Final Year Project Report

Higher-Order Debugging and Logging for OCaml

By

Darius Foo

Department of Computer Science

School of Computing

National University of Singapore

2015/16

Final Year Project Report

Higher-Order Debugging and Logging for OCaml

By

Darius Foo

Department of Computer Science

School of Computing

National University of Singapore

2015/16

Project No: H018600
Advisor: Dr. Chin Wei Ngan
Deliverables:
Report: 1 Volume
Source Code: 1 DVD

Abstract

OCaml, a statically-typed functional language, has a sophisticated type system which eliminates many classes of runtime errors. Programs which compile are guaranteed to be well-typed. However, the strictness of the type system is sometimes an impediment to programmers, especially when it comes to debugging.

In this report, we survey common ways of debugging OCaml programs, including online debuggers and function-level tracing. We propose improvements to the OCaml debugger and implement an interactive frontend for it. We also explore code generation techniques to ease the process of instrumenting code by hand, and describe type system extensions that may augment them.

Subject Descriptors:
D2.5 Testing and Debugging
D3 Programming Languages

Keywords:
ocaml, debugger, tracing, ppx, code generation

Implementation Software and Hardware:
Ubuntu 14.04.2 LTS, OCaml 4.02.3

Acknowledgement

I would like to thank my friends and family for their support, and my advisor, Dr. Chin Wei Ngan, for his guidance and patience. Without them, I would not have be able to progress so far on this project.

Table of Contents

Title **i**

Abstract **ii**

Acknowledgement **iii**

1 Introduction **1**
1.1 Background 1
1.2 Problems 1
1.2.1 Lack of ad hoc polymorphism 2
1.2.2 Lack of accessible tools 4

2 Related Work **5**
2.1 *printf* debugging 5
2.1.1 Debugging libraries 5
2.1.2 Syntactic metaprogramming 6
2.2 Debuggers 7
2.2.1 Exploratory debuggers 7
2.2.2 Algorithmic debuggers 7

3 Towards a Solution **9**
3.1 Debuggers 9
3.2 Function tracing 9

4 Debuggers **11**
4.1 Architecture of `ocamldebug` 11
4.1.1 Events 11
4.1.2 Time-travel 11
4.1.3 Other features 12
4.1.4 Limitations 12
4.2 Extending `ocamldebug` 12
4.2.1 Initial architecture 13
4.2.2 Final architecture 14
4.3 Evaluation 16
4.3.1 Synchronisation of time stamps and debuggee output 16
4.3.2 Exchanging structured data 17
4.3.3 Debouncing server output 17
4.3.4 Debugger modifications 17
4.3.5 Contributions 18

5 Function Tracing **19**
5.1 Overview of ppx 20
5.2 Implementing ad hoc polymorphism 20
5.2.1 Monomorphisation 22
5.2.2 Higher-order functions 22
5.2.3 Abstract data types 23
5.2.4 Shortcomings 23
5.2.5 Lifting transform 24
5.3 Generating function boilerplate 26
5.3.1 Non-recursive tracing 26
5.3.2 Recursive tracing 27
5.4 Customisable tracing 28
5.4.1 Modules for configuration 28
5.4.2 Annotation DSL 29
5.5 Wrapping of call sites 29
5.5.1 Identifying call sites 30
5.5.2 Persisting the ppx environment 30
5.5.3 An implementation 31
5.5.4 Wrapping transform 32
5.6 Evaluation 33
5.6.1 Integration with wider ecosystem 33
5.6.2 Limitations of typed ppx transformations 33

6 Conclusion **35**
6.1 Future work 36

References **37**

A Code Listings **A-1**
A.1 Testing A-1
A.1.1 Test cases A-1
A.1.2 Supporting definitions A-3
A.1.3 Output A-4

B Example of debugger use **B-1**

C Examples of module configuration **C-1**

1. Introduction

1.1 Background

Type systems are essential tools for checking some aspects of program correctness. They are often included with compilers and run automatically on all programs, and for this reason must perform tractable analyses. By nature this means that they can compute only approximations of a program's runtime behaviour (Pierce, 2002). A type system embodies a tradeoff between safety and expressiveness. It may reject programs that it cannot ensure are well-typed, despite the possibility that such programs may not actually fail at runtime.

OCaml, a language in the ML family, has such a type system. It is a relatively advanced language, supporting a mix of the functional, imperative, and object-oriented programming paradigms. Its type system is expressive enough to support parametric polymorphism and many higher-order constructs, yet safe enough that when programs compile, they often work right off the bat.

1.2 Problems

When they do not work, however, debugging OCaml programs can be a slow and tedious endeavor. It has particularly poor support for so-called *printf* debugging[1] due to its lack of ad hoc polymorphism. It also lacks accessible tooling for inspecting programs and reasoning about them.

[1] inspecting the state and execution flow of a running program by printing text to some output channel

1.2.1 Lack of ad hoc polymorphism

Ad hoc polymorphism (Strachey, 1967) is a language feature which allows a polymorphic function to denote a number of heterogeneous implementations, with the exact one depending on the types of arguments it is applied to. `show` from the Haskell Prelude is such a function, producing a string representation of any value it is applied to:

```
show 1
=> 1

show (True, "False")
=> (True, "False")
```

The exact implementation of `show` may vary across types of values, but these differences are abstracted away, and programmers are conveniently allowed to access them all in a type-safe way with a uniform name.

Ad hoc polymorphism is realised differently in many programming languages.

- Overloading (SML, F#, Java)
- Extension methods (C#)
- Implicit coercions (Scala)
- Traits (Rust)
- Typeclasses (Haskell)

Unfortunately, OCaml does not support these features.

Ad hoc polymorphism may also occur at runtime, by inspecting the types of values and accordingly dispatching to different functions. Overloading in this manner is common in many dynamically-typed languages, such as Ruby and JavaScript. OCaml's implementation, however, makes this difficult to achieve as well.

- OCaml primitives are unboxed, as the type system guarantees well-typedness. The absence of type information rules out polymorphic dispatch at runtime.

- Reflection is possible by annotating values with type information[2] or converting them into some uniform representation that may be used for dispatch[3]. This can be mechanically done via a preprocessing step (Madhavapeddy & Gazagnaire, 2009). This approach is invasive and cannot be easily integrated into existing projects which do not use such wrapped representations.

- A limited form of reflection is possible by using low-level primitives to inspect the raw representation of values (Scherer & others, 2008). This is fragile and does not always produce meaningful information.

Consequently, obtaining a meaningful string representation of a value is rather cumbersome in OCaml. Code like the following is commonplace.

```
print_endline (string_of_int 5)
```

`print_endline` expects a string, so we must explicitly coerce arguments we pass it with a *printer*[4]. These are typically verbose, as the coercion functions must be named after one or more of the intervening types.

Printing compound data types is similar. Here we define a variant.

```
type sexp = Str of string | List of sexp list
```

It is typically accompanied by a definition like the following, which would be applied everywhere we print values of the type.

[2] in the spirit of Haskell's `Typeable`

[3] a `typecase` construct, analogous to `instanceof` in Java

[4] a function for converting a value into a string

```
let string_of_sexp sexp =
  let rec aux = function
    | Str s -> "\"" ^ s ^ "\""
    | List ss -> "(" ^ string_of_list aux ss ^ ")"
  in "'" ^ aux sexp
```

All of this makes printing values tedious and cumbersome, especially in a large system with many modules and types.

1.2.2 Lack of accessible tools

printf debugging may be ubiquitous and simple, and useful for tracing a program if applied in a structured manner, but it is not very sophisticated. It clutters code and may not represent control flow accurately if output is buffered or concurrent.

More advanced tools exist to help find bugs, primarily *debuggers*: interactive programs for exploring the state of other programs. OCaml is distributed with such a tool, `ocamldebug` (Leroy et al., 2007). It has many features, including *time-travel debugging*, which is the ability to step backwards as well as forwards.

Unfortunately, `ocamldebug` is not easy to use. Its default interface is a prompt where the user types commands, such as `step`, `next`, and `go`, in order to navigate another program. Only one line of the *debuggee*[5] is visible at a time. The experience of using it may be compared to viewing a program's state through a keyhole, focusing very closely on a small portion at once.

Alternative frontends for Eclipse and Emacs make interactions with it more palatable (via graphical buttons and keybindings), and even add surrounding context when it stops at a checkpoint, but neither improves it drastically, merely wrapping the functionality that the command-line interface provides.

While `ocamldebug` is a impressive work of engineering, its usability and level of interactivity leave something to be desired, and it is not yet a great option for beginners to navigate programs.

[5] the program upon which a debugger acts

2. Related Work

The previous section gave an overview of the default ways of debugging in OCaml. Developers have, however, resorted to various others means to improve their debugging experience and tools. A number of approaches are surveyed here, along the two main axes of function tracing debugging and online debuggers.

2.1 *printf* debugging

2.1.1 Debugging libraries

One way to reduce the number of printer definitions is to keep them abstract. Printing functions for polymorphic structures should be accordingly polymorphic, reflecting the structure they apply to.

For example, here is a higher-order printer which reflects the structure of lists.

```
let string_of_list pr xs =
  let rec aux xs =
    match xs with
    | [] -> ""
    | [x] -> pr x
    | y :: ys -> pr y ^ "; " ^ aux ys
  in "[" ^ aux xs ^ "]"
```

Here is a similar version for tuples.

```
let string_of_tuple pr_a pr_b (a, b) =
  "(" ^ pr_a a ^ ", " ^ pr_b b ^ ")"
```

These higher-order functions are composable. To print a list of tuples of integers, for example, these printers may be partially applied and combined.

```
let result = string_of_list (string_of_tuple
    string_of_int string_of_int) [(1, 2); (3, 4)]
```

This approach is simple, and effective for large projects, where the overhead of defining printers is amortised by the amount of reuse they will see.

The downsides are that it is fairly ad hoc, and specific to a project's data types. It would still result in a significant amount of boilerplate code to write and maintain. That is where the next solution comes in.

2.1.2 Syntactic metaprogramming

Compile-time code generation and metaprogramming can ease the burden of maintaining boilerplate. This can be applied in a variety of ways.

- Shorthand notation may be defined for printers using `cppo`, a C-like preprocessor for OCaml (Jambon, 2009).
- Syntactic extensions with `Camlp4` may stand in for valid fragments of code. They may extend the OCaml syntax in arbitrary ways – an upside for sheer flexibility, but a downside for interoperation with existing language tooling. Prior work has been done to extend the syntax to support easier function tracing (Yeo, 2013b).
- Printers may be mechanically generated from type definitions using `ppx_deriving` (Zotov, 2014).

- `ppx` may be applied in many other ways (Furuse, 2015). We will explore such applications more deeply.

2.2 Debuggers

2.2.1 Exploratory debuggers

Function tracing allows us to view the state of a program retroactively, after it has finished executing. On the other hand, online debuggers give us insight into the state of a program at runtime, often with an interactive interface. This may be useful if, for example, we would like to zoom in on a particular piece of code and reason about it in locality. We may then proceed into subcomputations to check their results, and so on.

We term debuggers which support this mode of operation *exploratory*, as they are guided by human intuition. `ocamldebug` is such a debugger.

2.2.2 Algorithmic debuggers

Algorithmic debugging (Shapiro, 1983) instead sees the progress of a debugger guided by an algorithm rather than a human user's intention or intuition. The interface to this sort of debugger is a series of yes/no questions about function inputs and outputs, posed to the user or a formal specification. Based on the answers, the debugger determines which parts of the program are correct, and which are likely to contain bugs. It navigates programs in this manner until it can unambiguously identify a problematic area.

A concrete application of this technique to OCaml is in a type debugger (Tsushima & Asai, 2014). In contrast to other debuggers, which are meant to help unravel runtime errors, this one diagnoses compile-time type errors.

Type errors in OCaml usually manifest as unsafisfiable constraints, and may appear a distance away from the actual source of error. This can be disorienting to new programmers. The type debugger addresses this with a modified top-down type inference algorithm (which corresponds more directly to human intuition), and asks questions about the possible types of

expressions. This helps locate the actual mistyped expression quickly.

3. Towards a Solution

We have looked at existing ways of debugging and covered why they are less than ideal.

In this project, we aim to improve the state of OCaml debugging.

The rest of this report will describe how we will do so, along the two main axes of online debuggers and *function tracing*.

3.1 Debuggers

Existing methods surveyed generally have orthogonal functionality, in the sense of applying either to online debugging *or printf* debugging.

We explore enhancements to the user interface of the debugger that *unifies* these these two approaches, trying to find a point in the design space which allows us to take advantage of the interactivity and immediacy of an online debugger, as well as the overview that *printf* debugging provides.

This is done by extending `ocamldebug` with a user-interface similar to that of *printf* debugging: a program's log. Rather than rely on single-stepping and breakpoints to navigate a program, a user would simply indicate interesting positions in the log, and `ocamldebug`'s time-travel facilities would take the user to that point in the running program, so inspection of the program's runtime environment can be carried out.

3.2 Function tracing

We also consider improved support for *function tracing*, a specialisation of *printf* debugging that adds structure to the way we inspect program state.

Function tracing treats *functions* as the basic unit of inspection, logging all inputs and outputs, so the hierarchy of calls becomes apparent. Additional functionality for filtering these logs and narrowing the results down to the interesting bits may also be provided.

Our improvements add code generation capabilities using a specialised form of ad hoc polymorphism, easing the process of instrumenting code by hand. We also describe type system extensions which may improve the robustness of our approach.

Finally, we evaluate the possibility of integrating our two enhancements.

4. Debuggers

We first survey the features and inner workings of `ocamldebug`, then elaborate on our enhancements.

4.1 Architecture of `ocamldebug`

`ocamldebug` is the debugger provided in the standard distribution of OCaml from INRIA. It is itself an OCaml program, and its operation is deeply integrated with the official runtime, and in particular, the bytecode interpreter.

To use it, programs to be debugged must first be compiled to bytecode. `ocamldebug` starts debuggees as separate processes and communicates with them, instructing the runtime where (and *when*) to `step` and `go`. It may be seen as a relatively lightweight client to the runtime, providing an alternate means of controlling the execution of programs.

4.1.1 Events

The *event* is the unit of execution that the debugger is concerned with, analogous to the statement of an imperative language. Events may be thought of as positions in source code immediately preceding or following expressions; they also denote the entry and exit points of subcomputations.

4.1.2 Time-travel

The debugger works by maintaining *checkpoints*, which are *events* annotated with a *time stamp*. Time stamps increase monotonically with the debuggee's progress. An event may be associated

with many time stamps. Debugger commands move backwards and forwards *to* time stamps and may navigate them out of order.

4.1.3 Other features

- The debugger communicates with the runtime over a socket using an ad hoc protocol.
- Features not provided by the runtime, such as the notion of breakpoints, are implemented in the debugger.

4.1.4 Limitations

- `ocamldebug` does not support arbitrary code execution at checkpoints. Interactions with debuggees are limited to querying their state, such as the values of variables and the contents of the runtime stack.
- Time-travel debugging does not work seamlessly with effects in general, especially those external to the runtime. This is an important point as OCaml is an impure language. Internal effects such as mutation and randomness may be undone, but printing to a terminal or writing to the file system cannot be. In that case, executing an impure part of a program, then stepping backwards and executing it again may produce a different result.
- `ocamldebug` is deeply integrated with the bytecode interpreter. Nontrivial extensions to it must necessarily be done with a thorough understanding of the OCaml bytecode format and how the runtime works.

4.2 Extending `ocamldebug`

We began our improvements with additions to an existing debugger frontend. `tuareg-mode` (Leroy, Vouillon, Zimmerman, Garrigue, & others, 1997), a mature Emacs package for OCaml, was the chosen starting point.

`tuareg-mode` provides keybindings and an Emacs Lisp API that map almost directly to functionality provided by the default command-line interface, which is also accessible for finer-grained control. As it is an Emacs package, the command-line interface uses `comint-mode`[1] and thus has all of Emacs' editing facilities, including history and completion.

Our idea was to produce a log of the program's execution, in the same way one would run a program to completion in order to see its printouts. This log would be generated in Emacs and augmented with interactive elements, facilitating navigation through the debuggee. This would integrate with existing functionality, such as breakpoints.

4.2.1 Initial architecture

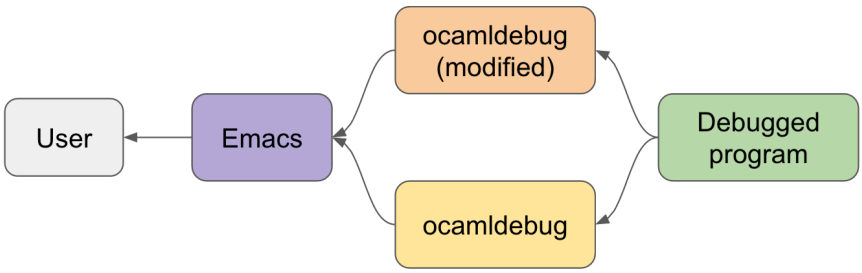

Figure 4.1: Initial architecture, with two copies of `ocamldebug`

The initial plan was to run a separate, modified instance of `ocamldebug` to generate the log. Emacs would communicate with this second process, glean useful information about the running program, and use *that* information to control the first `ocamldebug` instance. This setup is shown in figure 4.1.

To this end, `ocamldebug` was modified so it would single-step continuously to the end of a program and output time stamps along the way. The output of the running program would be interspersed with the time stamps of debugger events in an Emacs buffer. The buffer would then be augmented with links for controlling the debugger interactively.

This worked well as a first cut, but was not very flexible.

[1] a major mode for controlling interactive programs

- It was inefficient, requiring two separate instances of the debugger to be started.
- The second `ocamldebug` instance was unusable for anything other than generating the initial log.
- The interface wasn't much improved, with limited potential for interactivity.
- User input and debugger/program output were interleaved.
- Modifications were crude, and the majority were compiled directly into the debugger.

4.2.2 Final architecture

It was clear that the modifications needed to be restructured. The primary motivation was to separate user input from debugger and program output. This would allow a higher level of interactivity. A useful feature might be to modify the log in response to user input, formatting it differently depending on progress through the debuggee. This wouldn't be possible if there was a static log, generated from just a single run of the debuggee.

This led to the setup shown in figure 4.2.

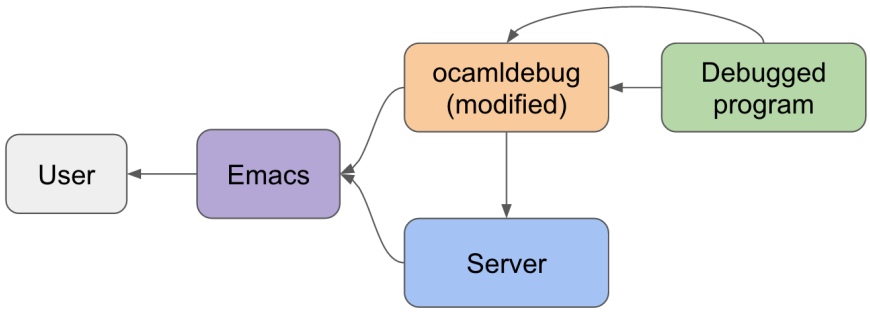

Figure 4.2: Final architecture, with intermediate server

An intermediate program (the *server*) would sit between `ocamldebug` and Emacs. It would communicate with `ocamldebug`, receiving both debugger events and program output from it. This information would be buffered and formatted for display in Emacs, which would remain the means of user input. Direct debugger modifications would now consist of only those required to notify the server of meaningful events.

Logic for formatting output was previously written in Emacs Lisp; we moved this into the server as well. Rather than send unstructured text verbatim to Emacs, we formulated a simple communication protocol[2] and implemented it in OCaml. This way we would benefit from the type-checker and have more confidence in the server code being correct.

The remaining Emacs Lisp would then only consist of *glue* code, tying everything together and handling user interactions. There would now be only one instance of `ocamldebug` running.

All in all, moving in this direction abstracted away the complexity in the debugger and allowed more features and optimisations.

The final result is shown in figure 4.3.

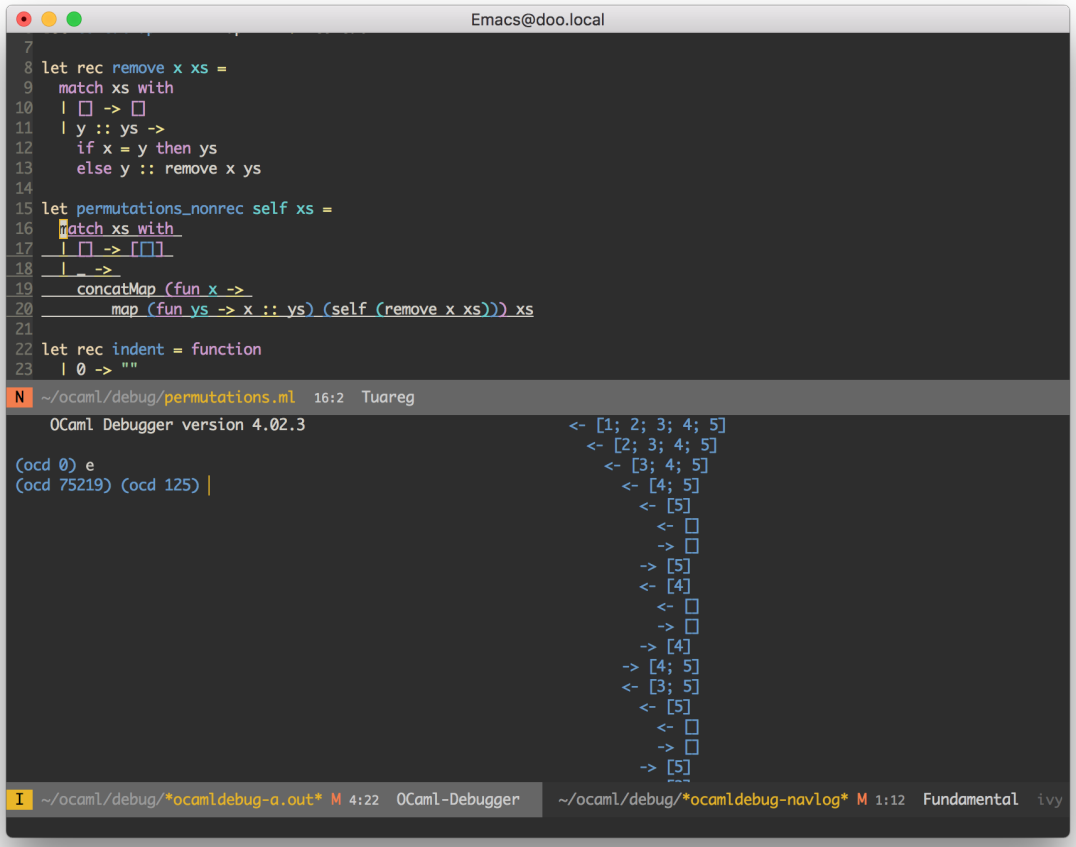

Figure 4.3: `ocamldebug` running in `tuareg-mode`, with a view of the source code on top, the prompt on the left, and an interactive buffer to the right.

[2]based on S-expressions, the lingua franca of Emacs

4.3 Evaluation

The interplay between `ocamldebug`, the server, the debuggee, and Emacs gave rise to substantial complexity. A number of implementation issues, as well as the decisions made to overcome them and the tradeoffs involved, are discussed here.

4.3.1 Synchronisation of time stamps and debuggee output

It is possible to relate debugger time stamps to debuggee events reliably, but relating the former to debuggee *output* is slightly more tricky, as debugger and debuggee run concurrently, and debuggee instructions may result in a printout which only appears after a delay.

However, the following was observed.

- Relevant time stamps are generally sparse: barring the use of low-level I/O primitives, consecutive lines of output have time stamps differing by ~ 5 units.
- The delay is slight: within ± 1 unit of the actual time stamp on average.

Thus this could be solved by uniquely mapping each output line to a small bucket of time stamps. Having this mapping be unique is important for a number of reasons.

- When debugging programs, the same output lines may appear multiple times, for example, when logging the base cases of a recursive function. The textual content of the line isn't enough to uniquely identify it.
- Due to `ocamldebug`'s ability to step backwards, the *exact same output line* may appear multiple times. Knowing that it is not a different line helps us keep track of it when dynamically changing the log.
- Information sent to the server can be optimised. By uniquely mapping the time stamps, the relevant ones can be identified, and only those sent across. In experiments on a small program, only 653 time stamps out of $75,219$ in total were sent and stored with the new scheme: under 1% of what would have been with the old.

4.3.2 Exchanging structured data

Emacs communicates with processes by polling for output, and a single round of polling may only pull in a partial S-expression, which cannot be immediately parsed.

We buffer the result, augment it with a header and footer, and escape it as needed. This ensures that we always have a well-formed S-expression to generate the log with.

This solves the problems introduced by Emacs' polling. It also allows us to handle arbitrarily large inputs, limited only by memory.

4.3.3 Debouncing server output

For the system to support as high a degree of reactivity as possible, we make it respond to every single change in debugger state. That is, when a debugger step occurs,

- A pair of time stamp and outline line is sent to the server.
- The server processes it, then sends the formatted log to Emacs.
- Emacs polls for it, applies formatting, then presents it to the user.
- The user clicking a link at this point will trigger another debugger step.

This is great for making the debugger feel responsive, but it ends up being rather inefficient when single-stepping over a large duration. While the optimisations above reduce the number of events by two orders of magnitude, hundreds of events may still be generated when only a few are really needed.

Our solution is to debounce server output. While it will receive all of the events from `ocamldebug`, the server will send only a small number to Emacs for display. This way Emacs remains responsive, with heavier computation delegated to the processes it controls.

4.3.4 Debugger modifications

On top of the modifications required to communicate with the server, we extended the debugger with a few new features.

- `ocamldebug`'s default output was modified to remove most output (redundant now that a separate window was used), more compactly display a time stamp.

- The `end` command was added, which single-steps to the end of the debuggee. It does this in large strides, and is roughly equivalent to repeatedly executing `step 10000` until the program finishes. This is useful for generating the entire log of a program to navigate with.

4.3.5 Contributions

Bugs encountered in the course of these debugger modifications were reported, and in some cases fixed. A listing of these follows.

- `http://caml.inria.fr/mantis/view.php?id=7015`
- `https://github.com/ocaml/tuareg/issues/66`
- `https://github.com/ocaml/tuareg/pull/74`
- `https://github.com/ocaml/tuareg/pull/75`

We now continue along the other axis, describing techniques that ease function tracing.

5. Function Tracing

We focus now on improving the process of function tracing.

Given some OCaml function `f` to trace, much of the code for printing argument and result values looks the same (barring differences in arity). When factored out, we end up with something like this.

```
let f : int list -> string option =
  fun a ->
    let pr_arg = string_of_list string_of_int in
    let pr_result = string_of_option id in
    Debug.no_1 "f" pr_arg pr_result f a
```

The only variation left is in the explicit coercions. They are difficult to remove due to the lack of ad hoc polymorphism in OCaml – we do not have a way to convert implicitly between types.

Additionally, the burden of writing tracing code correctly still remains, especially when wanting to trace in different ways, for example if we are only interested in the values of certain parameters, or if we want to monitor every recursive call.

In addition to being tedious to write, such code is ad hoc and prone to copy-paste bugs and inconsistencies.

Code generation may solve these problems.

We first discuss how to generate the coercions automatically, by implementing a specialised form of ad hoc polymorphism. Next, we cover the actual code generation, and describe how different tracing configurations are supported.

5.1 Overview of ppx

ppx (Frisch, 2014) is a facility in the OCaml compiler for transforming intermediate syntax trees. It augments the OCaml syntax with a fixed set of annotation nodes, not permitting arbitrary syntactic extension (unlike the previous widely-used preprocessor, camlp4). This allows it to take and return valid OCaml ASTs, thereby making it easier to integrate and use.

The diagram below shows where ppx sits in the compilation pipeline. It operates on the parse tree, and so its transformations are by default purely syntactic, unable to make use of type information.

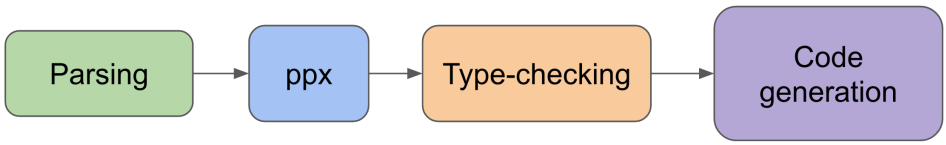

Figure 5.1: Traditional (untyped) ppx in the compiler pipeline

Traditionally, an enhancement such as this would be implemented as a compiler extension. However, that carries with it substantial overhead: the burden of keeping up with upstream work, and having to reconcile incompatible changes.

ppx was chosen for this as preprocessors have access to compiler internals, allowing shallower extensions to be distributed as as libraries. This makes installation and maintenance much easier.

5.2 Implementing ad hoc polymorphism

Considering just the functional languages with type inference, we can group their implementations of ad hoc polymorphism into broad categories:

- Overloading
  - Closed (SML)
  - Open (F#)

- Typeclasses (Haskell)

The objective is to implement a scheme such that we have the function `show`, with the polymorphic type $'a \rightarrow string$. It provides a means of converting any OCaml value into a string.

```
show 1
=> 1

show (true, "true")
=> (true, "true")
```

As we only require this particular function to be overloaded (and we do not yet have a means of allowing overloaded functions of any types to work correctly), we implement SML's closed overloading, in which only primitives are polymorphic.

Following (Furuse, 2015) (Yeo, 2013a), this will be implemented using a *typed* `ppx` transformation – that is, we type-check the compilation unit we are given, to determine parameter types. This allows us to then transform it with type information.

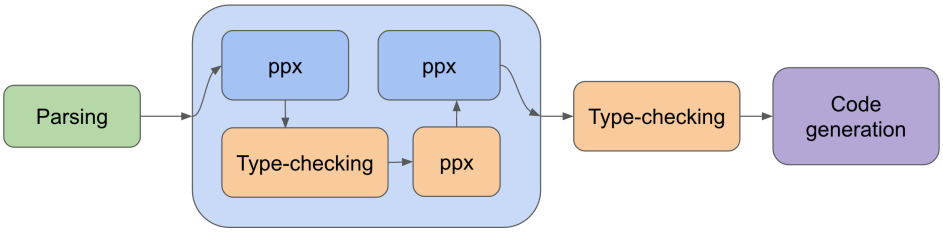

Figure 5.2: Augmented (typed) `ppx` in the compiler pipeline

As the output of the `ppx` process is an untyped parse tree, the types must be stripped before it returns. The second type-checking pass, performed as part of the normal compilation pipeline, ensures that the generated code is well-typed.

5.2.1 Monomorphisation

Ad hoc polymorphism on primitives requires only a simple scheme: we can inspect the types of the argument to `show` and accordingly insert the correct monomorphic printer.

```
show 1

Format.sprintf "%d" 1
```

For compound types, we mechanically compose simpler printers with partial application, using the same technique introduced in 2.1.1.

```
show (true, "true")

Format.asprintf "%a"
  (PolyPrint.Printers.pp_tuple Format.pp_print_bool
    Format.pp_print_string) (true, "true")
```

5.2.2 Higher-order functions

When `show` is used in a higher-order context, we run into a problem: here the *identifier* `show` is a polymorphic function of the type $'a \rightarrow string$. We cannot tell what `show` may be applied to.

```
let apply printer value = printer value in
apply show 1
```

Our solution is to apply eta abstraction.

```
let apply printer value = printer value in
apply (fun x -> show x) 1
```

This does not change semantics, but by wrapping `show` with a lambda function, we take

advantage of the fact that the type of the lambda must be specialised during type-checking, giving us the monomorphic $int \to string$. This lets us use the above scheme to generate the printer.

5.2.3 Abstract data types

Our scheme is not sufficient for abstract data types, whose interface sometimes admits no information about them.

```
(* User-defined *)
module M : sig
  type t
  val thing : t
  val pp_t : Formatter -> t -> string
end = ...
```

We solve this by following conventions laid out by `ppx_deriving` (Zotov, 2014).

```
show M.thing

Format.asprintf "%a" M.pp_t M.thing
```

When asked to `show` a value of the polymorphic type `M.t`, we default to looking for a printer named `pp_t` in module `M`. This behaviour works for the majority of cases, and integrates nicely with `ppx_deriving`. The exact printer to use can still be overridden by manually specifying it in 5.4.

5.2.4 Shortcomings

The scheme described above covers the common cases, but falls short when dealing with polymorphism. The main difficulty is that we can't implement `show` : $\forall' a.' a -> string$ meaningfully without at least some constraints on `'a`, or some notion of the operations it supports (much like a typeclass constraint).

In the following example, `show` is applied to the polymorphic parameter of `stringify`. Without more information available, we default to a constant string.

```
let stringify x =
  show x

stringify 1
=> <polymorphic>
```

This is unsatisfactory, but it is a shortcoming of our implementation within a `ppx` processor: while we can access compiler internals, we cannot modify them, leaving us unable to implement deep changes to the type-checker.

The problem is that we cannot identify call sites, as the architecture of `ppx` leaves us with little options (more on this in 5.5.4).

Instead, we propose a hypothetical type-checker extension which would allow us to get around this.

5.2.5 Lifting transform

We propose a transformation on OCaml syntax trees called the *lifting transform*. It allows us to abstract over function implementations and propagate information from call sites *into* the function.

This transformation has a variety of use cases.

- Abstracting over recursive calls (more on this later)
- Generalising implementations
- Polymorphism

The predicate p marks functions as eligible for transformation. Given an eligible function, we may transform it by abstracting over uses of polymorphic primitives in the body and turning them into parameters. This is expressed in the following inference rule.

$$\frac{\text{let} f = \text{fun } x \to y : T_x \to T_y, p(y)}{\text{let} f = \text{fun } s\ x \to y[a \rightarrowtail s] : T_a \to^a T_x \to T_y}\ [def]$$

Additionally, we change the type of the function, annotating its function arrow $\to^a$. This works as a means of carrying information about this function in the type, marking this type as incomplete.

```
let stringify x =
  show x

let stringify s x =
  s x
```

The earlier rule dealt with function definition sites which used polymorphic primitives. Now, we account for use sites of those functions.

Now that we have transformed function definitions to add a parameter, we must also transform the corresponding call sites. The annotated function arrow provides a means of identifying places where this parameter should be added, allowing us to sidestep the tricky issue of function identity by considering only their type.

$$\frac{(a : T_a \to^a T_x \to T_y)(x : T_x), f(\Gamma) : T_a}{(a\ f(\Gamma) : T_x \to T_y)(x : T_x) : T_y}\ [use]$$

Γ is the static environment of each call site. $f(\Gamma)$ is a means of determining a suitable argument to be inserted where the extra parameter would go. Intuitively, this rule specifies that where there is an annotated function arrow, a suitable argument which depends on the call should be inserted into the call, removing the annotated arrow and making the types consistent again.

With the lifting transformation, we may solve the problem of *lifting* uses of polymorphic functions out, to their call sites.

In this case, p is a check for whether a lambda expression contains a polymorphic primitive. p, which operates on OCaml syntax trees, is inductively defined: it is true for expressions which apply a polymorphic primitive, or lambda expressions $funx- > y$ for which $p(y)$ is true.

Intuitively, it identifies functions which use polymorphic primitives, or functions which in turn use the former, and so on.

f is a constant function that just returns `show`.

In so abstracting over the implementations of polymorphic primitives, we have turned `stringify` into `apply` from earlier, reducing this problem into the earlier one. Thus we may solve this with the same scheme we use for higher-order functions.

5.3 Generating function boilerplate

We make use of `ppx` annotations to indicate that a function should be traced.

```
let plus x y = x + y
  [@@trace]
```

We also define a second polymorphic primitive, `print`. The conceptual implementation is given below.

```
(* Conceptual implementation *)
let print x = show x |> print_endline
```

It may be implemented in this manner with the lifting transform. Currently, however, it is implemented as a primitive as well.

Tracing may be carried out either non-recursively, in which case we only log the topmost call, or recursively, where we log every single intermediate call. For this reason we have two different kinds of annotations, `[@@trace]` and `[@@tracerec]`.

5.3.1 Non-recursive tracing

Generating code for non-recursive tracing is straightforward: we simply print each parameter, followed by the function's result. The new primitive `print` does the heavy lifting of figuring out types for us.

```
let plus x y =
  print x; print y;
  let result = x + y in
  print result;
  result
```

5.3.2 Recursive tracing

A general scheme for tracing recursive functions is as follows.

1. Make function non-recursive by abstracting over recursive call.

2. Take the fixed point of the function ('tying the knot').

3. Add instrumentation.

The following function:

```
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
  [@@tracerec]
```

gets transformed into this:

```
let fact n =
  let fact_original self n =
    if n = 0 then 1 else n * self (n - 1) in
  let rec aux n =
    print n;
    let result = fact_original aux in
    print result;
    result in
  aux n
```

Following the rules, we abstract over the recursive call in `fact`, making it the parameter `self`. `fact` is no longer recursive, now capturing only one step of the recursive computation.

`aux` is the instrumentation function which invokes `fact`. It prints its arguments and results, but more importantly, 'ties the knot' back up, taking the fixed point of `fact`. Having done this, calls to `aux` are now woven into every recursive call, all of which will be traced.

5.4 Customisable tracing

5.4.1 Modules for configuration

Now that we can generate tracing code for all kinds of functions, we want to generalise how tracing is carried out, by allowing the user the freedom to override parts of the process. This is done by allowing `[@@trace]` to take a module parameter.

```
let plus x y = x + y
  [@@trace Config]
```

OCaml modules may contain many kinds of language constructs, types or values. Here we focus on the latter, using an object for late binding so parts of the process can be stubbed or changed. The raw printing of parameter types now delegates to functions within the module.

```
let plus x y =
  let plus_original x y = x + y in
  Config.run2 "plus"
    ("x", show x, x)
    ("y", show y, y)
    show plus_original
```

We augment calls with extra (statically-available) information about each parameter, such as its name and string representation, so that user-defined printing functions can display more information if chosen. We also use `show` in higher-order context here to print the function's result.

Examples of how module configuration can change logging behaviour are given in C.

5.4.2 Annotation DSL

Annotation parameters are not simply limited to one configuration module; they actually form a small DSL, which comprises a sequence of expressions. The configuration module is one example of a valid expression.

Another is a comma-separated list of parameter names, to indicate which are interesting.

```
[@@trace Config; x, y]
```

Interesting parameters are included in tracing output. If this option is omitted, all parameters will be considered interesting.

```
[@@trace Config; x { printer = string_of_int }, y]
```

Parameter-specific configuration is also supported: for example, a specific printer to use can be supplied for each parameter, to override the inferred default.

The DSL is simple and extensible. New features can be implemented by simply interpreting different types of expressions.

5.5 Wrapping of call sites

An additional feature supported is the wrapping of traced functions at their call sites. This allows us to make call site information, such as the file and line number where calls originate, available in traces. It may be stored somewhere and retrieved when a function is actually called. This relaying of information is possible within the configuration framework we have.

```
let rec fact n = ...
  [@@tracerec]

fact 5

(PolyPrint.DefaultTraceConfig.call1
  "fact" (__FILE__, __LINE__) fact 5)
```

The problem is to identify calls to annotated functions and wrap them with a function from the configuration module.

5.5.1 Identifying call sites

We dealt with this problem previously, however there are a number of new challenges in this context. For one, we are dealing now with OCaml functions in general, not simply library functions with known names. The problem of function identity once again comes up: how do we tell that one function value is *the same one* as another? In a higher-order language with aliasing, local function names are far from sufficient.

We also dealt exclusively with functions of a single argument. With functions of any arity, there is additional complexity introduced by partial application, and there being multiple possible call sites for the *same* function value. We must be sure to provide the information for the right call site, which we define to be the first time the function is applied.

5.5.2 Persisting the `ppx` environment

The architecture of `ppx` preprocessors is another source of complexity. We sidestepped this earlier but deal with it now.

As a codebase may be extremely large, `ppx` invocations are designed to operate on single compilation units at a time, modularly and independently. Thus, successive invocations of `ppx` preprocessors do not share memory.

This is an issue because if we have seen a traced function definition, we must remember to wrap its call site, which may not appear in the same compilation unit. The environment

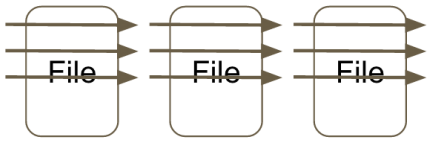

Figure 5.3: Multiple partial passes over one compilation unit at a time

between `ppx` invocations must be persisted somehow.

Fortunately, call sites come first in topological ordering of a codebase by build dependencies. We can exploit this fact when designing our solution.

5.5.3 An implementation

Unfortunately, we are constrained once more by the limitations of a `ppx` implementation, and cannot modify compiler internals in a significant way.

Our present implementation uses a triple of local function name, arity, and type to determine uniqueness of functions. The environment is shared between `ppx` invocations by persisting it to a file.

While it works in practice, this solution is not completely watertight.

- Functions may be wrongly considered equal. This has varying severity depending on how call wrapping is used.
- Parallel builds are affected, as the shared state of the environment introduces implicit dependencies between compilation units. Synchronising when accessing this shared state reduces parallelism, but not synchronising allows races, and may affect the correctness of generated code.
- An automatic means of clearing the environment is missing, as within a `ppx` preprocessor, we no longer have a notion of compilation units, merely syntax trees. As a result, we never know when one cycle of compilation begins and ends, and when it is safe to clear the enrivonment. This may be resolved by manually deleting the environment file and performing a fresh build, but this may not be desirable if there is a large codebase to be

built.

To address these shortcomings, we also propose a type system extension, similar in spirit to that used in 5.2.5 to identify call sites, once again utilising types to propagate call site information.

An additional benefit to using the type system this way is that we may persist the `ppx` environment using the same facilities which the OCaml compiler does: interface (mli/cmi) files. This resolves the issues with parallel builds and stale configuration.

5.5.4 Wrapping transform

We propose another transformation on OCaml syntax trees called the *wrapping transform*. Given an annotated function, this allows us to precisely tell when it is first applied, so we may wrap it at that point.

The first inference rule simply translates information from a `[@@trace]` annotation into a starred arrow type. This allows a function's annotated state to be propagated via the type system, and also persisted to its interface (mli/cmi) file.

$$\frac{\text{let } f = \text{fun } x \to y : T_x \to T_y \text{ [@@trace]}}{\text{let } f = \text{fun } x \to y : T_x \to^* T_y} \; [trace]$$

Traced functions must then be unwrapped at call sites to lose their starred arrow. The next two rules do precisely this, distinguishing saturated from partial application, so as to wrap the function at the correct (first) application site.

$$\frac{(a : T_1 \to^* T_2 \to ... \to T_n \to T_r)(b_1 : T_1)...(b_n : T_n)}{call_n \; a \; b_1...b_n : T_r} \; [call_n(saturated)]$$

In the partial application case, we abstract over the remaining parameters with a lambda function.

$$\frac{(a : T_1 \to^* T_2 \to ... \to T_n \to T_r)(b_1 : T_1)...(b_m : T_m), m < n}{\text{fun } b_{m+1}...b_n \to call_n \; a \; b_1...b_n : T_r} \; [call_n(partial)]$$

This scheme may be further generalised by extending the starred arrow with additional information, should more need to be propagated.

5.6 Evaluation

5.6.1 Integration with wider ecosystem

Implementing a typed `ppx` preprocessor is a relatively unexplored means of implementing a library, and in the course of this project bugs were discovered and reported in many parts of the ecosystem (compiler, Merlin, ocamlbuild, ocamlfind, libraries, preprocessors, and testing framework). A listing of these follows.

- `https://github.com/whitequark/ppx_deriving/issues/61`
- `https://github.com/the-lambda-church/merlin/issues/483`
- `https://github.com/whitequark/ppx_deriving/issues/73`
- `https://github.com/mjambon/cppo/issues/27`
- `https://github.com/ocaml/ocamlbuild/issues/66`
- `https://github.com/mirage/alcotest/pull/51`
- `https://bitbucket.org/camlspotter/typpx/issues/1`

Not all were fixed, sometimes due to deeper architectural issues. In general, typed `ppx` preprocessors are still a relatively new means of implementing compiler extensions, and some incompatibilities should be expected to result from their use.

5.6.2 Limitations of typed `ppx` transformations

A number of caveats underlie the use of `ppx` processors, typed and untyped.

- They have a fixed position in the compiler pipeline. This means that they may only transform parse trees or typed syntax trees. Changing the actual workings of the type system or augmenting interface (mli/cmi) files with more information is not possible.

- They plug into the build system in a very specific manner. Their architecture restricts them to processing single compilation units at a time, independently of others. `ppx` preprocessors also take and return serialised syntax trees, with no actual notion of the compilation unit they are operating on.

- Typed transformations cannot be used in the toplevel, or in environments where program phrases are processed in isolation (such as Merlin 2.3.1), as this requires access to the full typing environment of a program in order to insert printers. It is also infeasible to transfer huge chunks of state like the typing environment between toplevel phrases.

- Typed transformations are sensitive to the order in which other preprocessors are applied, as they requires a full picture of the typing environment in order to operate. For example, this preprocessor should be applied only after `ppx_deriving`, because references to `ppx_deriving`-generated functions would fail to type-check if it hadn't already run.

- `ppx` preprocessors which generate code do not interoperate seamlessly with `ocamldebug`, as generated code is absent from actual source files, and does not contain the same debugging information that regular compiled code does. This makes it difficult to map generated lines to source lines.

- `ppx` preprocessors are in general specific to certain compiler versions, as they depend on compiler APIs, but typed preprocessors are more so, as they depend not only on the syntactic portions, but on the typed portions as well.

6. Conclusion

OCaml has an advanced type system which eliminates whole classes of errors at compile-time, but its rigidity and strictness may sometimes be an impediment to programmers, especially when quick-and-dirty modifications for the purposes of debugging programs are required. Thus, there is a pressing need for better debugging tools, especially for the language to be made more approachable to beginners.

We have surveyed many ways of debugging OCaml programs, and begun by implementing an improved, interactive front end for `ocamldebug`, the debugger that ships with the standard distribution of OCaml.

Following that, we investigated the use of a typed `ppx` transformation for automatically generating code required for function tracing, building in a means of customising it to a variety of needs, and implementing a specialised form of ad hoc polymorphism in the process. There is much potential for compiler extensions distributed as libraries, and this effort only scratches the surface.

We also proposed a number of type system extensions which could make up for the shortcomings of the current implementation, which is limited by its nature as a `ppx` preprocessor.

Having explored the state of debugging in OCaml, on both the axes of online debugging and function tracing, we believe that the situation a little improved by our efforts, but that there is much more room left to innovate. We hope that our enhancements will be eventually be integrated into a cohesive and interactive environment for exploring OCaml programs.

6.1 Future work

Further work in this direction would likely involve integrating the two halves of this.

While the modifications to `ocamldebug` unify the axes of function tracing and online debugging somewhat, they are mostly incompatible with the `ppx` code generation enhancements, for the reasons listed in 5.6.2: generated ASTs do not existing in source code and do not have debugging information which a debugger would use to identify a source line. Thus, they cannot be stepped through.

Next steps on this project would thus be:

- To investigate the OCaml bytecode format
- Figure out how debugging information is stored and preserved in source code
- Restore this debugging information by somehow annotating generated syntax trees

This would be implemented as a compiler extension of some sort.

Robustness of the implementation could also be improved by utilising a compiler extension rather than a `ppx` preprocessor. This may not, however, be desirable, due to the overhead of maintaining a compiler fork. Further research may thus be done into how to improve the implementation without sacrificing the convenience of distribution as a `ppx` extension.

References

Frisch, A. (2014). Syntax extensions without camlp4.

Furuse, J. (2015). *Typeful ppx and value implicits*. ICFP.

Jambon, M. (2009). Cppo: the c preprocessor for ocaml.

Leroy, X., Vouillon, J., Zimmerman, I., Garrigue, J., et al. (1997). Emacs ocaml mode.

Madhavapeddy, A., & Gazagnaire, T. (2009). Dynamic types for ocaml.

Minsky, Y., Madhavapeddy, A., & Hickey, J. (2013). *Real world ocaml*. CA, USA: O'Reilly Media.

Pierce, B. (2002). *Types and programming languages*. MA, USA: MIT Press Cambridge.

Scherer, G., et al. (2008). Batteries included.

Shapiro, E. Y. (1983). *Algorithmic program debugging*. Cambridge, MA, USA: MIT Press.

Silva, J. (2011). A survey on algorithmic debugging strategies. *Adv. Eng. Softw.*, *42*(11), November, 2011, 976–991.

Strachey, C. (1967). *Fundamental concepts in programming languages*. Lecture notes for International Summer School in Computer Programming.

Tsushima, K., & Asai, K. (2014). *Report on an ocaml type debugger* (Technical report). Ochanomizu University.

Yeo, J. (2013a). Fiddling with the ocaml type system.

Yeo, J. (2013b). A syntax extension to make ocaml debugging easier.

Zotov, P. (2014). Type-driven code generation for ocaml.

A. Code Listings

A.1 Testing

We tested the debugger on compact programs with particularly large asymptotic complexity. As they can generate very large printouts with relatively small inputs, they were suitable for assessing the robustness of the implementation. Indeed, many issues would not have been diagnosed if we had only used small inputs.

A.1.1 Test cases

We begin with the following function for computing all the permutations of a list.

```
let rec permutations = function
  | [] -> [[]]
  | xs ->
    concatMap (fun x ->
      map (fun ys -> x :: ys) (permutations (remove x xs))) xs
```

We would like to trace it recursively, so we transform it manually following the technique outlined in 5.3.

```
let permutations xs =
  let nesting = ref 0 in
  let rec aux xs =
    nesting := !nesting + 1;
    print_endline @@ indent !nesting ^ "<- " ^ string_of_list string_of_int xs;
    let result = permutations_nonrec aux xs in
    print_endline @@ indent !nesting ^ "-> " ^ string_of_list (string_of_list string_o
    nesting := !nesting - 1;
    result
  in aux xs
```

The function is then invoked as follows.

```
print_endline @@
  string_of_list string_of_int (permutations [1; 2; 3])
```

When this program is compiled to bytecode, we use the modified `ocamldebug` to interact with it.

Supporting definitions are given below. A full example of debugger interactions is given in appendix B.

A secondary test case was a function for computing edit distance, due to (Minsky, Madhavapeddy, & Hickey, 2013). A similar technique is used to log its inputs.

```
let edit_distance_nonrec self s t =
    match String.length s, String.length t with
    | (0, x) | (x, 0) -> x
    | (len_s, len_t) ->
      let u = drop_suffix s 1 in
      let v = drop_suffix t 1 in
      let cost_to_drop_both =
        if s.[len_s - 1] = t.[len_t - 1] then 0 else 1
      in
      min (self u t + 1) @@
        min (self s v + 1) (self u v + cost_to_drop_both)

let edit_distance s t =
  let nesting = ref 0 in
  let rec aux s t =
    nesting := !nesting + 1;
    print_endline @@
      indent !nesting ^ "<- \"" ^ s ^ "\" \"" ^ t ^ "\"";
    let result = edit_distance_nonrec aux s t in
    print_endline @@
      indent !nesting ^ "-> " ^ string_of_int result;
    nesting := !nesting - 1;
    result
  in aux s t
```

A.1.2 Supporting definitions

`concatMap` is the composition of `List.map` and `List.concat`. It ensures that nested lists produced by a `map` operation are flattened.

```
let concatMap f x = map f x |> concat
```

`remove` removes a value from a list.

```
let rec remove x xs =
  match xs with
  | [] -> []
  | y :: ys ->
    if x = y then ys
    else y :: remove x ys
```

Given an integer, `indent` outputs twice that number of spaces.

```
let rec indent = function
  | 0 -> ""
  | n -> " " ^ indent (n - 1)
```

`string_of_list` is a printer that converts a list into a string.

```
let string_of_list pr xs =
  let rec aux xs =
    match xs with
    | [] -> ""
    | [x] -> pr x
    | y :: ys -> pr y ^ "; " ^ aux ys
  in "[" ^ aux xs ^ "]"
```

`drop_suffix` removes a number of characters from the end of a list.

```
let drop_suffix str n =
  String.sub str 0 (String.length str - n)
```

A.1.3 Output

```
<- [1; 2; 3]
  <- [2; 3]
```

```
      <- [3]
        <- []
        -> [[]]
      -> [[3]]
      <- [2]
        <- []
        -> [[]]
      -> [[2]]
    -> [[2; 3]; [3; 2]]
    <- [1; 3]
      <- [3]
        <- []
        -> [[]]
      -> [[3]]
      <- [1]
        <- []
        -> [[]]
      -> [[1]]
    -> [[1; 3]; [3; 1]]
    <- [1; 2]
      <- [2]
        <- []
        -> [[]]
      -> [[2]]
      <- [1]
        <- []
        -> [[]]
      -> [[1]]
    -> [[1; 2]; [2; 1]]
  -> [[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1]]
[[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1]]
```

B. Example of debugger use

Following (Silva, 2011), we try the debugger out on a program which implements insertion sort. The program contains a subtle bug, so we start `ocamldebug` to try and locate it.

```
let rec insert (x : int) (xs : int list) =
  match xs with
  | [] -> [x]
  | y :: ys ->
    if x >= y then
      (x :: y :: ys)
    else (y :: (insert x ys))

let rec insort = function
  | [] -> []
  | x :: xs -> insert x (insert xs)
```

After transforming the program to log every call, we run it to completion and end up with the following trace.

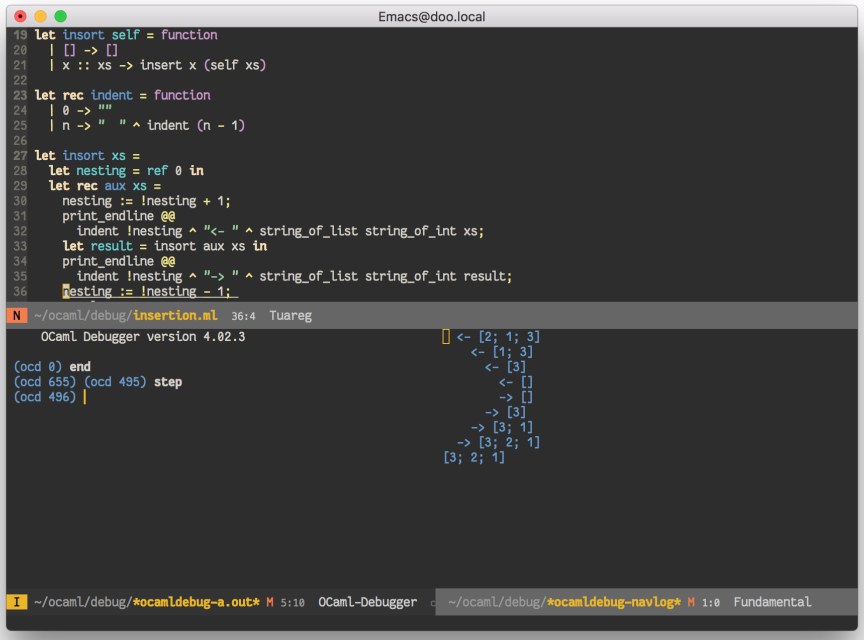

Figure B.1: The base cases look correct, but it seems that the input `[1; 3]` results in the output `[3; 1]`. We click on `[3; 1]` in the right window, then step forward to get to the point after the `insort` has returned.

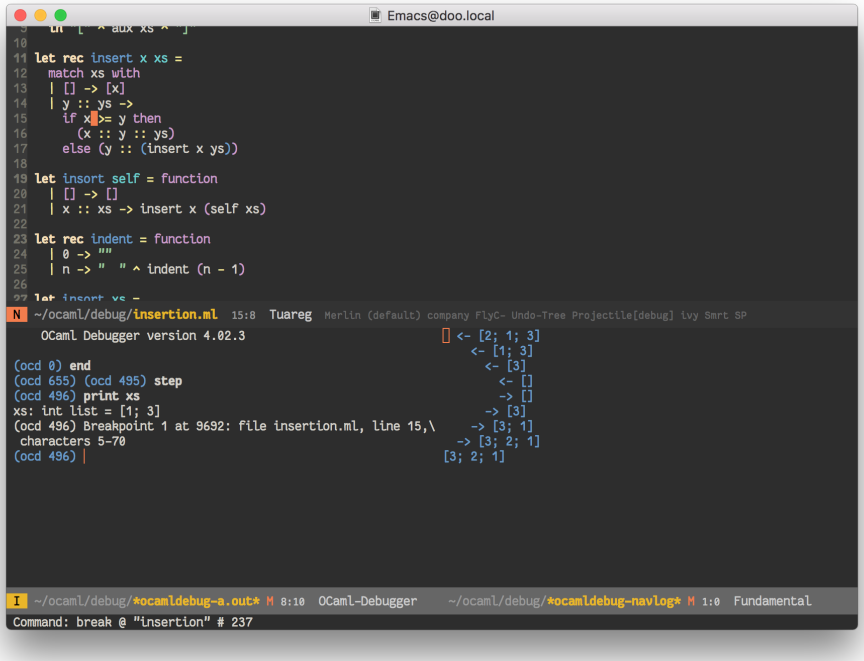

Figure B.2: `xs` has value `[1; 3]`, so we put a breakpoint in the recursive case of `insert`.

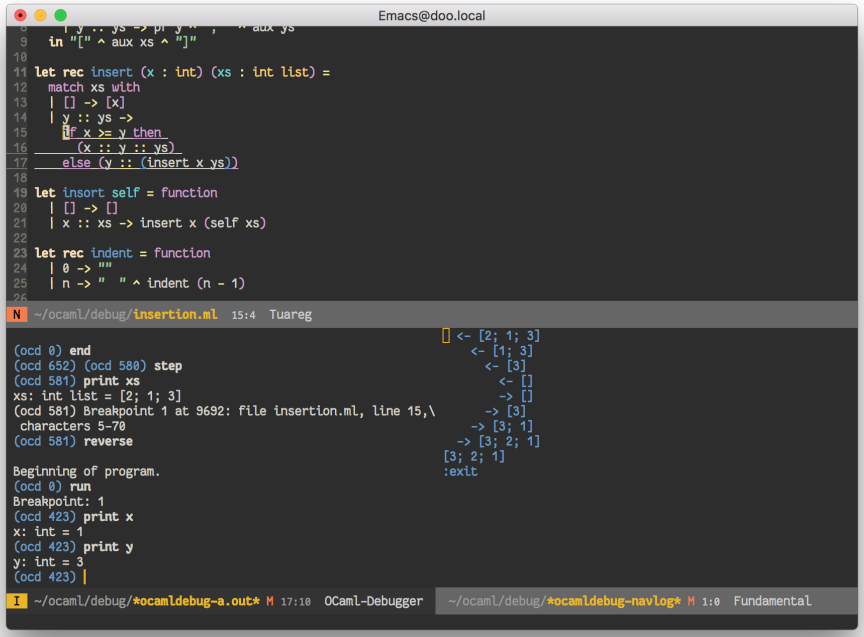

Figure B.3: Now we print the values of `x` and `y`. It seems that we should be ordering `x` first, i.e. entering the consequent. However, we enter the alternative here instead. Therefore, the bug is that the condition is wrong; it should be `x <= y`.

C. Examples of module configuration

Output when using `[@@trace Default]`:

```
fact <- n = 3
fact -> 6
```

Output when using `[@@tracerec Default]`:

```
fact <- n = 3
fact <- n = 2
fact <- n = 1
fact <- n = 0
fact -> 1
fact -> 1
fact -> 2
fact -> 6
```

Output when using `[@@tracerec Recursive]`:

```
fact <- n = 3
  fact <- n = 2
    fact <- n = 1
      fact <- n = 0
      fact -> 1
    fact -> 1
  fact -> 2
fact -> 6
```

Output when using `[@@tracerec Minimal]`:

```
fact 3
  fact 2
    fact 1
      fact 0
      = 1
    = 1
  = 2
= 6
```