Chamelon : A Delta-Debugger for OCaml

Milla Valnet[1,2,3](✉), Nathanaëlle Courant[3], Guillaume Bury[3], Pierre Chambart[3], and Vincent Laviron[3]

[1] École Normale Supérieure, Université PSL, 75005 Paris, France
milla.valnet@lip6.fr
[2] Sorbonne Université, CNRS, LIP6, 75005 Paris, France
[3] OCamlPro, 75014 Paris, France

Abstract. Tools that manipulate OCaml code can sometimes fail even on correct programs. Identifying and understanding the cause of the error usually involves manually reducing the size of the program, so as to obtain a shorter program causing the same error—a long, sometimes complex and rarely interesting task. Our work consists in automating this task using a minimiser, or *delta-debugger*. To do so, we propose a list of unitary heuristics, i.e. small-scale reductions, applied through a dichotomy-based state-of-the-art algorithm. These proposals are implemented in the free Chamelon tool. Although designed to assist the development of an OCaml compiler, Chamelon can be adapted to all kinds of projects that manipulate OCaml code. It can analyse multifile projects and efficiently minimise real-world programs, reducing their size by one to several orders of magnitude. It is currently used to assist the industrial development of the flambda2 optimising compiler.

1 Introduction

Program errors sometimes occur oten large inputs, of hundreds or even thousands of lines. Identifying and isolating the error is often a long and tedious task, which generally involves manually minimising the size of the input as much as possible. The aim of a minimiser is to automate this work.

Sometimes called delta-debugging, this idea was developed in 1999 by Andreas Zeller [11] in order to isolate the cause of a program error by iteratively applying simplifications. It is defined as a methodology reducing a problem while preserving a certain property—here, the error. The tool thus does not eliminate the error, but on the contrary points to it.

This method is already used for languages such as C, with C-reduce [1], SMT-lib [6], or via implementations of Zeller's original work [11]. Nonetheless, the problem remains well studied. Zeller worked with Hildebrandt [10] to identify the inputs and interactions that cause programs' failure, using Mozilla browser user inputs as case study, and then demonstrated with Cleve [3] that delta-debugging works just as well for identifying errors due to the code itself as to its parameters. Seeing any debugging tasks as special cases of minimisation problems, he uses this method with Choi [2] for thread scheduling failures, and with Cleve [4] to identify which variables and at which execution step the error occurs. Finally, Leitner et al. combine this approach with slicing to reduce the size of failure cases in random test generation. Some also improved the state of the art with machine learning [5], probabilistic algorithms [9], etc.

In OCaml, however, the existing debugging tools are limited to type errors [7]. This project therefore proposes the first general-purpose minimiser for OCaml code, Chamelon. Although initially designed to assist an OCaml compiler development in the industry, such a tool may prove useful for other projects using or manipulating OCaml code. This work makes the following contributions:

- a list of OCaml-specific minimisation heuristics;
- combined with a state-of-the-art technique to perform dichotomy-based minimisations;
- an OCaml implementation supporting multi-file projects and runtime errors available as open-source software;
- with a modular design to support the development of various kinds of OCaml projects.

Outline. Sect. 2 presents the tool usage. Section 3 explains the unitary heuristics proposed to minimise the program, while Sect. 4 explains how they are combined. Section 5 shows extensions of this work.

2 Tool Usage

2.1 Development Context

The tool Chamelon is a delta-debugger for OCaml programs, available as open-source software on GitHub[1]. Earlier results on Chamelon were presented in French [8]. It was originally designed to support the development of the flambda2 optimising compiler[2], developed by OCamlPro and used in particular by Jane Street. Indeed, when flambda2 failed on programs correct according to the standard compiler, identifying the error cause in flambda2 was not always easy. However, Chamelon is built in a modular way, reducing a program size while ensuring an user-given condition, and can be used in various context.

2.2 Usage

```
chamelon input -c command -e error
```

To use `chamelon`, all we need to do is giving it an `input` file, a `command` to execute and an `error`, that is, the string we want to find in the command's standard output. `chamelon` then prints a log of applied transformations in the standard output, and when done, the output is a minimised version of the input,

[1] https://github.com/Ekdohibs/chamelon.
[2] https://github.com/ocaml-flambda/flambda-backend.

such that the command output still contains the error. To minimise a set of files, we only need to provide the command with several inputs. This way, Chamelon can be used in different settings.

A simple real-world use case is available online[3]. By following instructions in `README-CHAMELON.md`, we can make Chamelon reduce the size of an input file trigerring a `Fatal_error` in flambda2, to help understand the origin of the error.

2.3 Experimental Results

The tool is currently used daily at OCamlPro to help flambda2 development for almost a year. It gave results on real cases of failure, significantly reducing the output program size. Among the experimental results, it was able to minimise a 650-lines program[4] that failed to compile into a program of just 6 lines causing the same error, identifying a problem in the optimisation of pattern matching[5]:

```
let offset ~byte_order  byte_n =
  match byte_order with | `Little_endian -> 0 | `Big_endian -> byte_n
let pack_unsigned_16 ~byte_order  =
  __ignore__ ((offset ) ~byte_order 0);
  __ignore__ ((__dummy__ ()) ((offset ) ~byte_order 1));
  __dummy__ ()
```

We also tested the minimiser on larger programs. For example, given a 3842-lines program on which the compiler was failing, the minimiser reduced it to 22 lines, in around thirty minutes on an average laptop. Often, the output can still be minimised by hand. However, the tool automates a large part of the work. Finally, in a multi-file framework, the minimiser is also able to merge or delete files, resulting in a minimised copy of the project that triggered the error.

Note that reducing the size of the program is not the only interesting action of the minimiser. Indeed, when a simplification is not done, it means that it removed the error, which can therefore be exploited. We not only benefit from the size-reducing, but also from the *minimality* of the program with regard to the heuristics.

3 Heuristics

The concept of the approach is to compose and combine different unitary heuristics, applying each of them as much as possible before trying the next. We present here the different heuristics implemented to minimise an OCaml program. It should be noted that, having initially targeted compilation problems, our approach aims much more at identifying errors caused by a certain code structure than by a certain semantics or execution: this therefore guides our choice of heuristics.

[3] https://github.com/Ekdohibs/flambda-backend/tree/chamelon-demo.
[4] https://github.com/janestreet/core_kernel/blob/master/binary_packing/src/binary_packing.ml.
[5] fixed by https://github.com/ocaml-flambda/flambda-backend/pull/1073.

3.1 Suppress Definitions

Delete definitions starting from the end. The first simple heuristics consists in deleting all definitions—of variables, types, modules, etc.—starting from the end. It aims at removing the code located after error's cause, on which the error does not depend.

Replace expressions by dummy values. When definitions cannot simply be removed, we try to replace them with the simplest possible values. The challenge is then to determine which trivial value we want to replace our expression with while respecting type constraint. For ground types, we simply replace expressions of type `int` by 0, those of type `float` by 0.0, those of type `char` by `'0'`, those of type `string` by `""` and those of type `unit` by `()`. For the other types, we used:

```
external __dummy__ : unit -> 'a = "%opaque"
```

Here, `__dummy__ ()` is of type `'a`, and can therefore replace an expression of any type. It is based on the external primitive `opaque`: when compiled, it is considered as a function returning an arbitrary value—here, a function of type `unit -> 'a` because of the annotation. However, at runtime, it behaves like the identity function: for this reason, the value of `__dummy__()` is `()`, causing a type error. When targetting compilation failures, this is not a limitation. However, to generalize the tool's use cases, this problem will be adressed in Sect. 5.

3.2 Simplify Abstract Data Types

Suppress constructors from ADTs. A first heuristic consists in deleting a constructor `Cons` from an algebraic data type. This involves propagating this deletion of in the code: expressions `Cons`$(e_1, \ldots, e_n)$ are replaced by `__dummy__ ()`, and patterns using `Cons` are simply removed.

Delete fields from record types or constructors. When deleting an entire constructor is not possible, we instead delete its fields. After deleting its ith field's definition, we go through the code to delete the ith field in `Cons(e1,..,en)` expressions, and the ith sub-pattern in each `Cons(p1,..,pn)` pattern—replacing variables bound by `pi` with `__dummy__ ()`.

3.3 Simplify Code

Modify attributes. We remove attributes of functions, modules, etc. from the program to make it less verbose. However, `local [never|always]` and `inline [never|always]` to functions can also provide valuable information about the origin of the failure, forcing the compiler's inlining strategies.

Inline functions. Inlining a function, i.e. replacing it with its definition at call site, can lead to additional simplifications.

Flatten modules. Flattening modules means removing variables definitions from `module Name = struct ... end` block. To avoid name conflicts between variables from the module and variables defined in the program, we chose to precede the name of the variable by the name of the origin module : this change is then propagated throughout the program.

3.4 Remove Simplification Artifacts

Situations that would not or only rarely appear in real user code may appear after applying the above heuristics:

Remove dead code. For each variable, module and type, we go, and when not used, we simply delete their definition.

Simplify pattern matching. When the match contains a unique one-variable pattern, we replace `match e1 with x -> e2` by `e2` in which `x` has been textually substituted by `e1`.

Sequentialize function calls. After simplifications, we may obtain a function application of the form `(__dummy__ ()) e1 ... en`. We sequentialise its by evaluating each argument separately, to get non-nested expressions. We use the primitive `external __ignore__ : 'a -> unit = "%ignore"` . We then transform `(__dummy__ ()) e1 ... en` into:
`__ignore__ e1 ; ... ; __ignore__ en ; __dummy__ ()`

Simplify `rec` and unused arguments. After replacing expressions and definitions by `dummmy`, arguments of a function may no longer be used. We then delete them and propagate their deletion to all of the function's call sites. When the ith argument of the function `f` is deleted, all occurrences of `f` are replaced by `(fun x1 ... xn -> f x1 ... xi-1 xi+1 ... xn)`. Similarly, when the function is no longer recursive, we remove the keyword `rec`.

Simplify sequences. Expressions of the form `(); e` are replaced by `e`.

4 The Iteration

A unitary heuristic can possibly be applied at different points in a program: when trying to delete a constructor from an ADT, many constructors are possible candidates. We call "n-th program point" the n-th position, while reading the program's AST, where it can be performed. When trying to apply it at a program point—e.g. deleting one of those constructors, there are three possible cases:

– This simplification does not remove the error: the program has been reduced!
– This simplification removes the error: we do not want to apply it.
– The index of the point is greater than that of the last modifiable point.

We iterate this way: we take as input the program, a heuristic, and a position. We then attempt to apply the heuristics at this position. If minimisation is possible, we iterate over the new program without incrementing the position, since after simplifying the nth point, the next modifiable point is the new nth. If minimisation is not possible, the next position is examined. Finally, if the position is too large, the whole program was examined, so we return.

Dichotomic Optimisation. In Chamelon, this loop is otpimized by dichotomy, as initially suggested by Keller [11], by no longer trying to minimise locations one by one, but rather a set of locations of length 2^n. This method improves efficiency by a factor of 10 on real programs of a few thousand lines.

Heuristics Order. The application order of the different heuristics was determined experimentally, on a small sample of tests, mainly by finishing with the heuristics removing the simplification artefacts. For more robust and efficient scheduling, further research and testing could prove useful.

5 Extensions

Multifiles. In real use cases, a project is made of multiple interdependant files. We have therefore adapted Chamelon to work on such projects:

- First, we try deleting as many files as possible, in the order of dependencies;
- Then, we try merging as much files as possible;
- Finally, each remaining file is minimised with previous methods.

Note that every object modification must be propagated to all dependencies. For example, if an argument of a function `f` is deleted, it must be deleted at each `f` call sites, in each of the program's dependencies. To use Chamelon in multifile mode, we need to provide it with the list of files to minimise, in dependencies order—which can be given by `ocamldep` tool.

Runtime. The work presented so far focused on compile-time errors. However, errors may also occur at runtime. To handle this, we replaced the `__dummy__` values, causing runtime errors, using an algorithm which, given an input type, generates an expression of the same type, as concise as possible.

Compatibility. The implementation uses OCaml compiler libraries to manipulate abstract syntax trees. A compatibility library is implemented, so that changing of compiler version only requires some information about the new AST.

Adding Heuristics. Implementing a new heuristics is low-cost: we only need to write the transformation through existing mappers function for OCaml AST.

6 Conclusion

In the future, an interesting extension would be to make the Chamelon minimiser compatible with `dune`—the OCaml build system. Finally, through its use in real-world examples, we aim at improving existing heuristics and finding new ones, so as to make it more robust, more efficient and faster. In the end, this work combines various minimisation heuristics with a state-of-the-art iteration technique and a modular design, offering the first delta-debugger for and in OCaml, available for its community!

Artifact. The artifact associated to this paper and demonstrating the use of Chamelon on different programs is available at https://doi.org/10.5281/zenodo.12520654.

References

1. C-reduce project. https://github.com/csmith-project/creduce
2. Choi, J.D., Zeller, A.: Isolating failure-inducing thread schedules. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 210–220 (2002)
3. Cleve, H., Zeller, A.: Finding failure causes through automated testing. In: Ducassé, M. (ed.) Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, 28–30 August 2000 (2000). https://arxiv.org/abs/cs/0012009
4. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering, pp. 342–351 (2005)
5. Heo, K., Lee, W., Pashakhanloo, P., Naik, M.: Effective program debloating via reinforcement learning. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, pp. 380–394. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3243734.3243838
6. Kremer, G., Niemetz, A., Preiner, M.: ddSMT 2.0: better delta debugging for the SMT-LIBv2 Language and friends. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 231–242. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_11
7. Sharrad, J., Chitil, O.: Refining the delta debugging of type errors. In: Proceedings of the 33rd Symposium on Implementation and Application of Functional Languages, IFL 2021, pp. 10–19. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3544885.3544888
8. Valnet, M., Courant, N., Bury, G., Chambart, P., Laviron, V.: Chamelon: un minimiseur pour et en ocaml. In: 35es Journées Francophones des Langages Applicatifs (JFLA 2024) (2024)
9. Wang, G., Shen, R., Chen, J., Xiong, Y., Zhang, L.: Probabilistic delta debugging. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, pp. 881–892. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3468264.3468625
10. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**(2), 183–200 (2002). https://doi.org/10.1109/32.988498
11. Zeller, A.: Yesterday, my program worked. today, it does not. why? SIGSOFT Softw. Eng. Notes **24**(6), 253–267 (1999). https://doi.org/10.1145/318774.318946