MLExplain

Kévin Le Bon, Alan Schmitt

MLExplain

Kévin Le Bon (Inria) Alan Schmitt (Inria)

Abstract

MLExplain is a step-by-step interpreter for OCaml that enables the user to inspect both their program's state and the interpreter's state itself. This interpreter is derived from JExplain, a step-by-step interpreter for JavaScript. The original goal of this work is to show that JSExplain can easily be reused with another language. MLExplain also aims to provide the user with a better understanding of the semantics of OCaml.

1 Introduction

The semantics of a programming language can be very complex. When a language has no specification, the semantics is then defined by the implementations of the language, i.e., interpreters and compilers. However even when a specification is available, it can be difficult to understand why the execution of a specific program results to a certain output.

The goal of the *JSExplain* project [2] is to describe the execution of a JavaScript program by showing every step of an interpreter whose behavior is very close to the specification of JavaScript. In this paper, we show how we adapted JSExplain to OCaml.

Unlike JavaScript, OCaml has no official specification. However, OCaml code execution is fairly simple because the number of language constructions is small. We have written an interpreter for OCaml's typed abstract syntax tree (AST). This AST is still close to the source code yet it provides additional crucial information. For instance, it mentions resolved names, which is useful for module and signatures inclusion and mandatory for interpreting features like named parameters and optional parameters in functions. The semantic we give to OCaml is higher level than the one described in *ZINC* [3] virtual machine, i.e., an interpreter for bytecode after compilation.

2 JSExplain

2.1 A Double Debugger

We call a *double debugger* an interpreter that is able to run step by step and display the program's state and current location in source, as well as the state and location in the source of the interpreter itself. JSExplain is a double debugger for JavaScript. The aim of this tool is to help a user to better understand JavaScript's semantics, by providing an interpreter that is very close to the specification of JavaScript. JSExplain's interface highlights the expression of the program currently evaluated as well as the instruction of the interpreter that is executed.

2.2 Architecture

The core of JSExplain is a JavaScript interpreter written in a subset of OCaml. This interpreter is compiled to a subset of JavaScript and instrumented to generate a trace of its execution when run. This trace can then be navigated using a web-based tool.[1]

The subset of OCaml we support for writing the interpreter is purely functional with variables, constants, sequence, conditional, let-binding, function definition, function application (with support for prefix and infix functions), data constructors, records (including record projections, and the "record-with" construct to build a copy of a record with a number of fields updated), tuples (i.e., anonymous records), and simple pattern matching (only with non-nested patterns, restricted to data constructors, constants, variables, and wildcards). For convenience, let-bindings and functions may bind simple patterns (as opposed to only variables). We also support `ppx` extensions for monadic programming, to simplify the handling of errors.

The target language of our compiler is a purely functional subset of JavaScript where there is never any type conversion, where objects are used as records (no use of their `prototype` field), and where tuples are encoded into arrays.

We instrument the compilation to JavaScript by adding code that logs events, namely entering a function, creating a scope, assigning a variable, and exiting a function. Each event captures the state, the stack, and the values of all local variables in scope of the interpreter code at the point where the event gets triggered. Code that is traced is compiled with this instrumentation, whereas code that is not traced, for instance supporting libraries, is directly compiled to JavaScript.

When using the tool, the source JavaScript program is parsed and run through the instrumented interpreter, thus generating a trace. We can then navigate this trace to see the state of the interpreter as well as the state of the interpreted program. Recovering the information about the interpreted code is not completely straightforward. For example, to recover the fragment of code to highlight, we find in the trace the closest previous event that contains a call to function with an argument named `_term_`. This argument corresponds to the AST of a subexpression, and this AST is decorated (by the parser) with locations. Note that,

[1] `https://jscert.github.io/jsexplain/branch/master/driver.html`

for efficiency reasons, we associate to each event from the trace its corresponding `_term_` argument during a single pass, performed immediately after the trace is produced.

Similarly, we are able to recover the state and environment associated with the event. The state of the interpreted program consists of four fields: the strictness flag, the value of the `this` keyword, the lexical environment, and the variable environment. We implemented a custom display for these elements, and also for values of the languages, in particular for objects: one may click on an object to reveal its contents and recursively explore it.

Finally, a web interface is presented to the user, to navigate in the trace. This navigation can be forward and backward, and at all times the source code and interpreter code are highlighted, and their states are displayed. We also provide a way to specify breakpoints, using arbitrary JavaScript expressions, that reference both the interpreter and interpreted programs. One may thus reach a point where some interpreter line is being evaluated with some source value satisfying a predicate, or vice-versa.

Note that JSExplain, and by extension MLExplain, runs a program to its completion to generate the trace, and it is that trace that is explored using the web interface. It thus does not currently offer the ability to alter the state of the program as it is being explored.

3 MLExplain

At first glance, JSExplain seems tightly linked to JavaScript. However, the architecture of the project is quite modular and can be used for a different programming language, by providing both an interpreter written in the subset of OCaml that we support and a way to display OCaml values.

3.1 Two compilers

To interpret OCaml code, we first need to parse and type it. However, our compiler does not allow us to use the standard library nor an external one, as they are not written in the subset of OCaml that we support. As we do not need to trace the parsing nor the typing, we have been able to separate totally the frontend from the interpreter itself and use another compiler to compile it. Figure 1 describes the architecture of the whole application.

The frontend is compiled with *js_of_ocaml* [4], the compiler of the project *Ocsigen* [1]. The largest part of the frontend is a module that serializes the *Typedtree* – the typed AST of OCaml – into a JavaScript object compatible with the backend interpreter's own representation of the AST. We use the OCaml compiler, namely the `compiler-libs` library, to do the actual parsing and typing. We thus have the guarantee that the typed AST we get is the same as the official OCaml compiler AST.

Unfortunately, we cannot directly use the typed AST, as *js_of_ocaml* takes OCaml bytecode as input, which does not include sufficient information such as the names of constructors. After parsing and typing, we thus navigate the AST and call JavaScript functions that create the data in the expected format.

3.2 Compiler limitations

As stated before, the backend of our interpreter is written in a purely functional strict subset of OCaml. As a consequence, it was necessary to encode some features, such as exceptions. To this end, we use a monadic approach and rely on `ppx` extensions to simplify the syntax.

As an example, consider the `option` monad, whose main operator is defined as follows.

```
let bind_option opt f = match opt with
| Some value -> f value
| None -> None
```

To use this operator, one would need to write the following.

```
bind_option
  (function_that_could_fail 0)
  (fun a -> continuation a)
```

Such code can become difficult to read, especially when one is chaining several functions.

The use of `ppx` lets us write much simpler code, which is transformed to the previous version during parsing.

```
let%some a = function_that_could_fail 0 in
continuation a
```

3.3 Web interface changes

The data structures manipulated by MLExplain are very different from those manipulated by JSExplain since they don't interpret the same language. We thus had to modify the web interface so that it can recognize the different syntactic elements of OCaml and pretty print them correctly as well as OCaml values.

Fortunately the web interface is very simple. It is composed of some auxiliary JavaScript files, the main JavaScript file – `navig-driver.js` – containing the logic of the application and the corresponding HTML page. The only modified file is `navig-driver.js`. The resulting tool can be tested online[2] and is shown in Figure 2. The top-left panel shows the interpreted program, here a simple eval function using GADTs, with the state of the program depicted on the right. The bottom panel shows the interpreter, with its state below. One can see that at this point the `eval` identifier is being evaluated, with the values of `r`, `l`, and `b` shown on the right.

[2] `https://docteur-lalla.github.io/mlexplain/branch/master/driver.html`

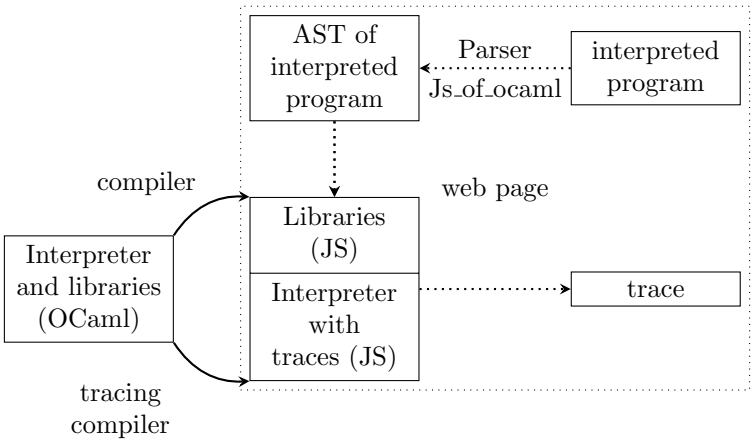

Figure 1: Architecture of MLExplain

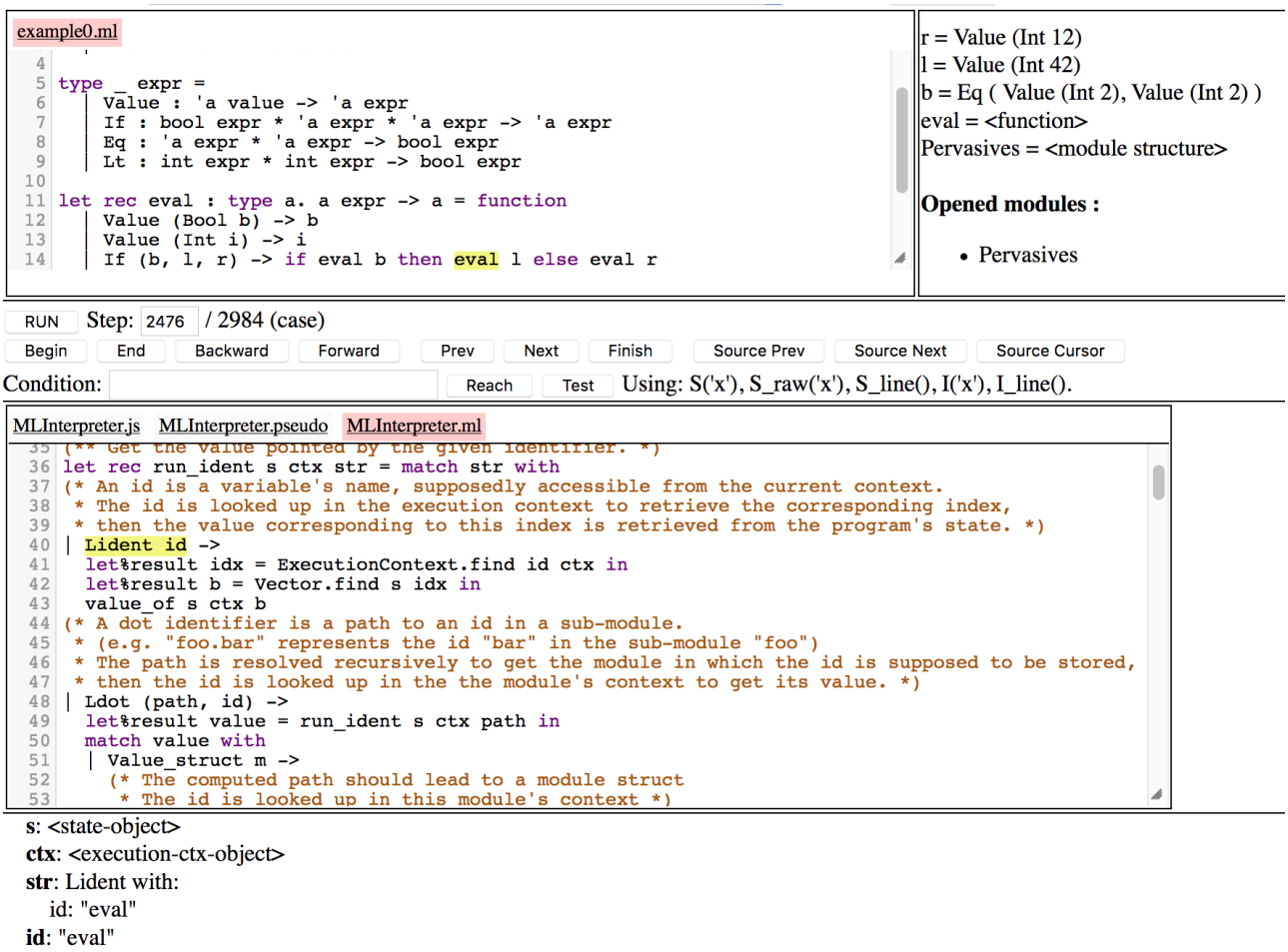

Figure 2: MLExplain

	Comments	Code	Bind calls
Interpreter	164	394	62
Primitives	31	35	0
Execution context	22	13	0
Frontend	41	542	0
Syntax definition	17	65	0
Auxiliary files	72	232	2
Total (no .mli)	392	1621	70
Total (with .mli)	401	1636	70

Figure 3: Statistics about the project's code

3.4 Design Choices

We now motivate our design choices. First, we use JavaScript because MLExplain is a witness that JSExplain can easily be adapted to other languages, by providing three components: an interpreter written in the subset of OCaml that our compiler supports, a parser of the language, and a library to display runtime values. It would be possible to sever all links to the infrastructure provided by JSExplain, but this would require significant work.

Second, it is possible to use an existing debugger to explore how an interpreter works. This is how we initially debugged our JavaScript interpreter and this was the motivation to start working on JSExplain. Indeed, our interpreter is written as a purely functional set of recursive functions that extensively use monadic operations implemented in continuation passing style. It is difficult to correctly navigate such functions, as stepping over a continuation means reaching the end of the program. One thus has to be aware of whether a function call involves a continuation or a usual function. In addition, we did not find an easy way to display the state of the interpreted program, even when relying on the use of fixed names for the term, the state, and the context. The use of fixed names may seem fragile, but as we only rely on a few of them and as the signature of the evaluation functions are uniform, we have not observed any issue with this approach.

4 Statistics

We give some statistics about the project in Figure 3. As can be seen, the largest part of the code is the frontend, in particular the conversion from OCaml's AST to our format. The interpreter is quite short, because the execution of OCaml is actually very simple. Note that we support all of OCaml, with the exception of objects and record patterns, which we have not had the time to implement.

5 Conclusion

We have shown how JSExplain can be easily adapted to other programming language, in this case OCaml. Most of the complexity of OCaml being its typing, writing the interpreter was quite straightforward. The current implementation of our interpreter is very simple and contains a lot of documentation. The architecture of the code is as unsurprising as possible, enabling easy extensions. As the design of our interpreter is very simple, it could be considered as a first step towards a specification of OCaml. MLExplain also provides a comfortable environment to experiment with the semantics of OCaml. It is thus a good playground for future evolution of the language.

As future work, we plan to extend the subset of OCaml supported by our compiler, to be able to directly use the standard library when writing interpreters. We also plan to implement the few OCaml constructs missing from our interpreter. In addition, it would be most useful to document the evaluation order of every OCaml construction to make sure our interpreters follows it. Finally, we want to improve the web interface and to allow the user to load code from multiple files.

References

[1] Vincent Balat, Pierre Chambart, and Grégoire Henry. Client-server Web applications with Ocsigen. In *WWW2012*, page 59, Lyon, France, April 2012.

[2] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. JSExplain: A Double Debugger for JavaScript. In *The Web Conference 2018*, pages 1–9, Lyon, France, April 2018.

[3] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[4] Jérôme Vouillon and Vincent Balat. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44, 08 2014.