The Design and Implementation of an Abstract Interpreter for OCaml Programs

Benoît Montagu

The Design and Implementation of an Abstract Interpreter for OCaml Programs: A Preliminary Report on the Salto Analyser

BENOÎT MONTAGU, Inria, France

We report on a work in progress that aims at defining an effective static analyser for OCaml programs, by leveraging abstract interpretation techniques. The goal of the Salto static analyser is to detect precisely which exceptions an OCaml program might raise, and to report problematic cases, where a program execution might rely on elements of the OCaml semantics that are deemed under-specified or undefined. The Salto analyser exploits a novel abstract domain to represent inductively defined sets of trees, that draws inspiration from the theory of recursive types, from tree automata, and from the abstract domain of Type Graphs. The analyser itself is defined using a dynamic fixpoint solver, *i.e.*, a generic library that implements an iteration strategy that finds a post-fixpoint. The solver automatically inserts widening points to ensure the convergence of the iteration process, and aims at limiting the unnecessary computations that may be asked by the iteration strategy.

ACM Reference Format:
Benoît Montagu. 2023. The Design and Implementation of an Abstract Interpreter for OCaml Programs: A Preliminary Report on the Salto Analyser. In . ACM, New York, NY, USA, 4 pages.

1 WELL TYPED PROGRAMS CANNOT GO WRONG

As a member of the ML family, the OCaml programming language [Leroy, Doligez, et al. 2022] benefits from the increased safety that is induced by strong typing [Milner 1978]: "*well typed programs cannot go wrong*". This means that values are always used in a consistent manner by well typed programs, so that abrupt crashes, *e.g.*, memory-related errors such as segmentation faults, are ruled out by the type safety theorem.

The ML type system does not, however, protect against other kind of programming mistakes, such as forgetting to catch some exceptions. Extensions of ML based on type and effect systems have been developed [Leroy and Pessaux 2000] to detect such programming mistakes.

Uncaught exception in OCaml programs may occur for a variety of reasons, such as:

- Explicit raises of exceptions,
- Non-exhaustive pattern matching,
- Out of bound array accesses (and similarly for strings and packed byte arrays),
- Divisions by zero,
- Assertions explicitly written by a programmer,
- Initialisations of recursive modules,
- Polymorphic equality tests and comparisons that involve functions.

Other parts of the OCaml language have deliberately under-specified or unspecified behaviours, and would benefit from early detection. This includes:

- Arithmetic overflows and underflows,
- Lazy patterns, whose evaluation may produce side effects,
- Conditions in patterns (`when` clauses), that may produce side effects,

ML Family Workshop, 2023, Seattle

- Calls to *unsafe* functions (unchecked array accesses, unmarshalling, *etc.*).

In this talk, we present an early prototype of a static analyser for OCaml programs, that is based on the theory of abstract interpretation. The Salto analyser[1] is a *whole program* analyser that infers, for each sub-expression of a program that might be reachable, which values it may reduce to, and which exceptions its evaluation may raise.

The exceptions that result from memory or stack exhaustion, as well as the recently introduced features of OCaml 5.0—*e.g.*, parallelism, algebraic effects—are out of the scope of this project for the moment.

2 A VALUE ANALYSIS FOR HIGHER-ORDER PROGRAMS

It is common knowledge that the control flow of higher-order programs cannot always be statically known, and that it makes the static analysis of such programs difficult. The dynamic nature of the control-flow graph is even exacerbated by the fact that exceptions are *first-class* values in OCaml (they can, for instance, be stored in data structures, or be passed as arguments to functions).

Many earlier work have been devoted to control flow analysis (CFA) [Jones 1981; Shivers 1991; Wright and Jagannathan 1998] for the λ-calculus. The Salto analyser is based on recent work [Montagu and Jensen 2021] that defines a CFA as a value analysis, by abstracting the control flow traces that are produced by program executions. These control flow traces record which calls are performed, and which results are returned, and keep track of the contexts in which such control flow events are produced. For every reachable program point, the analyser infers an abstract value, that denotes an inductively-defined set of values that may be produced by the sub-expression at this program point.

The Salto analyser employs a novel abstract domain, whose salient feature is the presence of a *fixpoint* constructor $\mu\alpha.a$ to represent inductive sets of values. This idea is drawn from the theory of equi-recursive types [Pierce 2002, Chapter 4], and can be understood as a term-based representation of the Type Graphs that were used to analyse Prolog programs [P. Van Hentenryck et al. 1994; Pascal Van Hentenryck et al. 1995].

Similarly to the methodology described in the "*Abstracting Definitional Interpreters*" article [Darais et al. 2017], the Salto analyser is defined as a function in *open recursive style*—*i.e.*, a functional of type $(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)$—that is passed as argument to a dynamic fixpoint solver, and produces a function of type $\tau_1 \to \tau_2$ that computes *on demand* a solution of type τ_2 when it is provided with an argument of type τ_1. The solver we have implemented is inspired by existing solvers from the literature [Charlier and Van Hentenryck 1992; Schulze Frielinghaus et al. 2017; Seidl and Vogler 2018], that can handle abstract domains of unbounded heights, by automatically inserting calls to a widening operator. Widening operators are an essential ingredient of abstract interpretation, that are responsible for finding valuable generalisations of invariants, and ensure the convergence of the iteration process. A similar approach based on dynamic fixpoint solvers has been chosen to implement the Goblint static analyser for C programs [Vojdani et al. 2016].

3 CURRENT STATE OF THE IMPLEMENTATION AND FUTURE CHALLENGES

The prototype implementation of the Salto analyser currently supports the purely functional subset of the OCaml core language only. This includes higher-order functions, recursion, algebraic datatypes, and exceptions. The Salto prototype has not been released yet. So far, it is able to analyse programs of the size of a few hundreds lines of code.

The analyser takes as input the *typed* abstract syntax tree (AST) of OCaml programs that is produced by the typing phase of the compiler, and transforms it into a simplified AST, that has less

[1] https://salto.gitlabpages.inria.fr/

constructs, so as to alleviate the implementation effort of the actual analysis. For example, a *single* construct is responsible for pattern matching in this simplified AST.

The analysis is designed to handle *untyped* programs, and is so far independent from the typing discipline enforced by the OCaml typechecker. This leaves open the future possibility of exploiting type information during the analysis to infer more precise results, or to guide the creations or merges of disjuncts in abstract values.

Our short- to mid-term goals is to expand the set of features supported by the analyser, such as:

- Mutable state,
- Laziness,
- Precise support for strings, floats, finite sets and maps,
- Modules, functors, and first-class modules.

On the longer term, we would also like to support:

- Infinite (coinductive) data-structures,
- Recursive modules and their initialisations,
- Objects and classes,
- System-related features, such as primitives for file-system operations, process management, and signals (`System` and `Unix` modules),
- OCaml 5.0 specific features, *e.g.*, algebraic effects, parallelism,
- Features related to the implementation of the OCaml runtime, *e.g.*, that depend on how data are represented in memory, or that are related to the garbage collector.

On a broader scope, we are also interested in exploring the following research questions:

- Can we detect when the behaviour of a program is sensitive to the order of evaluation of function arguments, of tuple members, *etc.*?
- Can we detect when the behaviour of a program is sensitive to the initialisation order of its compilation units?
- Can more modular analyses be developed for OCaml programs, that would produce precise *function summaries*, such as relational analyses [Andreescu et al. 2019; Montagu and Jensen 2020] that infer relations between inputs and outputs of programs?

ACKNOWLEDGMENTS

This work is funded by a two-year grant from Nomadic Labs and Inria.

We would like to thank Pierre Lermusiaux, Thomas Genet and Thomas Jensen, from Inria and Université de Rennes, for their great contributions to the theory and implementation of the Salto analyser, as well as Mehdi Bouaziz and Vincent Botbol, from Nomadic Labs, for their essential support and feedback.

REFERENCES

Oana F. Andreescu, Thomas Jensen, Stéphane Lescuyer, and Benoît Montagu. Jan. 2019. "Inferring Frame Conditions with Static Correlation Analysis." *Proc. ACM Program. Lang.*, 3, POPL, Article Article 47, (Jan. 2019), 29 pages. DOI: 10.1145/3290360.

Baudouin L. Charlier and Pascal Van Hentenryck. 1992. *A Universal Top-Down Fixpoint Algorithm.* Tech. rep. USA. ftp://ftp.cs.brown.edu/pub/techreports/92/cs92-25.pdf.

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. "Abstracting definitional interpreters (functional pearl)." *Proc. ACM Program. Lang.*, 1, ICFP, 12:1–12:25. DOI: 10.1145/3110256.

P. Van Hentenryck, A. Cortesi, and B. Le Charlier. 1994. "Type analysis of Prolog using type graphs." In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation - PLDI '94.* ACM Press. DOI: 10.1145/178243.178479.

Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Mar. 1995. "Type analysis of Prolog using type graphs." *The Journal of Logic Programming*, 22, 3, (Mar. 1995), 179–209. DOI: 10.1016/0743-1066(94)00021-w.

Neil D. Jones. 1981. "Flow Analysis of Lambda Expressions (Preliminary Version)." In: *International Colloquium on Automata, Languages and Programming*. Springer Berlin Heidelberg, 114–128. DOI: 10.1007/3-540-10843-2_10.

[SW] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon, *The OCaml System* 2022. URL: https://ocaml.org.

Xavier Leroy and François Pessaux. 2000. "Type-Based Analysis of Uncaught Exceptions." *ACM Trans. Program. Lang. Syst.*, 22, 2, 340–377. DOI: 10.1145/349214.349230.

Robin Milner. Dec. 1978. "A Theory of Type Polymorphism in Programming." *Journal of Computer and System Sciences*, 17, 3, (Dec. 1978), 348–375. DOI: 10.1016/0022-0000(78)90014-4.

Benoît Montagu and Thomas P. Jensen. 2020. "Stable Relations and Abstract Interpretation of Higher-Order Programs." *Proc. ACM Program. Lang.*, 4, ICFP, 119:1–119:30. DOI: 10.1145/3409001.

Benoît Montagu and Thomas P. Jensen. 2021. "Trace-Based Control-Flow Analysis." In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 482–496. DOI: 10.1145/3453483.3454057.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, Mass. ISBN: 978-0-262-16209-8.

Stefan Schulze Frielinghaus, Helmut Seidl, and Ralf Vogler. Aug. 2017. "Enforcing Termination of Interprocedural Analysis." *Formal Methods in System Design*, 53, 2, (Aug. 2017), 313–338. DOI: 10.1007/s10703-017-0288-5.

Helmut Seidl and Ralf Vogler. 2018. "Three Improvements to the Top-Down Solver." In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*. Ed. by David Sabel and Peter Thiemann. ACM, 21:1–21:14. DOI: 10.1145/3236950.3236967.

Olin Shivers. 1991. "The Semantics of Scheme Control-Flow Analysis." In: *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (PEPM '91). Association for Computing Machinery, New York, NY, USA, 190–198. ISBN: 0897914333. DOI: 10.1145/115865.115884.

Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. "Static Race Detection for Device Drivers: The Goblint Approach." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. ACM, 391–402. DOI: 10.1145/2970276.2970337.

Andrew K. Wright and Suresh Jagannathan. 1998. "Polymorphic Splitting: An Effective Polyvariant Flow Analysis." *ACM Trans. Program. Lang. Syst.*, 20, 1, 166–207. DOI: 10.1145/271510.271523.