Global Semantic Analysis on OCaml programs

Thomas Blanc, Pierre Chambart, Michel Mauny, Fabrice Le Fessant

Global Semantic Analysis on OCaml programs

Thomas Blanc[1,2], Pierre Chambart[1], Michel Mauny[2], and Fabrice Le Fessant[3,1]

[1]OCamlPro, Paris
[2]ENSTA-ParisTech - Université Paris-Saclay
[3]INRIA Paris-Rocquencourt

Abstract

In this talk, we will present an ongoing project at OCamlPro, the development of a semantic analyser of OCaml code based on abstract interpretation techniques. This analysis relies on the presence of the whole program at compile time, it should work on full actual programs and shows interesting promises in terms of uncaught exceptions detection.

1 Idea

We want to realize an exhaustive semantic analysis on OCaml code. As most of OCaml's behavior is known through typing, more powerful analyses have not been implemented yet. This is different from proof-checking on programs, as the semantic is fully automatically inferred and we don't need the user to provide any kind of annotations. This could, however, provide good safety assertion as it may be used to check for uncaught exceptions.

OCaml's type-system already provides a good hint on program behavior, however, it is useless with respect to errors such as division by zero, array access or random assertion checking. Typing to determine uncaught exceptions has been implemented in [2], yet it was limited by the type system and hard to adapt to recent modifications to the OCaml language (notably first-class modules), making it obsolete now.

2 Issues

The main problem when doing analysis on OCaml is higher-order functions. First-class polymorphic functions make analysis highly complex, as the flow becomes dynamic and split apart. Moreover, the possibility of exception raising at application nodes gives a really different behavior depending on the call-site and on the function called that types cannot express.

Also, programming in OCaml relies a lot on easy memory allocation and reading. A lot of structures are produced and a lot of aliasing has to be remembered in order to have a working analysis. Moreover, mutable blocks add a whole layer of complexity to the problem of representing the memory.

As of others problem we encountered while working on this project, we can name separate compilation, argument evaluation order, the presence of external C functions, objects, threads and dynamic linking.

Authors' addresses: thomas.blanc at ocamlpro.com, pierre.chambart at ocamlpro.com, fabrice.le_fessant at inria.fr, michel.mauny at ensta-paristech.fr

3 Our approach

The best workaround for the problem of higher-order functions is to use whole-program analysis. This allows to check values not only for how they can be used, but also on how they will be used. Knowing that, we looked for a good representation of a program's flow.

First, we used the compiler-libs to generate each source file's lambda representation. This representation is then transformed into a xlambda (eXplicit-lambda) tree which is basically lambda with a few more explicit things:

- The lambda is put into ANF form: evaluation order of function arguments and sub-expressions is made explicit.
- Curryfication is made explicit: all function applications are unary.
- Closure conversion is performed: each function is called with a closure, containing its free variables.
- Primitive's control flow is made explicit (||, &&, division by zero, array access, ...).
- Variable names include the global module name. Global values become let-bindings.

The xlambda is then turned into an hypergraph representation. Basically, each node is a program state and each edge is an operation. Operations that may or may not raise an exception (such as function calls) have one predecessor and two successors. We then concatenate the graphs from each of the files to start the analysis. A fixpoint iteration is done on the graph with function application edges dynamically replaced by the corresponding subgraphs.

Basically, we propagate environment information through the graph, adding at each node a superset of all possible values for each variables until no additional information can be found. This is not simple, as we want to remember memory aliasing (a test on a first member on a tuple should affect the tuple as well) but we don't want to keep too much relational information. For example, a tuple that may contain either $(1, 1)$ or $(2, 2)$ will be remembered as a tuple containing $(\{1, 2\}, \{1, 2\})$, making $(1, 2)$ a valid possibility.

Several abstract interpretation techniques are used to make the analysis reliable while working in reasonable time. For example, a for loop that may go for a lot of iterations will only be passed on for fixed number of times, with the widening technique[1].

This approach looks quite promising and we expect to have applicable results before September, notably interesting possibilities with respect to uncaught exceptions and dead code detection.

References

[1] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[2] François Pessaux and Xavier Leroy. "Type-based analysis of uncaught exceptions". In: *Proc. 26th symp. Principles of Programming Languages*. To appear. ACM Press, 1999.