language features:

Monomorphic types The ML type system makes a distinction between monomorphic and polymorphic types, and in certain contexts only monomorphic types are permitted. Many existing programs are typeable only because, say, `int` $\to$ `int` is monomorphic, and would break if it became a polymorphic type.

[1]https://github.com/ocamllabs/ocaml-effects

Signature matching Comparing a module implementation against its interface requires not only inferring polymorphic types, but checking whether a given polymorphic type is more polymorphic than another.

Invariant contexts While OCaml supports (explicit) subtyping, not all type parameters are either co- or contra-variant. For instance, the type parameters to `ref`, the indices of GADTs, and unannotated abstract types are neither co- nor contra-variant.

Subtyping makes type inference difficult by breaking unification, so the usual approach is to infer *constrained types* of the form $A|\mathcal{C}$, where $\mathcal{C}$ is the set of constraints between type (and later also effect) parameters in A [2]. However, there are a number of practical problems. First, it is hard to determine when a constrained type $A|\mathcal{C}$ is an instance of $A'|\mathcal{C}'$, causing problems for compatibility with the module system. Next, constraint generation in the inference algorithm needs to be directed in order to keep track of covariance and contravariance. This causes problems with the current inference algorithm of OCaml, which mostly works with equations and is undirected. Finally, constraints are cumbersome to write and difficult to read, decreasing chances of adoption in the programming community.

A possible solution for subtyping is to encode constraints in types, potentially dropping some of them, which results in types that satisfy a weak form of principality: the inferred type is unique and captures most of possible typings of the given program, but not all of them.

For row polymorphism, typability of existing programs poses a problem. These programs, which may cause any effect provided by OCaml (input/output, references, ...), should receive an annotation, say `IO`, that distinguishes them from pure programs. Furthermore, existing monomorphic types should remain monomorphic. For example, a function `old_fun` that used to have a type `unit` $\to$ `unit` should get a type `unit` $\to$ (`unit!IO`). However, one then cannot type the program `if X then old_fun () else perform E`, as the type of the left branch does not contain a row variable and cannot be expanded to mention `E`.

A possible solution for this issue is to give monomorphic types to existing monomorphic programs, but allow a limited form of subeffecting, which weakens the effect annotation during application. Then, for example, `old_fun` would have a type `unit` $\to$ (`unit!IO`), but its application `old_fun ()` would get the type `unit!`[`IO`$|\rho$].

References

[1] Daan Leijen. Koka: Programming with row polymorphic effect types. In *MSFP*, pages 100–126, 2014.

[2] François Pottier. Type inference in the presence of subtyping: from theory to practice. Technical Report RR-3483, INRIA, 1998.

[3] Matija Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014.

[4] Keith Wansbrough and Simon L. Peyton Jones. Once upon a polymorphic type. In *POPL*, pages 15–28. ACM, 1999.