Direct Interpretation of Functional Programs for Debugging

John Whitington   Tom Ridge

University of Leicester
UK

`{jw642,tr61}@le.ac.uk`

We make another assault on the longstanding problem of debugging. After exploring why debuggers are not used as widely as one might expect, especially in functional programming environments, we define the characteristics of a debugger which make it usable and thus widely used. We present initial work towards a new debugger for OCaml which operates by direct interpretation of the program source, allowing the printing out of individual steps of the program's evaluation. We present OCamli, a standalone interpreter, and propose a mechanism by which the interpreter could be integrated into compiled executables, allowing part of a program to be interpreted in the same fashion as OCamli whilst the rest of the program runs natively. We show how such a mechanism might create a source-level debugging system that has the characteristics of a usable debugger (such as being independent of its environment) and so may eventually be expected to be suitable for widespread adoption.

1 Introduction

This paper describes early work on a new approach to debugging programs written in the functional programming language OCaml [25]. This approach is informed by a look at the history and characteristics of debugging, a mild astonishment at the lack of use of debuggers in the functional programming community in particular, and the identification of characteristics a usable debugger must have. In addition to presenting the work which has been done, we give informed speculation on the final shape of our debugger, since the implementation is not yet complete. In this section, we look at the problem of debugging from a historical perspective, then go on to discuss debuggers for modern functional programming languages. In Section 2, we decide how to proceed – what our principles will be – and give examples of how the interpreter might be used. Section 3 presents OCamli, an interpreter for OCaml programs, and describes its architecture. We propose a possible interface for the debugger, based upon OCaml syntax extensions. Finally, in Sections 4 and 5, we assess what has been done and discuss future work.

1.1 The debugging problem

As early as in 1965, surprise was expressed that the shift from machine code to assembly language to compiled languages to block-structured compiled languages, and the simultaneous vast improvements in computing power and cost, had not led to as great a reduction in the frequency or severity of bugs. Halpern, in "Computer programming: the debugging epoch opens" [22], writes *That tendency to err that programmers have been noticed to share with other human beings has often been treated as if it were an awkwardness attendant upon programming's adolescence, which like acne would disappear with the craft's coming of age. It has proved otherwise ... Many of us expected compiler languages to eliminate*

all bugs except those so glaring as to leap to the first fresh eye cast on the program. ... An unfriendly behaviorist studying programmers might conclude that we deliberately elaborate our tasks so as to keep the bug rate constant." [22]. Balzer, in his 1969 description of the Rand Corporation's EXDAMS debugger [2], explains that this surprise was widespread in the industry, at a time when more and more large programs were being written in modern compiled languages: *"With the advent of the higher-level algebraic languages, the computer industry expected to be relieved of the detailed programming required at the assembly-language level. This expectation has largely been realised. Many systems are now being built in higher-level languages (most notably MULTICS). ... However, the ability to debug programs has advanced but little with the increased use of these high-level languages."* [2]. Hamlet, writing in 1983, gives a possible reason – the lack of attendant progress in debugging paradigms, suggesting that high level languages might require different kinds of debugging tools rather than mere analogs of existing ones: *"Debugging techniques originated with low-level programming languages, where the memory dump and interactive word-by-word examination of memory were the primary tools. 'High-level' debugging is often no more than low-level techniques adapted to high-level languages."* [24]. It is fascinating to see Halpern, quoted earlier, writing on the same topic forty years later, in 2005: *"The most remarkable thing about debugging today is how little it differs from debugging at the dawn of modern computing age, half a century ago. ... We've made little progress in debugging methods in half a century, with the result that projects everywhere are bogged down because of buggy software."* [23]

Debuggers are still not as widely used as one might expect, even in difficult domains. In a paper on debugging practices for complex legacy systems, Regelson and Anderson write: *"The major item noted by survey respondents was that few people really have learned to use the capabilities of their debuggers"* [35]. Debugging in industry is sporadic. Parnin and Orso [31], writing specifically about automated debugging techniques, say: *"Although potentially useful, most of these* [debugging] *techniques have yet to demonstrate their practical effectiveness. One common limitation of existing approaches, for instance, is their reliance on a set of strong assumptions on how developers behave when debugging."* [31]. This limitation of a "reliance on a strong set of assumptions" as the key to understanding why people do not use debuggers is a recurring theme. For example, Brady, in a paper about a debugging tool for experienced users [3], explains that prior systems tried too hard to be approachable for novices, and the verbosity of their commands alienated the experienced. He writes: *"In a debugging program it is of prime importance that the program be simple, flexible, and highly efficient to use."* [3]. Evans and Darley, in their 1966 survey of online debugging systems, agree, explaining that when designing the interface to a breakpoint-based debugger: *"Here, as in other aspects of on-line work, convenience is critical."* [13]. Eisenstadt [12] lists three principles for the usability of a debugger:

- *Allow full functionality at all times. Debugging environments that prevent access to certain facilities make matters worse.*
- *Viewers should be provided for "key players" (any evaluable expression) rather than just for "variables".*
- *Provide a variety of navigation tools at different levels of granularity.* [12]

Grishman [19] echoes the first principle, arguing that a debugger is at its best when it is at its most widely applicable: *"...to facilitate maintenance, the same program was to be usable in both batch and interactive modes. Second, to facilitate distribution, the system had to be usable without any modification to the operating system."* [19].

Let us now look at the debugging of programs written in functional languages.

1.2 Debugging functional programs

We should like to build a new debugger for the functional language OCaml, trying to learn lessons from the history of debugging. Some of these lessons will no doubt be language-agnostic, but we expect functional languages to have special requirements. And so, we should take a tour of existing debuggers for functional languages and examine to what extent they are usable and used.

Debuggers for functional languages have often followed the pattern of those for imperative languages, even though the mental model of evaluation as expression-rewriting is different. Concepts such as breakpoints appear often, for example. Such debuggers come in several flavours. Some work by extending low-level executable debuggers such as GDB or LLDB with extra routines to allow reconstruction of expressions, some modify the program as it is being compiled, inserting information which can be used by a specialised debugging program, and some work simply by providing macros or extra routines for debugging or logging.

Of course, most functional programming languages have a Read-Eval-Print Loop, which is used not only for learning and programmming-by-experimentation but also for light testing and debugging. Debuggers for functional languages aim to provide facilities over and above the REPL. The limitation of the REPL-as-debugger approach is that debugging often occurs due to an unexpected failure in production, rather than something the programmer provokes deliberately (which we would probably call testing). It is worth noting a practical point: in many languages, it is possible to build a REPL automatically with all libraries and modules used in a project linked in, for example by typing `make repl` instead of just `make` – a boon for usability.

Standard ML Refreshingly, debugging was considered, at least in passing, during the early stages of the design of Standard ML, as Hall and O'Donnell [21] quote Milner [28] recalling: *"ML does not use lazy evaluation; it calls by value. This was decided for no other reason than inability to see the consequences of lazy evaluation for debugging (remember that we wanted a language which we could use rather than research into), and the interaction with the assignment statement, which we kept in the language for reasons already mentioned."* [21][28]. However, the history of debugging tools for Standard ML has not always followed this pattern. Wadler [47] records the story of Tolmach and Appel's debugger [42], which was deeply intertwined with the compiler and runtime of the SML/NJ Standard ML compiler. As the SML/NJ implementation evolved, the debugger fell out of step, and is no longer available. Standard ML developers *"must return to older, more manual debugging methods"* [47]. This is a reminder that keeping a tool which is not part of the standard language toolchain up-to-date requires either frequent modification, or a design which is fundamentally distanced from the language. Of course the spectrum of effort required to update a debugger for a new version of the language is broad. It is likely any tool other than a GDB-style one (operating solely on executables) will always require some updating with each new toolchain release. One practical way to ensure this, if only socially, is to make it part of the toolchain.

The Poly/ML implementation of Standard ML contains an interactive debugger which operates not in a separate environment, but within the usual REPL. For example, here we set a breakpoint on a list reversal function, and ask for the values associated with some names:

```
Poly/ML 5.7.1 Release
> PolyML.Compiler.debug := true;          initialise the debugger
val it = (): unit
> fun rev [] = []
#   | rev (h::t) = rev t @ [h];
```

```
val rev = fn: 'a list -> 'a list
> PolyML.Debug.breakIn "rev";              enable the debugger on our function
val it = (): unit
> rev [1, 2, 3, 4];
function:rev
debug > h;                                 ask for values at the debugger prompt
val it = ?: 'a
debug > t;
val it = [?, ?, ?]: 'a list
```

Notice, though, that even our simple polymorphic list reversal prevents the Poly/ML debugger from printing out the full details of the values we would like to see (one can give the type manually, but giving the wrong type can crash Poly/ML, the documentation advises).

F#   Microsoft's F# [40] is an example of a functional language tightly integrated into (and shipped by default with) a platform of frameworks, libraries and so on, based on the Common Language Runtime [27]. Thus, we would expect F# to be an interesting case when examining debugging functional programs in a broadly imperative scenario. The official guidance on debugging F# [16] is, however, a little disheartening:

> *Debugging F# is similar to debugging any managed language, with a few exceptions:*
>   • *The Autos window does not display F# variables.*
>   • *Edit and Continue is not supported for F#. Editing F# code during a debugging session is possible but should be avoided. Because code changes are not applied during the debugging session, editing F# code during debugging will cause a mismatch between the source code and the code being debugged.*
>   • *The debugger does not recognise F# expressions. To enter an expression in a debugger window or a dialog box during F# debugging, you must translate the expression into C# syntax. When you translate an F# expression into C#, make sure to remember that C# uses == as the comparison operator for equality and that F# uses a single =.* [16]

So the advantage of having a full IDE and a widely-used platform for the functional language to sit within is tempered by compromised support for debugging, at least in this case.

OCaml   We shall now look at typical methods used for debugging in the OCaml community, in addition to the use of the REPL for debugging-like tasks which we have already highlighted.

Inserting `printf` statements is a popular method of informal debugging and logging in many languages on many platforms. However, OCaml (unlike, for example, Haskell or Java), has no generic mechanism for printing user-defined data types. So one is limited to printing only parts of the data – such as strings or numbers, or forced to use custom printers, or limited to a library whose purpose is to provide custom printers. Such restrictions can be painful. Nonetheless, inserting `printf` statements is an example of a debugging mechanism which, whilst it may not always be effective, is at least available in almost all circumstances. Perhaps it is this aspect of its usability which explains its enduring popularity.

The OCaml REPL has a very basic tracing mechanism. For example, here we define a simple function and the tracer displays inputs to and outputs from the function as it runs:

```
# let rec f x = function 0 -> x | n -> f (succ x) (pred n);;
val f : int -> int -> int = <fun>
# #trace f;;                               enable tracing for our function
f is now traced.
# f 0 1;;                                  invoke the function
```

```
f <-- 0
f --> <fun>
f* <-- 1
f <-- 1
f --> <fun>
f* <-- 0
f* --> 1
f* --> 1
- : int = 1
```

Too much has been lost in the compilation process to provide more information about the evaluation process of the expressions. In particular, currying is not preserved. Values having polymorphic types cannot be printed but appear as <poly>.

The OCamldebug program is supplied with OCaml. It operates only on compiled and linked bytecode executables, not on native code executables or on source. The program must have been compiled with debug information. In addition, the build process must make a bytecode executable by default, or in addition to a native code one. The stand-out feature of OCamldebug is its ability to "time-travel" – that is to jump backwards in a program's execution as well as forwards. This is achieved by the use of the unix fork mechanism. The intention is to make it easier to "catch the bug in the act".

The program is run in a sequence of numbered steps. A step is something like a function application or a conditional branch. One may: jump to a numbered step, forward or backward; print out the source code at the current step; inspect a value from the source code; and set breakpoints based on source code positions. As we shall see, there are some limitations. Let us take an example run. We start the debugger with the program cpdf -version:

```
$ ocamldebug cpdf -version
        OCaml Debugger

(ocd) run
Loading program... done.
cpdf Version 2.2
Time: 49260
Program exit.
```

We go to time zero, the beginning of the program. We have "time-travelled".

```
(ocd) go 0
Time: 0
Beginning of program.
```

We step forward one step at a time. What we see is just module initialisation from OCaml's built-in Standard Library Pervasives.

```
(ocd) step
Time: 1 - pc: 7384 - module Pervasives
26     (Invalid_argument "index out of bounds")<|a|>
(ocd) step
Time: 2 - pc: 7552 - module Pervasives
164    float_of_bits 0x7F_F0_00_00_00_00_00L<|a|>
```

We move into code from the actual program (rather than module initialization) but we are still stuck in

Standard Library code, there being no way to ask OCamldebug to show only steps involving the user's main program only.

```
(ocd) go 20000
Time: 20000 - pc: 136812 - module Arg
277     else <|b|>if s.[n] = ' ' then loop (n+1)
(ocd) step
Time: 20001 - pc: 136828 - module Arg
277     else if s.[n]<|a|> = ' ' then loop (n+1)
(ocd) step
Time: 20002 - pc: 136864 - module Arg
278     else <|b|>n
```

We print some values by giving their names:

```
(ocd) print n                                    ask for value of n
n: int = 1
(ocd) go 20001
Time: 20001 - pc: 136828 - module Arg
277     else if s.[n]<|a|> = ' ' then loop (n+1)
(ocd) print s                                    ask for value of s
s: string = " Display this list of options"
(ocd) print loop                                 ask for value of loop
Unbound identifier loop
```

Some values cannot be found, or are opaque. We cannot alter the values within the debugging environment and re-run the code.

OCamldebug can be used in conjunction with the Emacs text editor [38] to provide for a smoother debugging experience via shortcuts for debugger commands, and the ability to jump to the source code position of a breakpoint. It is also possible to install printers for user-defined data types, although the manual cautions *"For technical reasons, the debugger cannot call printing functions that reside in the program being debugged."*

It is possible to use a debugger which works on executables (such as GDB) with OCaml, of course, but facilities are limited. The semantic gap between the source text and the executable in the functional model of computation is much wider than when debugging a language such as C. There is, however, an extension [36] to GDB in development, which allows for limited printing out of OCaml values using the type annotation files left behind during compilation.

Haskell   In 2005, in a survey of the users of GHC, the most prominent Haskell compiler, *"By far the most common request was for a debugger"* [26]. We shall describe three such debuggers briefly, and then discuss the results of a similar survey undertaken ten years later, in 2015. The paper describing the current debugger (shipped with GHC), says *"The most prominent working debuggers for Haskell are Hat and Hood."* [26], so we choose those to look at before examining the GHC debugger.

The Hat [6] debugger operates by recompiling programs in such a way that they dump a trace of the whole execution to file as the program runs, whether it ends normally or with an error. After the program has finished, the user runs tools which use the dumped data to explore the execution of the program. A transformed program runs about a hundred times more slowly than the original. However, Hat allows some modules (say the standard library) to be "trusted" and therefore untraced. This also enables Hat users to debug programs which use third-party libraries which Hat has not, or cannot, recompile.

The tools provided include `hat-observe` to show the arguments with which each function is called, `hat-trail` to explore computations backwards (to answer the question "Where did my bug come from?"), and `hat-explore` to step through computations.

However, there are problems. The trace can be enormous, even for modest program runs. This, together with the tracing slowdown, may restrict the debugging of programs which do not fail (or otherwise end) quickly. Since Hat relies on transforming Haskell programs into ones which are semantically equivalent but which also output trace data, it cannot be used with programs which make use of language extensions Hat does not know about. Thus, one usage of a recent Haskell extension in a codebase can rule out Hat as a debugger. To debug inside libraries, Hat also requires one to recompile all the libraries in tracing variants, for use with Hat.

The Hood debugger [17] works by printing out data structures at various "observation points" in the program, rather than using the stepping model of the typical imperative debugger. As with Hat, part of the motivation for its design choices revolves around the extra complication of laziness – with Haskell's primitive `Debug.trace`, for example, the act of printing something out might change the evaluation order of the program, and therefore suppress a bug, or at least complicate reasoning. Hood allows the user to insert points at which observations about data structures are collected without altering the observable behaviour of the program. The authors give examples of this method fitting particularly well with the point-free style of functional programming, the observation point acting as a quasi-identity-function in the middle of a chain of functions. For example, `consumer . observe "intermediate" . producer` as the equivalent to `consumer . producer` but storing the debug information for this observation point under the label `intermediate`, from where it may be retrieved later. The Hood tool itself can be used for viewing such information.

The 2007 GHC debugger [26] was designed by looking at the flaws of Hat and Hood and trying to avoid them. In particular, the authors list ways in which Hat and Hood are not always available – for example, not able to be used on all programs, or being limited to one compiler, or requiring re-compilation of libraries, or not being able to be used interactively, or not being able to print polymorphic values. They go so far as to say *"The debugger should work with everything and always be available, even if this means sacrificing functionality"*. We have seen this same observation about 'availability' or 'accessibility' as the cornerstone of usability in our review of the literature in Section 1.1.

The debugger is used by loading the program into the REPL in the normal way, and using the extended REPL commands provided by the debugger (for example `:break`) to control debugging. Values of in-scope names may be inspected, and the program single-stepped.

An email survey [9] targeting 16000 Haskell users (with 1240 responses), commissioned by a commercial Haskell contractor, asked respondents to fill in the blank in the following sentence: *"Debugging and Profiling: improvements in this would be ____"*. The results were Crucial 29%, Important 30%, Helpful 23%, Slight help 9%, No impact 4%. Total Crucial or Important 59%. A free response field was also provided. The responses include *"Debugging Haskell code is like groping in the dark with a hand tied behind your back."*, *"Debugging Haskell is still a pain for beginners and hampers adoption."*, *"To be honest I'm a bit 'afraid' of this part of Haskell."*, and *"I would never be able to convince my coworkers* [to adopt Haskell] *without decent debugging support."* This last answer alludes to a source of past disillusionment about the apparent lack of progress of the art of programming despite vast improvements in computing power, language design, and compiler tools. As the field advances, old problems are solved only to be replaced with ones which could not have been conceived of unless we had already solved the

old ones. Debugging is likely always to be needed, and unlikely to be eliminated in the way envisaged by the early pioneers of computing.

Lisp   Common Lisp [39] has a tracing function similar to the OCaml one we looked at earlier, although there are more sophisticated facilities: the user can ask for certain values to be printed at each step, or for tracing to begin or end only when a certain predicate related to the code holds. The tracer is itself implemented as a LISP macro.

```
[1]> (TRACE rev)                                        trace our function
;; Tracing function REV.
(REV)x
[3]> (rev '(1 2 3 4))                                    invoke it
1. Trace: (REV '(1 2 3 4))
2. Trace: (REV '(2 3 4))
...
3. Trace: REV ==> (4 3)
2. Trace: REV ==> (4 3 2)
1. Trace: REV ==> (4 3 2 1)
(4 3 2 1)
```

In a similar fashion to the OCaml tracer, only the inputs and outputs are shown, rather than a diagram of the evaluation of the insides of the function.

Racket [14], a modern Scheme implementation, contains two debugging tools. The first is a breakpoint-based debugger with an optional graphical interface. Panes show the stack and the values of local names. When execution is paused at the start of an expression, an alternative value may be substituted for an expression, for experimentation purposes. Similarly, when execution is paused at the end of an expression's evaluation, an alternative return value may be substituted. The second is an algebraic stepper, which can show each step of the evaluation of the program in the source language:
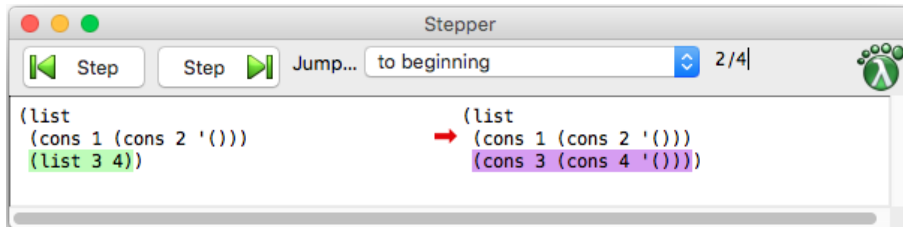
Both the debugger and the algebraic stepper use Racket's *continuation marks* scheme [7], which elaborates the program source such that, when it is compiled, enough information remains on the stack to point to, or even reconstruct the expression at the marked points. The debugger works for all Racket programs, but does not show the actual state of the expression being evaluated. The algebraic stepper does show the actual expression, but it only works for the small "Beginning student" and "Intermediate Student" languages, not the full Racket language. A recent paper by Cong and Asai [8] uses somewhat similar ideas to build an algebraic stepper for a small subset of Ocaml, for teaching purposes. One cannot debug into other libraries unless the libraries themselves have been compiled with such an elaboration. This may be mitigated somewhat by shipping an optional, elaborated version of the language's standard libraries. Such a stepping approach, nevertheless, appears to offer a compelling foundation for debugging functional programs, if it could be freed from its limitations.

Summary   We have explored some of the historical context of debugging as a persistent problem, and surveyed existing solutions in the functional world. Now we shall propose an approach of our own, which is rather surprising, but which we claim might form the foundation of a way of writing debuggers for functional languages which is free of many of the disadvantages of previous solutions.

2   Approach

We intend to tackle the problem of debugging by directly interpreting the program, allowing the intermediate steps of evaluation to be shown. When we say "directly interpreting", we mean just that – a completely naive step-by-step evaluation of the source code or AST without recourse to any kind of transformation or compilation whether involving a bytecode or not (we make this clarification because the REPL is often referred to as an interpreter though it does not work by interpretation.) Why interpretation? Because it allows the program to be run without the loss of information inherent in the compilation process. There is no reconstruction of information required, no lossy mapping back and forth between source and executable. It fits the model of functional programming as evaluation by reducing an expression to a value, rather than making the programmer think imperatively. This choice will, we hope, allow for a design which sweeps away many of the disadvantages of existing solutions, replacing them with one big disadvantage – that interpretation is extremely slow. We shall then work to mitigate that disadvantage to arrive at a usable debugger.

Before getting too deeply into this line of thought, let us begin with an example to illustrate the concept before returning to our argument, and to the specifics of our implementation.

2.1   An example

As is traditional, we consider a program for calculating the factorial of a positive number:

```
let rec factorial n =
  if n = 1 then 1 else n * factorial (n - 1)
in
  factorial 4
```

The upper portion of Figure 1 shows a naive visualization of the evaluation of this program. This is certainly not how we would write such an evaluation on paper. Although the evaluation shown is self-contained in the sense that each line of it is a valid program (which might seem a useful property) it is hard to see what is going on. It is large, both in width (how long the expression becomes) and length (how many lines are needed). Writing each evaluation step over multiple lines as we did with the original program above would not only increase the length, but make it difficult to visually compare adjacent lines. We must reduce the amount of information shown, even in this simple case.

Look now at the lower part of Figure 1, showing the output of our prototype system. We removed the definition of the `factorial` function itself (since it is recursive, its name will appear in the expression anyway.) We avoided printing any reduction step which leads to an expression such as **if** false or **if** true. We have not shown the intermediate steps of simple arithmetic which reduce 4 * (3 * (2 * 1)) to 24. We have removed trivial arithmetic (e.g. subtracting one), even when it involves variable names, such as reducing n - 1 to 3 directly rather than via 4 - 1. We have removed **let** bindings which apply to the

```
   let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in factorial 4
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in let n = 4 in if n = 1 then 1 else n * factorial (n - 1)
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in let n = 4 in if false then 1 else n * factorial (n - 1)
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in let n = 4 in n * factorial (n - 1)
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in let n = 4 in 4 * factorial (n - 1)
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * factorial (4 - 1)
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * factorial 3
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (let n = 3 in if n = 1 then 1 else n * factorial (n - 1))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (let n = 3 in if false then 1 else n * factorial (n - 1))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (let n = 3 in n * factorial (n - 1))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (let n = 3 in 3 * factorial (n - 1))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * factorial (3 - 1))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * factorial 2)
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * (let n = 2 in if n = 1 then 1 else n * factorial (n - 1)))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * (let n = 2 in if false then 1 else n * factorial (n - 1)))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * (let n = 2 in n * factorial (n - 1)))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * (let n = 2 in 2 * factorial (n - 1)))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * (2 * factorial (2 - 1)))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * (2 * factorial 1))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * (2 * (let n = 1 in if n = 1 then 1 else n * factorial (n - 1))))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * (2 * (let n = 1 in if true then 1 else n * factorial (n - 1))))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * (2 * 1))
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * (3 * 2)
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 4 * 6
=> let rec factorial n = if n = 1 then 1 else n * factorial (n - 1) in 24
=> 24
```

```
        factorial 4
n = 4  =>  if n = 1 then 1 else n * factorial (n - 1)
n = 4  =>  n * factorial (n - 1)
       =>  4 * factorial 3
n = 3  =>  4 * (if n = 1 then 1 else n * factorial (n - 1))
n = 3  =>  4 * (n * factorial (n - 1))
       =>  4 * (3 * factorial 2)
n = 2  =>  4 * (3 * (if n = 1 then 1 else n * factorial (n - 1)))
n = 2  =>  4 * (3 * (n * factorial (n - 1)))
       =>  4 * (3 * (2 * factorial 1))
n = 1  =>  4 * (3 * (2 * (if n = 1 then 1 else n * factorial (n - 1))))
       =>  4 * (3 * (2 * 1))
       =>* 24
```

Figure 1. A naive rendering of the evaluation of `factorial 4` showing each step of the evaluation, followed by an automatically reduced one, eliding a) parts of the evaluation of the **if** construct; b) the definition of a recursive function mentioned in the expression; c) the final portion of arithmetic; and d) trivial operations such as 3 - 1. In addition, **let** expressions unique in the whole expression are moved to the left, and basic syntax highlighting has been used. The expression to be reduced in each step has been underlined.

whole expression to the left hand side of the => arrow to avoid too many **let** n = ... instances making the output too wide. We have used simple syntax highlighting in the form of bold for keywords. Finally, we have underlined the expression to be reduced at each step. All changes have been made automatically. Each step is no longer a valid OCaml program, but the increase in readability is significant. Clearly, for larger programs, such elision will be even more important, since the focus needs to be on the currently-evaluating subexpression of a potentially huge expression representing the whole program. Note that all the intervening steps of the computation are performed, but certain lines are not printed. This means that the finer details of the computation may be inspected upon demand.

In the program trace we have already exhibited, it is clear that for realistic programs, the program trace (both its width and its length) may be significant. This issue is discussed in some detail by Taylor [41] and Pajera-Flores [30]. A practical solution, we claim, must involve providing ways of a) eliding information within a single step – reducing the width; b) eliding whole steps – reducing the length; c) searching the resultant trace, if it is still too large to spot the bug; and d) moving backward and forward through the trace to connect cause and effect in the computation.

2.2 Design choices

We choose to build a debugger which puts the notion of accessibility first (it is the core of usability), and everything else second. We claim that, without such universal availability, our debugger would be added to the growing pile of debuggers for functional languages which lie unused. And so this extremism is, in fact, in the service of practicality.

We can contrast the accessibility of a low-level debugger (such as GDB) which allows any executable to be debugged (assuming it was compiled with debug information enabled) to the much more limited accessibility of various high-level debuggers. We need to bridge this gap. So we should define precisely what trade-offs we are willing to accept. We can say that we will allow the following leeway in our extremism: 1) Just like GDB, we will insist upon the executable being compiled with a special flag. In fact, the requirement is rather stronger than GDB, since GDB is of some use on an executable not so compiled. 2) We shall not make any attempt to provide for the debugging of code from other languages (so, for example, C code linked into a primarily-OCaml executable will not be debuggable). But such executables will run properly. This means that, for example, if one suspects a bug exhibited in an OCaml executable is really a bug in the OCaml compiler or runtime, debugging it may require GDB in addition to our debugger. 3) There will doubtless be small ways in which the interpreter differs, whilst still being equivalent given the OCaml semantics. For example, its exact limits for stack overflows on non-tail-recursive code may be different, or the order of execution of threads may be different. This might mean that some bugs go away when switching to the interpreted version of OCaml. This might damage reproducibility when debugging certain kinds of low-level problems.

Notwithstanding these small things, we believe that this is the sensible approach for a high-level debugger. Let us list some of our principal aims here, to remind ourselves. We would like our eventual debugger (a) to be useable with any build system; (b) to work with mixed C/OCaml code; (c) to be able to debug libraries, not just the programmer's own code; (d) to be easy to keep in sync with the OCaml toolchain, so a new version can be released with each OCaml version; (e) to require minimal patches to the target toolchain; and (f) to be suitable for debugging the development of the OCaml compiler itself, and similar complex code.

A debugger which embodies these two kinds of requirement (negative and positive) ought to have the spirit of accessibility and so fulfil our needs.

2.3   Correctness and Maintainence

How will we know that the step-by-step interpreter is correct? Correctness is the major concern in any implementation of a programming language (which is why certified compilers have become such a hot topic), but we have a special extra concern. We must make sure that the interpreter matches the semantics of the native code OCaml compiler. It is no use trying to debug a program, only to find that the bug changes or disappears when using the interpreter. This concern exists also in the core OCaml distribution, unusually, since there are two compilers – bytecode and native code.

Our interpreter shares the front end with the bytecode and native code compilers, so we need not worry about correctness there – its behaviour should match exactly the bytecode and native code behaviour. In the actual evaluation, though, we must ensure the semantics of the language are obeyed to the same standard as in the compiled implementations. Formal proof will not work. There is no formal semantics of OCaml to prove adherence to and, in any event, without also formally proving the OCaml compiler, we cannot show the two are semantically equivalent. The OCaml distribution does come, however, with a large test suite, which we will use to test the interpreter. How do we know that it will continue to work when the OCaml compiler is updated? Languages change, and new language features are added. Of course, the best solution would be to eventually have our small-step interpreter included in the OCaml distribution itself. The reason for doing so would be social, not technical – it would ensure that for each release the interpreter would be updated along with the compilers. It is not unusual for debuggers to be included in the core distribution of a language. Failing such an inclusion, the debugger will have to be updated for each major release of OCaml. But these changes are likely to be rather easier than one might expect, due to the sharing of the front end. And some, such as the changes to typechecking internals in the front end which frequently appear in OCaml compiler change logs, will require no changes to the interpreter.

There is a marked difference in complexity between writing a simple big-step evaluator for an abstract syntax tree and its small-step counterpart. Recent work by Cong and Asai suggests a scheme whereby one may *"implement a stepper concisely by writing an evaluator that is close to a standard big-step interpreter"* [8], so there is reason to believe that this difficulty may also be overcome.

2.4   Functional program visualization

This paper is concerned principally with the overall shape of our approach and the technical details of OCamli, our interpreter for OCaml. However, we must also discuss the actual mechanisms for the visualisation of programs as shown in our factorial example. There is plenty of literature on this in the field of software visualization, as well as some earlier work in the field of functional languages. We take a brief review now (the following paragraphs appear in similar form in our earlier paper [48]).

Two useful surveys [45] [46] give a general overview of recent developments in this area, the first specific to functional programming, the second with wider scope. A very broad introduction [33] provides background. A comprehensive survey [37] of education systems for program visualization is useful too. We pick out a few recent systems for further discussion.

The *WinHIPE* system [30] is a recent incarnation of these ideas for the HOPE [4] language. It uses a step-by-step evaluation system, and explicitly addresses the problems of scale by elision of information and a focusing mechanism. The emphasis, however, is on graphical (tree-based) representations, an approach we shall not take, being of the belief that trees can often be, in fact, harder to read than well-pretty-printed program representations. The *Visual Miranda Machine* [1] provides a trace of the evaluation of a lazy functional program, together with a commentary showing the reason for choosing each evaluation step. There is a discussion of granularity, taking the example of the "list comprehension" language feature. *DrScheme* [15] provides, amongst many other facilities, an "algebraic stepper" for the Scheme language that can print out steps of evaluation. The stepper, however, supports only a subset of the language. The implementation is interesting, though – it reuses some of the underlying Scheme implementation to ensure equivalent semantics. Touretsky describes a LISP-based system [43] that produces mainly textual traces, but with some use of graphical elements to indicate the different scoping mechanisms peculiar to LISP. The presentation of *ZStep95* [44] begins by noting that debugging is, essentially, a human interface problem. The authors concentrate on the concept of *immediacy* (temporal, spatial, and so on), which they see as essential, and exhibit a stepping debugger for a functional language which can go back and forth through time. Another approach to this problem is as a special case of the more general concept of a *calculator* [34] [18], showing how it pertains to various logical systems with a mathematical basis, not just functional programs. *Prospero* [41] is a more fully-developed system, again for a lazy language. It includes methods for filtering the evaluation trace to elide information and a careful discussion of usability issues. A recent system for visualizing the evaluation of JavaScript, including its functional elements, is JSExplain [5]. It uses a reference interpreter derived from the language specification itself and so, in addition to its uses for teaching, it acts as a debugger for the specification of the JavaScript language itself.

These systems are mostly concerned with program visualization for teaching; we wish to bias ourselves towards the task of general debugging, hoping that some of the educational uses will be subsumed by it. The authors of DrScheme [15] urge caution here, choosing instead to build a "tower" of syntactically restrictive variants of Scheme specifically for educational purposes. They say that, due to the fact that so many sequences of characters are syntactically valid in Scheme, error messages are less confusing when the dialect is restricted – we would prefer to avoid this in the name of universality.

It is worth pointing out that much research in software visualization concerns overtly graphical approaches. We take a simpler line, sticking to pretty-printing. We claim that the most important aspect of a successful visualization is elision – reducing the information visible to just what is required so that large datasets may be understood easily. Programmers are used to seeing their program as text, and visualizing its evaluation as, for example, a graphical tree structure, is less useful for debugging large programs (it can be useful, of course, for visualizing program source code structure as opposed to evaluation traces).

3   OCamli: an interpreter for OCaml

This section presents the technical development and practical use of the prototype OCamli interpreter [29] for OCaml, considering both the method of step-by-step evaluation, and the heuristics used to make for concise, readable output. It does not support the whole language, but can load almost all of the OCaml Standard Library, and run its initialisation. It is the foundation upon which our eventual debugger will be built. Presently, it has a prototype command line interface which, on its own, is suitable for debugging some little OCaml programs.

We should like to have a command `ocamli` so we may write `ocamli test.ml` and the program will be interpreted, functioning in the same way as it would if compiled and then run. We will allow the flag `-show` to show the final result of the evaluation of the program:

```
$ ocamli -show test.ml
7
```

Here, the contents of `test.ml` is simply "1 + 2 * 3". The flag `-show-all` will show all the stages of computation:

```
$ ocamli -show-all test.ml
1 + 2 * 3
1 + 6
7
```

Such a program will need to read the source code, convert it to a representation suitable for direct interpretation, interpret it in a fashion which allows for the printing of each individual step, and print those steps out in a readable way.

3.1 A new representation for OCaml programs

It is possible to produce a step-by-step interpreter for OCaml which operates directly upon the parse tree data type exposed by the compiler-libs library (which is the OCaml toolchain's library form). However, the data type is not ideal. It holds much information which is not needed for interpretation, complicating pattern matching. At each step, we must then reconstruct such extra information to ensure that it is a valid parse tree again. A very early version of our interpreter was constructed using this method. The intent was to avoid introducing a new datatype (with maintenance issues), and to enable use of the existing OCaml prettyprinter. However, it quickly became apparent that the disadvantages outweighed the advantages. Consider, for example, the following code to add two integers, from a very early version of OCamli:

```
| Pexp_apply (expr, args) ->
    if List.for_all (fun (_, arg) -> is_value arg) args then          if all arguments are values
      begin match expr.pexp_desc with
      | Pexp_ident {txt = Longident.Lident (("*" | "/" | "+" | "-") as op)} ->      if an integer op
        begin match args with
          [(_, {pexp_desc = Pexp_constant (Const_int a)});
           (_, {pexp_desc = Pexp_constant (Const_int b)})] ->              extract integer values
           let result = calculate a b op in
             {e with pexp_desc = Pexp_constant (Const_int result)}          rebuild parse tree node
        | _ -> malformed __LOC__
        end
      end
    else
      (cases where one or more arguments not yet values)
```

We have to check each item in the list of things to be applied is a value, match against strings representing operators, and deal with many nested records, taking them apart and building them back up once we have evaluated a single step. Instead, we should like to be able to just write:

```
| Op (op, Int x, Int y) -> Int (calculate op x y)
```

This is the aim of our new representation for OCaml programs (we will not call it a representation of OCaml parse trees or Abstract Syntax Trees, because the information it must contain is different.) We call it TinyOCaml. Here is part of the main type (we skip most of it for brevity, together with some of the types to which it refers). Unlike the OCaml parse tree datatype, which is a complicated set of mutually-recursive datatype definitions, we have only a few here.

```
type t =
  Unit                                                    atomic types
| Int of int
| Bool of bool
| Float of float
| Record of (string * t ref) list                            record
| Tuple of t list                                             tuple
| Cons of (t * t)                                              list
| Nil
| Array of t array                                            array
| Constr of int * string * t option        user-defined data type constructor
| Fun of (label * pattern * t * env)                        function
| Function of (case list * env)              function with pattern-match
| Var of string                                            variable
| Op of (op * t * t)                                    binary operator
| Cmp of (cmp * t * t)                             comparison operator
| If of (t * t * t option)                          conditional statement
| Let of (bool * binding list * t)                       let-binding
| LetDef of (bool * binding list)             let-binding structure item
| TypeDef of (bool * Parsetree.type_declaration list)  user-defined type definition
| App of (t * t)                                     function application
| Seq of (t * t)                                    imperative ; operator
| While of (t * t * t * t)                                 while loop
| For of (string * t * forkind * t * t * t)                 for loop
| Raise of (string * t option)                          raise exception
| Match of (t * case list)                              pattern match
| TryWith of (t * case list)                           try...with block
| ExceptionDef of (string * Parsetree.constructor_arguments)  exception definition
| CallBuiltIn of                                        built-in primitive
    (typ option * string * t list * (env -> t list -> t))
| Struct of (bool * t list)                        module implementation
| Sig of t list                                       module signature
```

Let us look again at an example program, and see its evaluation as it may be printed on screen by OCamli:

```
   1 + 2 > 3 + 4
=> 1 + 2 > 7
=> 3 > 7
=> false
```

And here is what is going on inside OCamli – much simpler than directly manipulating the OCaml parsetree itself:

```
   Cmp (GT, Op (Add, Int 1, Int 2), Op (Add, Int 3, Int 4))
=> Cmp (GT, Op (Add, Int 1, Int 2), Int 7)
=> Cmp (GT, Int 3, Int 7)
=> Bool false
```

We shall now consider how to convert the OCaml parse tree into our new type for OCaml programs.

3.2   Conversion to and from real OCaml programs

Consider the following extract of the `of_real_ocaml` reader for converting an OCaml parse tree into the TinyOCaml representation:

```
| Pexp_construct ({txt = Lident "[]"}, _) -> Nil
| Pexp_construct ({txt = Lident "::"}, Some ({pexp_desc = Pexp_tuple [e; e']})) ->
    Cons (of_real_ocaml env e, of_real_ocaml env e')
```

This deals with the standard OCaml list syntax. Similar code deals with each other part of the OCaml syntax. Closure conversion is done at the same time (this is the env argument above), since it is convenient and avoids another pass.

Converting the other way (from TinyOCaml to OCaml's parse tree type) can be useful too, for example if we wish to use the built-in OCaml prettyprinter:

```
| Unit -> Pexp_construct ({txt = Longident.Lident "()"; loc = Location.none}, None)
| Int i -> Pexp_constant (Pconst_integer (string_of_int i, None))
| String s -> Pexp_constant (Pconst_string (s, None))
| Bool b ->
    Pexp_construct
      ({txt = Longident.Lident (string_of_bool b); loc = Location.none},
        None)
```

Note again the stark difference in verbosity between our type Tinyocaml.t (to the left of each arrow) and OCaml's parse tree type (to the right of each arrow).

3.3   Evaluating expressions

For this proof-of-concept, a very simple interpreter has been produced. It has no pretensions towards performance, either by preserving space and time efficiency vis-a-vis the same program compiled and executed, or with regard to constant overheads. Its job is to provide a minimal working example for experimentation (remember our mantra: accessibility first, everything else, including speed, second).

Evaluation strategy   To evaluate a step of a program (that is, something of type Tinyocaml.t), we must first determine if the program is a value. If it is, there is no evaluation to be done. If not, we find the reducible expression, following the OCaml order of evaluation (to the extent that OCaml specifies the order). We perform one step of evaluation only. This new expression may now be returned for printing, and we continue with the next step.

Let us look at a simple example, comparing with a traditional evaluator (whose job is to evaluate down to a value in one continuous operation). We choose the short-circuiting boolean conjunction operator &&. Here is a snippet from an all-at-once interpreter:

```
let rec eval = function
  | And (a, b) ->
      match eval a with
      | Bool false -> Bool false
      | Bool true -> eval b
```

We evaluate the left hand side `a` to a Tinyocaml.t representing a boolean (either `Bool true` or `Bool false`). If it is `Bool false`, this is the result. If it is `Bool true`, the code evaluates the right hand side to a value, and returns it. Contrast with the following, which evaluates just a single step:

```
let rec eval_step = function
  | And (Bool false, _) -> Bool false
  | And (Bool true, Bool b) -> Bool b
  | And (Bool true, b) -> eval_step b
  | And (a, b) -> And (eval_step a, b)
```

The first line of the pattern match in the step-by-step example is used when the left hand side has already been fully evaluated and is false: this is the short circuit. The second deals with a fully-evaluated true left hand side, and a fully evaluated right hand side. The third is the same as the second, but for a right hand side not yet fully evaluated: we have found the step which requires evaluation. The fourth and last is for an unevaluated left hand side: we evaluate the left hand side one step and leave the right hand side alone. By similar mechanisms it is possible to write a step-by-step evaluator for each other part of the language. We consider some of the more interesting ones now by way of further example.

Imperative programs Whilst OCaml is a functional language first, there is occasional use of imperative features, and we need to display them in a way which fits in. Consider the evaluation of the OCaml **for** construct. When compiled, the following piece of code will print `12345`:

```
for y = 0 + 1 to 6 - 1 do print_int y done
```

But how do we show it? It can be treated as an expression:

```
    for y = 0 + 1 to 6 - 1 do print_int y done
=> for y = 1 to 6 - 1 do print_int y done
=> for y = 1 to 5 do print_int y done
1=> for y = 2 to 5 do print_int y done
2=> for y = 3 to 5 do print_int y done
3=> for y = 4 to 5 do print_int y done
4=> for y = 5 to 5 do print_int y done
5=> for y = 6 to 5 do print_int y done
=> ()
```

Helpfully, the semantics of OCaml are such that **for** `y = 6` **to** `5` **do** `...` **done** is legal and does not execute the body, so we have a proper terminating condition. How is this implemented? The `For` constructor of the Tinyocaml.t data type looks like this:

```
For of string * t * forkind * t * t * t
```

Our example would be represented like this:

```
  For ("y",
       Op (Add, Int 0, Int 1),
       UpTo,
       Op (Sub, Int 6, Int 1),
       App (Var "print_int", Var "y"),
       App (Var "print_int", Var "y"))
```

We need two copies of the body, so that one may be evaluated step-by-step, and then, when it has been reduced to a value, the spare copy can be moved into place, and we go round again. Here are all the cases needed for step-by-step evaluation of the **for** construct:

```
| For (v, e, ud, e', e'', copy) when not (is_value e) ->                          evaluate from part
    For (v, eval env e, ud, e', e'', copy)
| For (v, e, ud, e', e'', copy) when not (is_value e') ->                          evaluate to part
    For (v, e, ud, eval env e', e'', copy)
| For (_, Int x, UpTo, Int y, _, _) when x > y -> Unit                                end condition
| For (_, Int x, DownTo, Int y, _, _) when y > x -> Unit                              end condition
| For (v, Int x, ud, e', e'', copy) when is_value e'' ->         advance the for loop using the copy
    For (v, Int (x + 1), ud, e', copy, copy)
| For (v, x, ud, e', e'', copy) ->                                              evaluate the body
    For (v, x, ud, e', eval (EnvBinding (false, ref [(PatVar v, x)])::env) e'', copy)
```

Note the final case, where the variable is bound for the evaluation. The treatment of **while** is similar. Now consider how to deal with another imperative construct: the reference. A reference in OCaml is a mutable cell containing a value. Here is a simple imperative program using a reference:

```
    let x = ref 0 in x := !x + 1
=>  let x = {contents = 0} in x := !x + 1
=>  let x = {contents = 0} in x := 0 + 1
=>  let x = {contents = 0} in x := 1
=>  let x = {contents = 1} in ()
=>  ()
```

Note that, even though the new value of the reference is lost in the final expression (), it is visible in the penultimate step, which is good enough. OCamli can emulate the low-level primitives used to implement some of OCaml's basic language features. When we opt to show the low-level primitives involved in the use of references, we see a somewhat longer version:

```
    let x = ref 0 in x := (!x + 1)
=>  let x = let x = 0 in <<%makemutable x>> in x := (!x + 1)
=>  let x = {contents = 0} in x := (!x + 1)
=>  let x = {contents = 0} in {contents = 0} := (!x + 1)
=>  let x = {contents = 0} in (let x = {contents = 0} in fun y -> <<%setfield0 x y>>) (!x + 1)
=>  (fun y -> let x = {contents = 0} in <<%setfield0 x y>>) (!{contents = 0} + 1)
=>  (fun y -> let x = {contents = 0} in <<%setfield0 x y>>)
      ((let x = {contents = 0} in <<%field0 x>>) + 1)
=>  (fun y -> let x = {contents = 0} in <<%setfield0 x y>>) (0 + 1)
=>  (fun y -> let x = {contents = 0} in <<%setfield0 x y>>) 1
=>  let y = 1 in let x = {contents = 0} in <<%setfield0 x y>>
=>  ()
```

Most users will not want this longer output by default, but it is helpful when we wish to see, for example, exactly what I/O calls are triggered by Standard Library functions.

Currying   When we teach functional programming we often say "every function only has one argument" but really, except in cases of partial application, programmers think of curried functions as a single function of multiple arguments. And how the programmer thinks is how the debugger must behave. Consider the default evaluation of (**fun** x y -> x + y) 4 5:

```
    (fun x y -> x + y) 4 5
=>  (let x = 4 in fun y -> x + y) 5
=>  (fun y -> let x = 4 in x + y) 5
=>  let y = 5 in let x = 4 in x + y
=>  let y = 5 in 4 + y
=>  4 + 5
=>  9
```

We always print **fun** x y -> instead of **fun** x -> **fun** y ->, since they are indistinguishable in the OCaml parse tree. There are a small number of places where semantically equivalent syntactic forms are not distinguished like this, and we would want eventually to modify the OCaml parser to retain information about the original form.

Returning to currying, the evaluation above is excessively verbose. When the -fast-curry option is added to the command line, the arguments will be applied at once:

```
    (fun x y -> x + y) 4 5
=>  let x = 4 in let y = 5 in x + y
=>  let y = 5 in 4 + y
=>  4 + 5
=>  9
```

This involves a more complicated matching on the program to identify all the arguments which can be applied. It is an example of the requirements of the visualization driving the implementation of the interpreter's evaluation model. In fact, combined with another option -side-lets (which pulls out let-bindings to the side), we get an evaluation which is better still:

```
        (fun x y -> x + y) 4 5
x = 4 y = 5 =>  x + y
      y = 5 =>  4 + y
            =>  4 + 5
            =>  9
```

In the future, we could go further, and do away with the step-by-step lookup of variables, imagining the optimal visualisation:

```
        (fun x y -> x + y) 4 5
x = 4 y = 5 => x + y
            => 4 + 5
            => 9
```

or even:

```
    (fun x y -> x + y) 4 5
=>  4 + 5
=>  9
```

This is perhaps what we might write if we were to do this on paper – when we write such evaluations informally we naturally skip "obvious" steps. It is the same when doing mathematics. We can see that most of the job of improving upon the naive visualization consists of removing information, rather than adding it.

Exceptions   As one would expect, exceptions (which interrupt the flow of evaluation) require a similar mechanism inside an interpreter. The complication of a step-by-step interpreter is that exceptions must be modelled in a step-by-step way too: we cannot let uncaught raises cascade all at once. The solution to this is to model exceptions in two ways: as a special Tinyocaml.t constructor Raise and using actual exceptions. Here is the exception definition we will use, which represents, for example, the result of evaluating **raise** (Failure "broken") as ExceptionRaised ("Failure", Some (String "broken")):

```
exception ExceptionRaised of string * Tinyocaml.t option
```

Here is the Tinyocaml.t constructor used to represent exceptions which need to be raised:

```
| Raise of string * Tinyocaml.t option
```

See how it mirrors the exception definition above. Now, let us consider the case of dividing two numbers, where the second may be zero. Here is the code from the interpreter:

```
| Op (op, Int a, Int b) ->
    begin try Int (calc op a b) with
      Division_by_zero -> Raise ("Division_by_zero", None)
    end
```

We use OCaml exception handling to check for `Division_by_zero` in the `calc` function, and if we see it, we build the `Raise` constructor as the result of evaluating this expression one step. This freezes the exception. What happens when, in the next step of evaluation, this `Raise` is found?

```
| Raise (e, payload) ->                        payload is the data carried with the exception
    match payload with
    | Some x when not (is_value x) ->
        Raise (e, Some (eval_step env x))              if payload not a value, evaluate one step
    | _ ->
      raise (ExceptionRaised (e, payload))                otherwise, the exception may be raised
```

We may need to evaluate the expression in the `Raise` one step if it is not a value (it might be **raise** (Fail (1 + 2)), for example). Thus, the `Raise` may take several steps to be processed. If it is a value, though, we can raise the actual exception. This will be caught in the evaluator, and mirrors the effect of the exception occurring in a compiled program. Here is code for the **try ... with** construct:

```
| TryWith (e, cases) ->
    if is_value e then e else                                     if body a value, return
      begin try TryWith (eval_step env e, cases) with            evaluate body one step
        ExceptionRaised (x, payload) ->                     if this step caused an exception
          match eval_match_exception env x payload cases with        see if it matches a case
          | FailedToMatch -> Raise (x, payload)                 if not, recreate the raise node
          | Matched e' -> e'                          otherwise, return the body of the matched case
      end
```

If the exception is not surrounded by a **try ... with**, it is not caught, and so is printed at the top level and the interpreter exits:

```
    1 + 1 / (1 - 1)
=>  1 + 1 / 0
=>  1 + raise Division_by_zero
Exception: Division_by_zero.
```

Let us add a **try ... with**:

```
    try 1 + 1 / (1 - 1) with Division_by_zero -> 2 + 2
=>  try 1 + 1 / 0 with Division_by_zero -> 2 + 2
=>  try 1 + raise Division_by_zero with Division_by_zero -> 2 + 2
=>  2 + 2
=>  4
```

Now we can see the whole process. As an improvement, we might like to annotate the penultimate step to indicate which expression matched.

Pattern matching How can we visualize pattern matching, one of the most widely-used, and praised, features of functional programming? Do we show the whole pattern, then jump to the right hand side of the chosen match? Do we show how the match matches? Consider an example:

```
   match 1 + 2 with 4 -> 0 | 3 -> 1 + 2 | _ -> 1
=> match 3 with 4 -> 0 | 3 -> 1 + 2 | _ -> 1
=> match 3 with 3 -> 1 + 2 | _ -> 1
=> 1 + 2
=> 3
```

In this method of visualisation, we simply show the whole match expression with all its cases, and each time a case does not match, we drop it from the front. As an option, in the future, we will allow the skipping of this process, and show just the case that matched. Functional programmers are good at spotting which case will be taken, and identifying when an unexpected one has been taken, signifying a bug.

Summary We now have a function which, given a program, can evaluate it one step. By calling the function repeatedly, feeding its own output back in as the next input, the program can be evaluated completely, step-by-step. With an appropriate pretty-printer, each step may be printed out. Sometimes the requirements of good visualization force changes to the evaluation method itself, changing the number or kind of steps.

3.4 Dealing with size by elision

This section concerns the important task of making the output readable (we discuss searching, which also reduces the output, in Section 3.6). So, what remains? Three things: 1) the showing or eliding of whole steps for things like simple arithmetic and variable lookups; 2) the hiding or showing of parts of the expression at each step; and 3) the default heuristics for eliding parts of individual expressions (for example, the internals of built-in functions).

For our first example, we shall consider how to automatically abridge the following arithmetic evaluation, of a type which frequently occurs at the end of a non-tail-recursive function application:

```
   1 * (2 * (3 * 4))
=> 1 * (2 * 12)
=> 1 * 24
=> 24
```

We wish to remove the middle two steps, leaving just:

```
   1 * (2 * (3 * 4))
=> 24
```

This can be done by a mechanism we call peeking.

Peeking In order to decide whether to show the current state, it is sometimes important to know the next state, and to remember the previous state. But how can we know the next state without evaluating it? One way, of course, would be to evaluate the whole program and print out its steps of execution offline. But we may wish to stop evaluation based on what is about to happen, and we cannot do this with a real running program with side effects, since we cannot roll back a side effect such as a network communication with a third party.

The solution is to add to the step-by-step evaluator the notion of *peeking*. In this mode, the evaluator identifies the reducible expression, but does not evaluate it. The calling function can then interrogate the interpreter to ask "What kind of operation would have been performed?". Presently, the answer is one of a short list, giving just enough information to provide for some of the elisions the OCamli prototype can perform:

```
type last_op =
    Arith                    simple arithmetic
  | Boolean                  &&, ||
  | Comparison               comparison operators
  | IfBool                   if true, if false
  | InsideBuiltIn            evaluation inside an external piece of code
  | VarLookup                variable lookup
```

In our example, we print the step if and only if a) the next state is a value or b) the current state is a value or c) Arith is not present for the previous state or d) Arith is not present for the next state. These four conditions, taken together, elide just enough steps of the arithmetic, but do not remove information we want to see. Similar conditions have been devised for the other kinds of elision listed in the last_op type.

Eliding within a step   Consider the following example with multiple structure items (a structure item in the parlance of the OCaml parse tree is a type definition or a top level let-binding):

```
let x = 1 + 2

let y = x + x

let z = 1 + y
```

First, of course, we begin by evaluating 1 + 2, and proceed from there. However, a lot of screen space is used by printing out these five lines (three code, two blank) for each step, and it can be hard for the user to follow along. Should we remove a structure item when it is no longer needed, assuming that the user is interested only in the final result of z? This results in a shorter but arguably incomplete trace. Or, instead, only show the structure item which is currently being evaluated? Most likely, this would be a configurable option with a sensible default, which is probably to reduce the trace as much as possible.

3.5   The Standard Library

OCaml comes with a small but useful library of routines. These fall broadly into three categories: (a) those which are simply there to provide a selection of common routines, useful for many programs, but which the user could write themselves – entirely in OCaml – if they wanted. For example, List.map; (b) those which are in the Standard Library because they are used in the implementation of the OCaml toolchain, but seemed to the authors to be generic enough as to be useful for the general programmer (when a programming language is in its infancy, the general programmer and the compiler author are one and the same); and (c) those which must be in the Standard Library because they provide facilities which pure OCaml programs could not provide, or use an external symbol, or talk to the runtime.

Categories (a) and (b) are easy to deal with – we are just interpreting standard OCaml code, so it is as if the user had themselves supplied the code. The OCamli interpreter knows how to load multiple modules as libraries using command line arguments. For example, the following command line loads modules A

and B, performing any module initialisation code, then executes the code given in the `-e` argument in an environment in which such modules exist:

```
$ ocamli a.ml b.ml -e 'let () = B.calc 10'
```

It is category (c) which requires special treatment. Functions which are external to OCaml are introduced like this:

```
external word_size : unit -> int = "%word_size"
```

This name might be exported directly or might be used in the definition of a Standard Library function which is then exported. In the example above, it indicates that a function of type **unit** → **int** is expected to be available at link time under the symbol %word_size and that it is to be given the name word_size. When we come across such an **external** declaration in a .ml file (such as when loading the Standard Library), how should OCamli deal with it? What we do is to write (or generate) a binding for it. The Tinyocaml.t datatype already exhibited contains the constructor CallBuiltIn:

```
CallBuiltIn of ... * (Tinyocaml.env -> Tinyocaml.t list -> Tinyocaml.t)
```

This inclusion of a native OCaml function into the Tinyocaml.t data type for programs is the mechanism by which the gap between the interpreted and native worlds is bridged. It represents an OCaml function which takes an environment and a list of Tinyocaml.t arguments, calls some external native function and returns a Tinyocaml.t result.

We can use this CallBuiltIn mechanism to build an interface to our function, and a way to look up such an interface by name so that, at runtime, it may be located and called by the interpreter:

```
external word_size : unit -> int = "%word_size"

let percent_word_size =
  let f =
    (function [Unit] ->
       begin try Int (word_size ()) with e -> exception_from_ocaml e end
     | _ -> failwith "%word_size")
  in
    ("%word_size",
     Fun (NoLabel, PatVar "*x", CallBuiltIn (None, "%word_size", [Var "*x"], f), []))
```

Notice the **external** declaration is retained. We then create an entry to look up this function in the table of primitives ("%word_size", x) where x is a function containing a CallBuiltIn. This table will be used for lookup when an **external** declaration is found in a .ml source file being interpreted. In this case, it is a function of one argument *x (the asterisk is a crude mechanism to mark such functions so they are not printed, since they are not part of the original source code). This function can then be applied to an argument in the interpreted world. The argument will be assigned the name *x and used by the native function – the result will be returned to the interpreted world. Now we need to look at the function f itself. It pattern-matches on the input argument list, requiring just one argument [Unit]. It tries to produce the output Int (word_size ()) by applying the native function word_size as defined by the **external**. This is the result. Should an exception be raised during the execution of word_size (either in OCaml code or C code) it comes into the OCaml runtime as an OCaml exception, and is then converted into a Tinyocaml.t representation of an exception by the function exception_from_ocaml. Curried functions may be defined using a helper for each arity. For example, for arity three:

```
let mk3 name f =
  (name,
   Fun (NoLabel, PatVar "*x",
     Fun (NoLabel, PatVar "*y",
       Fun (NoLabel, PatVar "*z",
             CallBuiltIn (None, name, [Var "*x"; Var "*y"; Var "*z"], f), []), []), []))
```

A function defined by this method may be partially applied as usual: only when all the arguments are actually applied in the interpreter will the native function f be run. To avoid writing all these bindings for the Standard Library by hand, a system has been developed which allows one to write, instead:

```
[%%auto external string_of_float : float -> string = "%string_of_float"]
```

The binding is then generated automatically. This system, works for most of the Standard Library functions, and so reduces OCamli's Standard Library file to a third of its previous size. Thus, we keep the part of OCamli which may need updating when OCaml is updated as small as possible. In Section 3.7, we shall see such a system might be expanded to allow an interface between compiled and interpreted code, and identify its limitations.

3.6   Searching

We have discussed various mechanisms for making sure that OCamli's output is reasonable in the default case, and that there are options for deciding what information to display. But we will want a proper searching mechanism too, especially for interactive scenarios. Of course, one way is to use standard command line tools like grep. How well would that work for a typical search on a typical program? We can foresee problems – for example, patterns may need to match independent of parenthesisation. In essence, we are searching the text not the program's syntactic structure.

The problem of searching in program code, either in textual or AST form, is known in the literature. Paul and Prakash's SCRUPLE system [32] uses an extended form of the programming language's own grammar, an approach from which we shall draw inspiration. Devanbu's GENOA [11] also reuses the language's parser in the context of source code analysis. Crew's ASTLOG [10] has similar aims. The distinction between "lexical matchers" (such as regular expressions) and "syntactic matchers" (which know the syntactic structure of what they are searching) is explored in Griswold et al's TAWK system [20].

If we are to provide tools of our own, what facilities might be useful? Here are the basic options provided in OCamli:

`-search`	*show only matching evaluation steps*
`-highlight`	*highlight the matching part of each matched step*
`-no-parens`	*ignore parentheses when matching*
`-regexp`	*search terms are regular expressions rather than the built-in system*
`-upto <n>`	*show the* n *lines preceding each result line*

For example, consider searching only for lines containing '4::':

```
=>  2::3::let l = [] in let f x = x + 1 in 4::map f l
=>  2::3::let l = [] in 4::map (fun x -> x + 1) l
=>  2::3::let l = [] in
       4::(let f x = x + 1 in function [] -> [] | a::l -> let r = f a in r::map f l) l
```

```
=>  2::3::let l = [] in
      4::(function [] -> [] | a::l -> let f x = x + 1 in let r = f a in r::map f l) l
=>  2::3::let l = [] in
      4::(function [] -> [] | a::l -> let f x = x + 1 in let r = f a in r::map f l ) l
=>  2::3::4::(function [] -> [] | a::l -> let f x = x + 1 in let r = f a in r::map f l ) []
```

This shows only the evaluation steps containing the text "4::", that is the ones where the list has almost been processed. Our search syntax is tailored to the job of searching OCamli's output. The search pattern is parsed using OCaml's lexer, and then we allow any amount of whitespace between tokens, skip parentheses (if -no-parens is set), and allow the underscore character _ to stand for any token. A regular expression is generated to represent this, and searching proceeds. For example, we can search with -search '[_; _; _]' for only those steps of evaluation which contain lists of length exactly three:

```
    List.map (fun x -> x + 1) [1; 2; 3]
=>  (let f x = x + 1 in function [] -> [] | a::l -> let r = f a in r::map f l ) [1; 2; 3]
=>  (function [] -> [] | a::l -> let f x = x + 1 in let r = f a in r::map f l ) [1; 2; 3]
=>  (function a::l -> let f x = x + 1 in let r = f a in r::map f l ) [1; 2; 3]
=>  [2; 3; 4]
```

The searches may be highlighted with -highlight:

```
    List.map (fun x -> x + 1) [1; 2; 3]
=>  (let f x = x + 1 in function [] -> [] | a::l -> let r = f a in r::map f l ) [1; 2; 3]
=>  (function [] -> [] | a::l -> let f x = x + 1 in let r = f a in r::map f l ) [1; 2; 3]
=>  (function a::l -> let f x = x + 1 in let r = f a in r::map f l ) [1; 2; 3]
=>  [2; 3; 4]
```

There are also options to alter the type and number of results:

-invert-search	*invert the search, showing non-matching steps*
-n	*show only n results*
-until	*show only until this matches a printed step*
-after	*show only after this matches a printed step*
-until-any	*show only until this matches any step*
-after-any	*show only after this matches any step*
-invert-after	*invert the after condition*
-invert-until	*invert the until condition*
-stop	*stop computation after final search results*
-repeat	*allow the after ... until result to be repeated*

These options allow the programmer to show output only after a search matches, and only until another search matches. For example with -after '3 + 1' -until '2::3::4':

```
=>  2::3::let l = [] in let f x = x + 1 in let r = 3 + 1 in r::map f l
=>  2::3::let l = [] in let f x = x + 1 in let r = 4 in r::map f l
=>  2::3::let l = [] in let f x = x + 1 in 4::map f l
=>  2::3::let l = [] in 4::map (fun x -> x + 1) l
=>  2::3::let l = [] in
      4::(let f x = x + 1 in function [] -> [] | a::l -> let r = f a in r::map f l) l
=>  2::3::let l = [] in
      4::(function [] -> [] | a::l -> let f x = x + 1 in let r = f a in r::map f l) l
=>  2::3::let l = [] in
=>  2::3::4::
      (function [] -> [] | a::l -> let f x = x + 1 in let r = f a in r::map f l ) []
```

These searching mechanisms were arrived at through conjecture about and exploration of the most likely useful tools. It remains to be seen what the best interface for our interpreter or debugger will be. We discuss one promising option now – a mechanism for annotating source code to be interpreted, leaving the rest of the program running natively at full speed.

3.7   An interface for debugging

In this section, we propose an unimplemented interface for debugging which uses the interpreter in a novel way, inserting it into a normal compiled OCaml program. In Section 3.6, we mentioned a system for expanding **external** definitions into shims for calling into C code:

```
[%%auto external string_of_float : float -> string = "%string_of_float"]
```

These [%%<name> <contents>] annotations are in fact a standard feature of recent versions of OCaml, called PPX annotations. They may be attached to any parse tree node. When OCaml compiles a file, it resolves such annotations by calling external processes (PPX extensions) which replace them with a parse tree. Once all annotations are resolved, the source is typechecked and compiled as usual. And so, we may use this mechanism to allow an OCaml program to be compiled as usual, except that the part we wish to debug (and so be interpreted) is marked with an [@interpret] annotation. This time, instead of communicating between OCaml and C code, we are communicating between the interpreted OCaml code and compiled OCaml code.

The use of [@interpret] annotations to control which parts of the code are executed natively and which parts are interpreted (and so have their steps of evaluation displayed on the screen) was motivated by our observation that a tool like OCamli on its own would not fulfil our usability needs, in particular our requirement that we must get it "inside" the build process. The improvement in speed (by interpreting only what we need to debug) is a side-effect, but a pleasant one. Is another such side-effect of this mechanism a natural and pliable human interface for debugging? If it is, we may be within sight of achieving our original aim of a usable debugger. The interface would be:

1. Notice that a misbehaviour is occurring.

2. Knowing or speculating upon the location of the root cause, insert one or more appropriate [@interpret] annotations in the code.

3. Recompile and run the code. The evaluation of the parts chosen will be shown.

4. If the source or nature of the bug is now clear:

   (a) Change the source to fix the bug.

   (b) Build and run again and inspect the output to be sure it is fixed.

   (c) Remove the [@interpret] annotation(s).

5. If the source or nature of the bug is not yet clear, due to a wrong or insufficient choice of [@interpret] annotations, return to step 2.

Let us explore the design space of such debugging annotations. Consider the following possibilities, for instance:

[@interpret] The piece of code annotated is interpreted, but functions it calls into are not. For example, consider the following buggy function on lists:

```
let rec pairs f a l =
  match l with
    [] -> rev a
  | [_] -> []
  | h::h'::t -> pairs f (f h h'::a) t

let x = pairs ( + ) [] [1; 2; 3; 4]
```

It is supposed to take, for example [1; 2; 3; 4] to [1 + 2; 2 + 3; 3 + 4] if the function f is addition. The input a is an accumulator to make it tail-recursive. There are two bugs in this version above. Firstly, for the case of the single-item list, the result should be rev a, as for the empty list. Secondly, the last case should read pairs f (f h h'::a) (h'::t).

We can add an [@interpret] annotation on to the outer invocation of pairs:

```
let rec pairs f a l =
  match l with
    [] -> rev a
  | [_] -> []
  | h::h'::t -> pairs f (f h h'::a) t

let x = [@interpret] pairs ( + ) [] [1; 2; 3; 4]
```

Now, upon compiling the program the interpreter is embedded, and all the code inside pairs is interpreted, but external calls (for example to rev) are native, and so elided from the output. We think this is the sensible default, both for elision of information and elision of computation. The output upon running the program would be the following:

```
   pairs ( + ) [] [1; 2; 3; 4]
{matches h::h'::t}
=> pairs ( + ) [3] [3; 4]
{matches h::h'::t}
=> pairs ( + ) [7; 3] []
{matches []}
=> [3; 7]
```

The first three lines are generated from the pairs function call itself, the last line from the returned value. Note that the default elision also does not show the details of the matches in as much detail as the output of OCamli we showed in Section 2.1. The bug is plain to see, so we correct it:

```
let rec pairs f a l =
  match l with
    [] -> rev a
  | [_] -> []
  | h::h'::t -> pairs f (f h h'::a) (h'::t)

let x = [@interpret] pairs ( + ) [] [1; 2; 3; 4]
```

We compile the code again, with the annotation in the same place, and try again:

```
   pairs ( + ) [] [1; 2; 3; 4]
{matches h::h'::t}
=> pairs ( + ) [3] [2; 3; 4]
{matches h::h'::t}
```

```
=> pairs ( + ) [5; 3] [3; 4]
{matches h::h'::t}
=> pairs ( + ) [7; 5; 3] [4]
{matches [_]}
=> []
```

Still there is a bug. Since the accumulator looks close to correct during evaluation, only to see the output disappear at the last moment, we deduce it must be the match case [_] which is wrong, and we correct it:

```
let rec pairs f a l =
  match l with
    [] | [_] -> rev a
  | h::h'::t -> pairs f (f h h'::a) (h'::t)

let x = [@interpret] pairs ( + ) [] [1; 2; 3; 4]
```

Here is the final, correct output:

```
    pairs ( + ) [] [1; 2; 3; 4]
{matches h::h'::t}
=> pairs ( + ) [3] [2; 3; 4]
{matches h::h'::t}
=> pairs ( + ) [5; 3] [3; 4]
{matches h::h'::t}
=> pairs ( + ) [7; 5; 3] [4]
{matches [_]}
=> [3; 5; 7]
```

Now, we can remove our annotation:

```
let rec pairs f a l =
  match l with
    [] | [_] -> rev a
  | h::h'::t -> pairs f (f h h'::a) (h'::t)

let x = pairs ( + ) [] [1; 2; 3; 4]
```

Our debugging is complete.

[@interpret-all] The piece of code annotated is interpreted, and so is every function it calls, if available (the default action of [@interpret] is that calls into other modules are native).

[@interpret-logto] The output is not written to standard output or standard error, but appended to a file. This can be used to separate the output of several annotations, or several runs of the same program, or as a crude logging mechanism.

[@interpret-env] Interpret only if an environment variable is set, otherwise run natively. This would allow code to remain unaltered after debugging, leaving the annotation in place in case the bug is not really fixed.

[@interpret-sub] Pause the program at the expression given, printing the current expression and allowing the programmer to substitute their own.

[@show-only] Simply show a given expression, but run it natively. This is a way of adding a generic printer to OCaml (like Java's toString or Haskell's show).

[@interpret-matching] Give a search term (like OCamli) and show only those lines of the evaluation when interpreting.

[@interpret-n <n>] Show only <n> times through this code point. After that, be silent whilst continuing to run.

[@exit-after <n>] Show only <n> times through this code point. After that, exit.

[@interpret-interactive <n>] Upon interpreting, dump into an OCamli-like interface which acts as an interactive debugger, setting and clearing breakpoints and so on.

We can see that many well-known mechanisms of debugging, such as breakpointing, find a new home here. The approach is in general a low-impact one: the programmer need use only the parts of the debugging system they wish to, or which suit their mental model or debugging style. We hope that this makes the debugger more likely to be used by more people.

It is possible to imagine other interface models, of course. We could, for example, produce a REPL-like program which is able to show steps of evaluation in addition to its usual functionality, for learning or testing, or light debugging. But the annotation-based one we have alighted upon seems promising, so we persevere with it for now.

It is important to point out that for this scheme to work, there would need to be a mechanism for types to be known at the time the annotation is expanded, such that values can be reliably used across the compiled/interpreted boundary. This might require a typed analog of the PPX system, where the annotation is expanded after typechecking has occured. But this is a relatively minor modification to the OCaml compiler.

Summary   We have demonstrated how to get our interpreter inside the build process of programs written in, or partly in, OCaml, and noted that this helps us achieve many of the aims of making a debugger which is available whenever the programmer needs it. We have seen how selective interpretation could lead to much more reasonable running times for programs we wish to debug by reducing the part which need be interpreted.

4 Evaluation

We have looked again at the problem of debugging, identified what we believe to be a debugger design which may lead to more people using debugging tools, and built a proof-of-concept of part of it for the functional language OCaml. By what criteria do we measure success? The primary measure, of course, is whether the tool, once finished, is widely used. Quantitively, we can measure two things: how many people use the tool in preference to another, and how many in preference to nothing at all or more often than they used their previous tool. Does it replace or merely complement other tools?

Our plan was to look at the literature and practice of debugging, and try to discern those qualities which separate a debugger which is useful and used from one which lies unused. This was motivated by observing the widespread feeling amongst programmers that using a debugger would be more common if only it were more easily applicable to their problems. That is to say, the feeling that there is nothing fundamentally impossible about producing a widely used debugger. We looked at this in the context of functional programming especially, working on the assumption that functional programming is different

enough from imperative programming that there are likely to be some differences in the debugging process.

We have identified what we believe to be the key requirements for a usable debugger – that it should be available all the time, whenever the programmer needs it, and that it should be sufficiently flexible so as to be unobtrusive when not in use. There are, of course, many other requirements for a good debugger. But we have claimed that without the accessibility requirement being fulfilled, the rest is in vain. We chose the radical approach to these key requirements: to build a naive, step-by-step interpreter. The supposed advantages were that this would result in accessibility-by-default, that there would be no information loss (since there is no compilation process), and that the obvious downside of interpretation – slowness – could be mitigated.

OCamli was written, firstly, to answer the question "What would an interpreter for OCaml look like?", and secondly to begin to explore the design space of visualizing (and thereby debugging) OCaml programs. However, it does not pass our tests of what a good debugger would look like – and not only because it does not yet support the whole language. Let us suppose now that OCamli were to be finished to support the whole of OCaml. What would it still lack, with regard to our principles and, in particular, the tests we set out in Section 2.2? OCamli, as presently constructed, does not meet the most important of the tests. It cannot be used as an alternative to the compiler except for the simplest of projects, and cannot be used with mixed C/OCaml code. These are design flaws, and impact upon usability, in particular the notion of accessibility we have been concerned with.

OCamli, then, is an interesting exposition of our central idea of debugging-by-interpretation, but flawed with regard to usability, both in its choice of interface, and by dint of its failure to address fully the notion of accessibility. We have identified a solution to this problems, a mechanism by which the interpreter may be placed in an executable, alongside native code. We believe this could solve the remaining problems.

5   Conclusions and future work

We have looked again at the history and present practice of debugging and tried to identify the essential characteristics which separate debuggers which are likely to be used from those which are likely not to be. Working from these principles, we have described a design for, and early prototype of, a new debugger for the functional language OCaml based on the concept of direct interpretation, and a design for a mechanism for it to be embedded into the build process in such a way that it is always available. We believe it to be promising, but it is too early to say if it really represents a significant step forward – the problem of debugging is old and intransigent.

As we have discussed, the ultimate test is, of course, whether anyone uses it. So, simply put, our most important item of future work is to provide a complete implementation as a concrete way of supporting (or undermining) our thesis.

Some of the technical mechanisms we have used to create this debugger are rather specific to OCaml – to what extent might the insights gained be useful in other languages? It would be interesting to see if this mixture of interpretation and compilation can be applied elsewhere. We have, thus far, made no attempt at preserving the time or space complexity of programs under interpretation, even when the intermediate steps are not shown. Is it possible to interpret programs in such a way that we can give a guarantee about the complexity?

In addition, we have not yet formalized certain parts of the implementation, for example how to match the OCaml compiler's actual order of evaluation, in cases where it is not specified in the language – an important part of being able to reproduce bugs using the debugger. There will doubtless be several difficult little issues like this.

The kind of diagrams our interpreter draws are also useful for teaching, that is to say testing out little programs a student is writing rather than debugging large codebases. It would be interesting to look at how exactly learning to program and debugging are intertwined or equivalent tasks, and see if our interpreter helps beginning students. Besides teaching and debugging, having a small-step interpreter readily available for a language, especially one which ranks equally with the compiler and can be mixed with it at will, may have more uses which we have yet to discover.

References

[1] Mikhail Auguston & Juris Reinfields (1994): *A Visual Miranda Machine*. In: *Software Education Conference, 1994. Proceedings.*, IEEE, pp. 198–203, doi:10.1109/SEDC.1994.475337.

[2] R.M. Balzer (1969): *EXDAMS – Extendable Debugging And Monitoring System*. In: *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, Memorandum (Rand Corporation), ACM, Rand Corporation, pp. 567–580, doi:10.1145/1476793.1476881.

[3] Paul T Brady (1968): *Writing an online debugging program for the experienced user*. Communications of the ACM 11(6), pp. 423–427, doi:10.1145/363347.363388.

[4] Rod M Burstall, David B MacQueen & Donald T Sannella (1980): *HOPE: An experimental applicative language*. In: *Proceedings of the 1980 ACM conference on LISP and functional programming*, ACM, pp. 136–143, doi:10.1145/800087.802799.

[5] Arthur Charguéraud, Alan Schmitt & Thomas Wood (2018): *JSExplain: A Double Debugger for JavaScript*. In: *The Web Conference 2018*, pp. 1–9, doi:10.1145/3184558.3185969.

[6] Olaf Chitil, Colin Runciman & Malcolm Wallace (2002): *Transforming Haskell for tracing*. In: *Symposium on Implementation and Application of Functional Languages*, Springer, pp. 165–181, doi:10.1007/3-540-44854-3_11.

[7] John Clements, Matthew Flatt & Matthias Felleisen (2001): *Modeling an Algebraic Stepper*. In: *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, Springer-Verlag, London, UK, UK, pp. 320–334, doi:10.1007/3-540-45309-1_21.

[8] Youyou Cong & Kenichi Asai (2016): *Implementing a stepper using delimited continuations*. Software Science 39, pp. 42–54, doi:10.29007/l2wb.

[9] Aaron Contorer (2015): *What do Haskellers want? Over a thousand users tell us*. Available at `https://www.fpcomplete.com/blog/2015/05/thousand-user-haskell-survey`.

[10] Roger F Crew et al. (1997): *ASTLOG: A Language for Examining Abstract Syntax Trees*. In: *DSL*, 97, pp. 18–18. Available at `https://www.usenix.org/legacy/publications/library/proceedings/dsl97/full_papers/crew/crew.pdf`.

[11] Premkumar T Devanbu (1999): *GENOA – a customizable, front-end-retargetable source code analysis framework*. ACM Transactions on Software Engineering and Methodology (TOSEM) 8(2), pp. 177–212, doi:10.1145/304399.304402.

[12] Marc Eisenstadt (1997): *My hairiest bug war stories*. Communications of the ACM 40(4), pp. 30–37, doi:10.1145/248448.248456.

[13] Thomas G Evans & D Lucille Darley (1966): *On-line debugging techniques: a survey*. In: *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, ACM, pp. 37–50, doi:10.1145/1464291.1464295.

[14] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy & Sam Tobin-Hochstadt (2015): *The racket manifesto*. In: *LIPIcs-Leibniz International Proceedings in Informatics*, 32, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, doi:10.4230/LIPIcs.SNAPL.2015.113.

[15] Robert Bruce Finder, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler & Matthias Felleisen (2002): *DrScheme: A Programming Environment for Scheme*. Journal of Functional Programming 12(2), pp. 159–182, doi:10.1017/S0956796801004208.

[16] *Visual Studio Docs: Debugging F#*. Available at `http://docs.microsoft.com/en-us/visualstudio/debugger/debugging-f-hash`.

[17] Andy Gill (2000): *Debugging Haskell by Observing Intermediate Data Structures*. Electr. Notes Theor. Comput. Sci. 41(1), p. 1, doi:10.1016/S1571-0661(05)80538-9.

[18] Doug Goldson (1993): *A Symbolic Calculator for Non-Strict Functional Programs*. The Computer Journal 37(3), pp. 177–187, doi:10.1093/comjnl/37.3.177.

[19] Ralph Grishman (1970): *The debugging system AIDS*. In: *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, ACM, pp. 59–64, doi:10.1145/1476936.1476952.

[20] William G Griswold, Darren C Atkinson & Collin McCurdy (1996): *Fast, flexible syntactic pattern matching and processing*. In: *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, IEEE, pp. 144–153, doi:10.1109/WPC.1996.501129.

[21] Cordelia V Hall & John T O'Donnell (1985): *Debugging in a side effect free programming environment*. ACM SIGPLAN Notices 20(7), pp. 60–68, doi:10.1145/17919.806827.

[22] Mark Halpern (1965): *Computer programming: the debugging epoch opens*. Computers and Automation 14(11), pp. 28–31.

[23] Mark Halpern (2005): *Assertive Debugging: Correcting Software As If We Meant It – Assertive debugging is a new way to make embedded systems ensure their own health by having your code monitor itself*. Embedded Systems Programming 18(6), pp. 28–35.

[24] Dick Hamlet (1983): *Debugging "Level": Step-wise Debugging*. SIGPLAN Not. 18(8), pp. 4–8, doi:10.1145/1006142.1006150.

[25] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy & Jérôme Vouillon (2018): *The OCaml Language*. Available at `http://ocaml.org/`.

[26] Simon Marlow, José Iborra, Bernard Pope & Andy Gill (2007): *A Lightweight Interactive Debugger for Haskell*. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, ACM, New York, NY, USA, pp. 13–24, doi:10.1145/1291201.1291204.

[27] Erik Meijer & John Gough (2012): *Technical overview of the Common Language Runtime, 2000*. Available at `https://research.microsoft.com/en-us/um/people/emeijer/papers/clr.pdf`.

[28] Robin Milner (1983): *How ML evolved*. Polymorphism: The ML/LCF/Hope Newsletter 1.

[29] *The OCamli Interpreter*. Available at `http://github.com/johnwhitington/ocamli`.

[30] Cristóbal Pareja-Flores, Jaime Urquiza-Fuentes & J. Ángel Velázquez-Iturbide (2007): *WinHIPE: An IDE for Functional Programming Based on Rewriting and Visualization*. ACM SIGPLAN Notices 42(3), pp. 14–23, doi:10.1145/1273039.1273042.

[31] Chris Parnin & Alessandro Orso (2011): *Are Automated Debugging Techniques Actually Helping Programmers?* In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, ACM, New York, NY, USA, pp. 199–209, doi:10.1145/2001420.2001445.

[32] Santanu Paul & Atul Prakash (1994): *A framework for source code search using program patterns*. IEEE Transactions on Software Engineering 20(6), pp. 463–475, doi:10.1109/32.295894.

[33] Marian Petre & Ed de Quincey (2006): *A Gentle Overview of Software Visualization*. PPIG Newsletter, pp. 1–10.

[34] Steve Reeves, Doug Goldson, Pat Fung, Mike Hopkins & Richard Bornat (1994): *The Calculator Project – Formal Reasoning about Programs*. In: *Software Education Conference, 1994. Proceedings.*, IEEE, pp. 166–173, doi:10.1109/SEDC.1994.475332.

[35] Elaine Regelson & Andrew Anderson (1994): *Debugging Practices for Complex Legacy Software Systems*. In: *ICSM*, pp. 137–143, doi:10.1109/ICSM.1994.336781.

[36] Mark Shinwell: *libmonda: Make OCaml native debugging awesome*. Available at `http://mshinwell.github.io/libmonda/`.

[37] Juha Sorva, Ville Karavirta & Lauri Malmi (2013): *A Review of Generic Program Visualization Systems for Introductory Programming Education*. Transactions in Computing Education 13(4), pp. 15:1–15:64, doi:10.1145/2490822.

[38] Richard M Stallman (1981): *EMACS the extensible, customizable self-documenting display editor*. ACM Books, doi:10.1145/800209.806466.

[39] Guy Steele (1984): *Common LISP: the language*. Digital Press 20, p. 124.

[40] Don Syme: *F Sharp at Microsoft Research*. Available at `https://www.microsoft.com/en-us/research/project/f-at-microsoft-research/`.

[41] Jonathan Paul Taylor (1996): *Presenting the Lazy Evaluation of Functions*. Ph.D. thesis, Queen Mary, University of London.

[42] Andrew Tolmach & Andrew W. Appel (1995): *A Debugger for Standard ML*. Journal of Functional Programming 5, pp. 155–200, doi:10.1017/S0956796800001313.

[43] David S. Touretzky (1989): *Visualizing Evaluation in Applicative Languages*. Commun. ACM 35(10), pp. 49–59, doi:10.1145/135239.135241.

[44] David Ungar, Henry Lieberman & Christopher Fry (1997): *Debugging and the Experience of Immediacy*. Communications of the ACM 40(4), pp. 38–43, doi:10.1145/248448.248457.

[45] J. Urquiza-Fuentes & J. A. Velázquez-Iturbide (2004): *A Survey of Program Visualizations for the Functional Paradigm*. In: *Proc. 3rd Program Visualization Workshop*, pp. 2–9. Available at `https://www.dcs.warwick.ac.uk/pvw04/p01.pdf`.

[46] Jaime Urquiza-Fuentes & J. Ángel Velázquez-Iturbide (2009): *A Survey of Successful Evaluations of Program Visualisation and Algorithm Animation Systems*. ACM Transactions on Computing Education (TOCE) 9(2), p. 9, doi:10.1145/1538234.1538236.

[47] Philip Wadler (1998): *Why No One Uses Functional Languages*. SIGPLAN Not. 33(8), pp. 23–27, doi:10.1145/286385.286387.

[48] John Whitington & Tom Ridge (2017): *Visualizing the Evaluation of Functional Programs for Debugging*. In: *6th Symposium on Languages, Applications and Technologies*, OASIcs 56, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 7:1–7:9, doi:10.4230/OASIcs.SLATE.2017.7.