Debugging Functional Programs by Interpretation

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

John Whitington

Department of Informatics

University of Leicester

July 2020

Debugging Functional Programs by Interpretation

John Whitington

Abstract

Motivated by experience in programming and in the teaching of programming, we make another assault on the longstanding problem of debugging. Having explored 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 likely to be widely used. We present work on a new debugger for the functional programming language OCaml which operates by direct interpretation of the program source, allowing the printing out of individual steps of the program's evaluation, and discuss its technical implementation and practical use.

It has two parts: a stand-alone debugger which can run OCaml programs by interpretation and so allow their behaviour to be inspected; and an OCaml syntax extension, which allows the part of a program under scrutiny to be interpreted in the same fashion as the stand-alone debugger whilst the rest of the program runs natively. We show how this latter mechanism can create a source-level debugging system that has the characteristics of a usable debugger and so may eventually be expected to be suitable for widespread adoption.

Acknowledgments

This research could not have been undertaken nor this thesis written without the continual support and good humour of my supervisors Tom Ridge, Neil Walkinshaw and José Miguel Rojas Siles. It was funded by the Engineering and Physical Sciences Research Council, part of UK Research and Innovation.

Contents

List of figures

I am in much dismay at having got into so amazing a quagmire & botheration with these Numbers, that I cannot possibly get the thing done today. … I am now going out on horseback. Tant mieux.

— Lovelace to Babbage, July 1843

Chapter 1

Introduction

All beginnings are delightful; the threshold is the place to pause.

— Goethe

This thesis addresses the question of debugging. Since the beginning of the computer age, the concept has been known, discussed, and solutions have been engineered. And yet, many programmers never touch a debugger. Is this a fundamental problem – is the whole notion of debugging tools a mirage – or is it simply that the right solutions have yet to be found?

And so, we have another go at the problem of debugging. It might be argued that "having another go at something" does not constitute research. We disagree wholeheartedly. To re-tread a path well-trodden with failure or limited success is not simply to repeat. To take the experience of the past with the technologies of today, beat back the weeds, and see if we can put some proper paving down, is worth the effort – and an essential part of the research process.

We concentrate our efforts on functional programming languages: long a niche area with promise; suddenly in the past ten years a mainstream field. There are certain characteristics of the functional way which are amenable to a different kind of debugging, one which we will show alleviates many previous barriers to practical debuggers.

We have produced some new debugging tools which show promise. We have learnt a great deal about the possibilities and inherent limitations of debugging. But, before all that, let us pause, and set the work in context.

1.1 Motivation from teaching

When teaching functional programming, we like to draw diagrams on paper like that shown in figure 1. Such diagrams help to visualize the computation not only in terms of what it computes, but the number of steps taken (roughly the number of lines in the diagram), the stack space required for a function which is not tail-recursive (roughly the width of the diagram), and the subexpression being evaluated at any step (the underlined portion). They can also be useful to the more experienced programmer as a debugging aid.

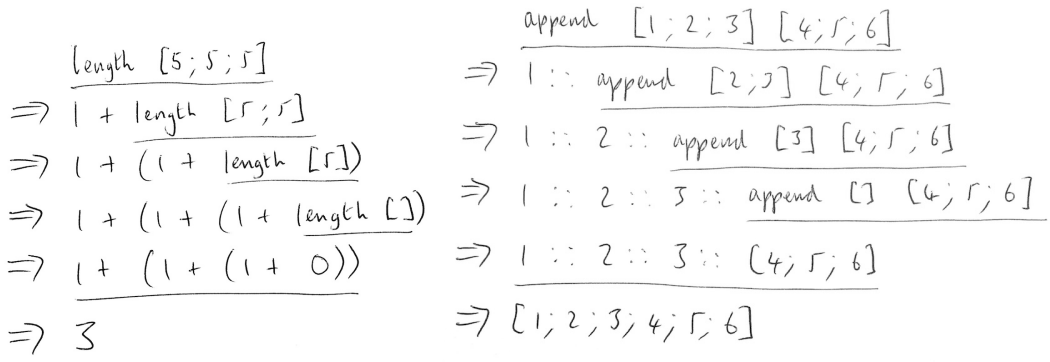


Figure 1. Handwritten diagrams for finding the length of a list and for appending two lists.

Stepping through a computation is one of the building blocks of debugging, though this is typically done with a compiled form, where much information is not available. Stepping through in source form means we have all information to hand.

What makes it possible to draw these diagrams on paper is the immutable nature of most functional programs. Stepping through an imperative program on paper involves drawing little boxes as storage locations and updating them, rubbing out and writing in new values as the program runs. However, whilst we need only write the source code of the imperative program once, in our functional diagrams we have had to repeatedly write out almost the same expression line after line. Such repetition suggests automation via computer would be useful. Notice also that our diagrams, when written by hand, implicitly omit information, such as the definitions of the functions and some of their internal workings. Thus, such automation will entail significant visualization and efficiency challenges, especially with large programs.

1.2 Motivation from the working programmer

The dream of debugging is this: having observed a misbehaviour caused by a bug, we quickly and easily assemble all relevant information, both about the program source and the full trace of the program's operation, and, describing ourselves concisely to the computer, we narrow the circumstances of the failure down, again and again reducing the search space, until we have the bug in our grasp, and understand it fully. The fix is then often easy.

And yet, this dream so often falls apart. Our program has been compiled and only limited debugging information is available, creating a lossy map back to the source locations, and we are in the dark. Our tools might not operate on all of the program for obscure technical reasons. We may lack a way of converting our intuition about what might cause the misbehaviour into a command to give the debugger to aid in the narrowing-down process. We throw the debugger away and insert a few print statements and spend an afternoon on what, in our dream world, we could fix in minutes.

We hope to show that, whilst debugging will always remain a somewhat mercurial, individual process, there is plenty of scope for improvement in the tools – provided we

stick steadfastly to some important principles.

1.3 Research questions

It is useful, when beginning a piece of research, to write down the questions we have in our mind. As we refine the scope during literature review and, of course, due to discoveries during the design and implementation of our tools, we will doubtless drop some of these, add others, and change emphasis. Nonetheless, we list now the research questions, as they were written down early in the project. We shall return to them when evaluating our work.

It appears that debuggers are not as widely used as one might expect, despite being common for decades. Why? This appears to be doubly true for functional programming languages. How does debugging practice vary among languages (compiled vs interpreted, stateful vs stateless)? What can we learn from debugging theory and practice since the dawn of the computer age?

Can we find a good way to visualize functional program execution? Is the automatic production of such diagrams always going to be inferior to drawing them on paper? How can we deal with scale? How can we show exceptions? What about imperative and mutable features? What are the practicalities of directly interpreting an Abstract Syntax Tree? Can the direct interpretation of the AST of a program ever or always have the same time or space complexity as running the compiled program?

Is such an interpreter useful for debugging? Taking into account of the current practice of debugging, is an interpretive debugger better than what is already available? If so, why, and in what ways?

Is there an alternative abstract machine which might allow for this kind of visualized debugging? That is to say, are we condemned to interpret the AST in the simplest way if we want to be able to properly visualize the evaluation in a human-readable manner? We know that this may have greater complexity of running time than well-known abstract machines. Can we design a bytecode that retains the ability to produce source code for the running computation upon demand, but which is much faster than brute interpretation and maybe even close to that of a normal bytecode system?

Could we, instead, build an interpreter which can work alongside the native code execution of a program, interpreting only when required? We could compile a program in a slightly different manner. It would run as usual, but when it comes to a part we wish to debug, it would begin interpretation. After this part, it would return to native code, as if nothing had happened.

1.4 Thesis

It is as well to foreshadow our central claims now, though we have not yet given support for them. We claim:

- That not many programmers use debuggers, even though they exist.
- That (almost) everyone could benefit from a debugger.
- That the reasons for this disparity are frequently incidental rather than intrinsic and include:
  - The inability of debuggers working on compiled programs to properly reflect their workings at a source code level.
  - The requirement to adapt build systems, tools, and environments to ensure everything is ready for the debugger to be used at any time.
- That many of these barriers melt away in the presence of an interpreter ranking equally to a native compiler, with the same language and toolchain support.
- That the huge disadvantage of slowness which comes with this approach:
  - Can be ameliorated more that one might expect.
  - Is not a show-stopper for most uses, since the external steps of debugging such as case reduction are still in place.
  - May be obviated by finding a way to produce mixed native/interpreted programs so that, in any case, the need for interpretation is much reduced.
- That such an interpretive approach is particularly suited to functional programs due to the mental model of calculation.

Our thesis is this: an interpretive debugger for a functional language is technically practicable and might be expected to make debugger use more widespread, routine and productive.

1.5 Contributions

The contribution of a practical tool-based piece of research such as this should be twofold: a new program or piece of code; and a new approach to a problem, exemplified by that tool, but which has reusable lessons for future researchers and implementers. Our principal practical, or technical contributions are:

- A direct interpreter for syntax trees of the statically-typed strict functional language OCaml [Leroy et al., 2018], demonstrating how such interpretation might work for real, sizeable programs.
- A mechanism by which the program is run natively, save for the focus of debugging, which is interpreted, forming a single executable which prints out step-by-step evaluation only of a given part of the program.

The implementation of these tools is unfinished; but enough exploratory work has been done to be reasonably sure that finishing them is a practical, not a research task. In terms of our wider contribution, our research output has also created:

- A narrative literature review of the history of debugging – in particular the very early development of debugging – from many perspectives, putting modern debugging into context.
- A survey of modern debugging systems for functional programming languages, sketching the similarities and differences between systems for lazy and strict languages, and following progress over time.
- A collation of the characteristics which make a debugging system usable, and thus likely to be used. These rules, we believe, transcend our focus on functional programming, and should apply more widely.

1.6 Note

This thesis contains paragraphs whose content is similar or, in some cases, identical to those in our recent papers [Whitington and Ridge, 2017b, Whitington and Ridge, 2019].

1.7 Summary

We have given a little preview and summary of the work in this thesis, without recourse to too much technical detail. We have described various concrete decisions: (1) to restrict ourselves to functional programming; (2) to work by analogy to drawn diagrams; and (3) to try to build an interpreter which ranks equally with the compiler. The first two of these decisions were taken right at the beginning of the project and, not being countermanded by the evidence gathered later, still stem from that time. The third, as an explicit aim, emerged only after the literature review. And so the justification for this latter decision must wait for chapter 3.

Before coming to that technical detail, we pause again, to review the history and present practice of debugging.

Chapter 2

Related work

> *We come mentally of age when we discover that the great minds of the past, whom we have patronized, are not less intelligent than we are because they happen to be dead.*
>
> — Cyril Connolly, *The Unquiet Grave*

In order to choose how to attack our problem, we shall need to know what has come before and to what extent it has succeeded or failed. We begin with an informal survey of the academic literature in debugging from just after the Second World War. Then, we shall look at recent work in functional languages, since that is our particular area of interest. Finally, we shall look at modern debuggers for imperative languages, seeing if there is anything to learn from them which we might apply to the functional world.

2.1 Seventy years of debugging 1949–2019

We take a broad look at debugging literature of all kinds over the last seventy years, seeking to tease out the principles which survive and problems which resist the enormous changes associated with the so-called computer revolution. We shall see the same ideas, or rhymes of ideas, again and again. This gives us hope – the problems are agreed upon, but also trepidation. If no-one has solved them yet, why should we imagine we can?

2.1.1 Debugging defined

Before we discuss debuggers, and in particular our new debugger, we ought to make sure our basic definitions are secure. What is debugging? In "A Software Debugging Glossary", Johnson defines it like this: *debugging: 1) n. the process of isolating and correcting mistakes in computer programs. 2) adj. having to do with debugging* [*Johnson, 1982*]. Most important here is the word 'process'. Already we can see the separation between isolation and correction, and that there is an order: isolate first, then correct. In his survey "Testing: principles and practice", Schach uses one method of isolation, that of testing, to define debugging: *The correction of a fault exposed by testing is termed debugging* [*Schach, 1996*]. If we use a sufficiently broad definition of testing, to include accidentally coming across a fault as well

as structured, active testing, this is as useful as the previous definition. Ko contrasts again detection of errors (called testing) and debugging: *Whereas verification and testing detect the presence of errors, debugging is the process of finding and removing errors* [*Ko et al., 2011*]. So, we begin debugging when and only when we have found a fault. Parnin and Orso break down the process into three subprocesses, and define debugging as the sum total of those:

> *The first activity, fault localization, consists of identifying the program statement(s) responsible for the failure. The second activity, fault understanding, involves understanding the root cause of the failure. Finally, fault correction is determining how to modify the code to remove such root cause. Fault localization, understanding, and correction are referred to collectively with the term debugging.* [*Parnin and Orso, 2011*]

Scowen subdivides debugging tools into two kinds, based upon when they do their work in relation to the point at which the existence of a bug is identified by the programmer:

> *Debugging tools can be classified as active or passive. An active tool is one which enables the programmer to specify what he wants after he has realized there is an error… A passive tool works automatically without any effort from the programmer, e.g. failure messages, store post-mortem, etc.* [*Scowen, 1972*]

In a nod to debugging as an essentially human process, in which boundaries are blurred, and in which rigour and instinct play equal roles, Hailpern and Santhanam advise us: *Note that the terms 'debugging', 'testing', and 'verification' are not mutually-exclusive activities, especially in everyday practice* [*Hailpern and Santhanam, 2002*]. In contrast, in line with the wholly rigorous procedures beloved of certification authorities, here is Ko's precis of the relevant IEEE definitions of the conditions leading to the need for debugging – what we and our quotations have hitherto referred to as faults or errors:

> *In this paper, we use definitions of error, fault and failure from IEEE standard 610.12-1990. A failure occurs when a program's output does not match documented output requirements, or the programmer's mental model of output requirements. Failures are ultimately the result of a fault, which is a run-time state of a program that either is or appears to be incorrect (as in assuming a lack of output from a debugging print statement to mean the code is not reached). Faults occur as a result of errors, which are program fragments that do not comply with documented program specifications or the programmer's mental model of specifications (such as a missing increment or misspelled variable name). Failures are usually the first indication to the programmer that one or more errors exist in the program, although some errors are found before they cause failures, since they may be similar to errors already found or may be apparent when inspecting code. While a failure guarantees one or more faults exist and a fault guarantees one or more errors exist, errors do not always cause faults, and faults do not always cause failures.* [*Ko and Myers, 2003*] *summarising* [*IEEE, 1990*]

It is worth remembering that debugging may (and frequently does) exist, of course, entirely in the absence of debuggers. From simply staring at code until one finds the fault, to experimenting with multiple inputs or inserting print statements to localise the problem. In fact, one of our claims is that not many people use debuggers (or at least not regularly), and one of our questions is simply "Why Not?". We must answer these questions first if we are to devote time to constructing yet another debugger. Here is Taylor, on the practical uses of print statements:

> *The use of print statements is one of the most common techniques adopted by programmers using an imperative language, when faced with a program whose behaviour they do not understand. By using print statements programmers can achieve a number of goals: they can confirm that a piece of code is being executed; they can trace the state of a number of variables; they can trace when a procedure is being called and with what arguments, and so on.* [*Taylor, 1996*]

Whilst considering what debugging is, it is worth wondering whether it could be eliminated. So far, this has proven largely to be a false dawn – even today, very few programs are formally proved correct (in any event, one may need to debug the proofs).

> *Considerable interest has been shown in recent years in the development of methods for proving that a given computer program has certain properties. If this avenue of research proves successful, we may one day see the virtual elimination or at least diminution in importance of the program debugging process.* [*Evans and Darley, 1966*]

> *Debugging would be unnecessary if programs could be proved to be correct … and debugging is likely to be necessary for some time yet.* [*Scowen, 1972*]

Dijkstra gives a similar argument:

> *If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.* [*Dijkstra, 1972*]

> *Already now, debugging strikes me as putting the cart before the horse: instead of looking for more elaborate debugging aids, I would rather try to identify and remove the more productive bug-generators!* [*Dijkstra, 1971*]

Now that we have defined debugging, let us continue on by defining debuggers (and in terms better than "something which helps one debug"). Again, let us begin with Johnson: *debugger: n. a collection of software tools to aid in debugging* [*Johnson, 1982*]. It seems very common that the debugger is considered a separate tool, and not part of the programming language or its compiler. Yes, a compiler may add special information to an executable to help the debugger, and yes, a compiler may ship with a debugger as part of an Integrated Development Environment, but programmers tend to think of the debugger, however

convenient to use, as a separate entity. Satterthwaite explains the role in a little more detail: *...a debugging tool is a service provided by the computing system to reduce the number of aspects of a program's behaviour which are misunderstood or poorly understood by the user* [*Satterthwaite, 1972*]. However, he is careful to point out an essential tension: we only need debuggers because we are inadequate programmers: *Debugging tools are not intended to relieve the programmer of any obligations to analyse his problem carefully or to construct his program in a disciplined manner* [*Satterthwaite, 1972*]. Gaines agrees but, even writing in 1969, it is clear to him that, whilst experimental programming is usually undesirable, it happens anyway: *Occasionally it is difficult to distinguish between debugging and writing of programs. This arises because the programmer is writing a program 'experimentally'* [*Gaines, 1969*]

We must try as hard as we can, using talent, discipline and all the tools of modern programming languages such as compilers and type systems, but in the end, there will likely always be the need for debugging as a process distinct from, and happening after, programming itself.

2.1.2 Early thoughts

Before continuing on to a detailed discussion of debugging theory and practice, it will be instructive to look at some of the very earliest published descriptions of what we now call debugging from those working near the beginning of the computer age, just after the Second World War. These early papers, being at the very beginning of the field, are written in a straightforward, practical way. They are often the first thoughts about topics and definitions and words we now take for granted. Looking back at how these thoughts were first formed can give an interesting perspective to our work on modern-day debugging. Our principal sources will be Maurice Wilkes and Stan Gill's descriptions of their work on the EDSAC computer [Wilkes, 1949, Gill, 1951], and Ira Diehm's 1952 ACM National Meeting article "Computer aids to code checking" [Diehm, 1952].

Writing in 1949, Wilkes has to define even the word programming for us, bringing into focus that, at that time, building the physical computer was the primary difficulty, and providing for it the correct sequence of instructions to make it perform the correct calculations for solving a given problem a secondary concern.

> *A good deal has been written about the design and construction of high-speed automatic calculating machines, but little has been said about the detailed steps which are necessary to prepare a problem for the machine and to obtain a solution – a process which is usually referred to as 'programming'.* [*Wilkes, 1949*]

We are reminded also of the division of labour. The job of the computer to silently and quickly obey a set of instructions; that of the programmer to provide the correct list of instructions. Again, this seems obvious now, but was perhaps less so at the very beginning.

> *The EDSAC is designed on the supposition that the programmer is responsible for making sure that his programme is correct and that the operations called for are correctly carried out by the machine.* [*Wilkes and Renwick, 1949*]

This division of labour is what we would now call an abstraction. Writing decades later, in his 1985 book "Memoirs of a Computer Pioneer", Wilkes explains that naivety about the relative difficulty of building a computer and programming it, and its implication: that debugging was important.

> *As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.* [*Wilkes, 1985*]

Writing in 1951, after only two years working with the machine they had built, the experiences of programming and debugging were already well-known, and recognisable in modern terms.

> *Experience has shown that such mistakes are much more difficult to avoid than might be expected. It is, in fact, rare for a program to work correctly the first time it is tried, and often several attempts must be made before all errors are eliminated.* [*Wilkes, 1951*]

The notion that most programs will require debugging was also discussed by Deihm and so we can imagine that, by the early 1950s, it was widely understood. *When a complex routine is tried on a computer for the first time, it is seldom found to be free from error* [*Diehm, 1952*]. Wilkes was well aware that programming, even then, involved two tasks with regard to correctness: trying to get the program right the first time and, knowing that it probably would not be, finding and fixing the remaining mistakes after the program had been fed to the computer and a misbehaviour noticed.

> *Since much machine time can be lost in this way a major preoccupation of the EDSAC group at the present time is the development of techniques for avoiding errors, detecting them before the tape is put on the machine, and locating any which remain undetected with a minimum expenditure of machine time.* [*Wilkes, 1951*]

Even then, it was clear that the diagnosis of the problem was much harder than simply knowing that there was a problem. This is another example of an observation which seems obvious in retrospect, but must not have been so at the time.

> *The difficulty lies not in detecting the presence of a mistake, but in diagnosing it. In practice its presence is nearly always obvious, for the character of most programmes is such that even a slight error will usually have an extensive effect.* [*Gill, 1951*]

Gill goes on to explain that, even in the days of early machine languages, many of the mistakes could be found beforehand with simple checks. Of course, these days we expect these "simple checks" to be done automatically by the use of compilers, type-checkers and so forth.

> *...experience with the EDSAC has shown that although a high proportion of mistakes can be removed by preliminary checking, there frequently remain mistakes which could only have been detected in the early stages by prolonged and laborious study. Some attention, therefore, has been given to the problem of dealing with mistakes after the programme has been tried and found to fail.* [*Gill, 1951*]

Having looked at early echoes of our modern debugging experience in these old texts, let us consider some of the differences. For example, Gill has to draw a distinction between program runs whose result is incorrect due to a faulty program being supplied and those which are incorrect due to the computer itself malfunctioning silently:

> *Two kinds of mistakes, or blunders, arise in the use of an automatic digital computing machine: (i) those resulting from faults in the machine itself, and (ii) those arising because the orders or data presented to the machine are not those required to obtain the results sought.* [*Gill, 1951*]

The same was true even for an output device. Wilkes explains how a number sent to an output device to be printed could be read back in again to check it survived the transport (this was before the invention of error correcting codes):

> *Included in the order code* [instruction set] *is a special order which causes the number set up on the teleprinter during the last output cycle to be read and placed in the store. The programmer can then take such steps as are necessary to verify that this number is the same as the one he intended to print.* [*Wilkes and Renwick, 1949*]

Let us turn to the first mentions of specific mechanisms for debugging in these texts. So-called single order operation is the most primitive method of debugging once the program was in the computer (as opposed to checking the program manually beforehand). *All high-speed computers are equipped with a means of causing orders to be obeyed singly, at the press of a button, to enable the progress of a calculation to be followed by eye* [*Gill, 1951*]. Of course, such single order operation is rather like doing a dry run of the program on paper before feeding it to the machine. Gill cautions that even for the small programs they were writing then, such a method of debugging was hardly natural or productive: *Single order operation is a useful facility for the maintenance engineer, but the programmer can only regard it as a last resort* [*Gill, 1951*].

It is clear from Diehm's paper that, even in these early moments, the thought that the computer itself may be able to help with debugging in more sophisticated ways than single order mode was present.

> *Careful proofreading and clerical checking are obvious but important methods of eliminating errors before going to the computer. On the other hand, our machines are intended to help to eliminate such drudgeries, so that we are interested in how the machines themselves can be used to analyze coding errors.* [*Diehm, 1952*]

Indeed, a precursor to tracing or logging was present in the early use of the EDSAC, where the primitive output device could be used to write a record of certain register or data values during the run of the program.

> *It might be thought that a good way of finding errors in a program would be to make the machine proceed order by order under the control of the "Single E.P." button … and to study the numbers in the machine by watching the monitors attached to the arithmetical unit and store. This, however, usually turns out to be a very slow and inefficient process, especially as the numbers are displayed in binary form. Methods have therefore been developed which permit the machine to proceed unhindered by the operator, whilst printing on the teleprinter a permanent record that can be studied at leisure, and that will assist in understanding the nature of the mistake.* [*Wilkes, 1951*]

The utility of abstraction (through subroutines), and the intuition that testing, debugging and abstraction are interlinked but separate human activities are both clear, even in early texts.

> *Library subroutines are all checked on the machine before being put into the library and are presumably free from error. This in itself would be a sufficient reason for having a library, quite apart from any other considerations. When subroutines are specially made for a particular program it is good practice to test them beforehand by means of short programs constructed for the purpose.* [*Wilkes, 1951*]

Some of these early feelings turn out to be less relevant given the passage of time and the great increase in computing power and abstraction mechanisms which have resulted. For example, the following paragraph feels rather foreign, perhaps because of the fact that machine errors are rarer, and that machine time is no longer at a premium.

> *The important principles to be followed, I believe, are that the procedures to be used at the computer should be planned in advance and mechanized as much as possible. There should be no attempt at human analysis of errors while at the computer. It is my feeling that the computer should not be operated by the programmer, for he will not follow a predetermined plan, but will make on-the-spot improvisations which are usually regretted later, and will leave the machine idle while he speculates on possible causes of errors. The operation is best done by another person, preferably one who devotes most of his time to computer operation and is, therefore, completely familiar with the controls and has an intuitive feeling, gained from experience, of possible machine errors.* [*Diehm, 1952*]

Already we recognise terminology which is familiar today: *The term 'break-point' is used to mean a special programmed halt which may be overridden by a manual switch* [*Diehm, 1952*]. Here are some of Diehm's early facilities, provided by auxiliary routines loaded into the computer at known addresses before the program is loaded:

- *An interpretive routine which provides a complete history of a specified cell* [storage location].
- *An interpretive routine which determines the path which the control has taken through another routine.*

- *A routine which determines the total effect, on the high speed memory, of each of several chosen sections of the routine being tested.* [*Diehm, 1952*]

Already the problems of scale are evident. It is no use having reams of debugging output if the programmer is overwhelmed by it:

> *What one tries to achieve in designing such auxiliary routines is to program the machine to select the pertinent information rather than to read out large quantities of data which must be searched through by the programmer.* [*Diehm, 1952*]

The early debugging process of the EDSAC, as described in "The Preparation of Programs for an Electronic Digital Computer" (arguably the world's first textbook on computer programming) include:

- A list of *"points to be checked"* for the manual proofreading of programs.
- The *'post-mortem'* method of debugging where a program is stopped and then, without clearing the memory, a little debugging program tape is loaded which can print out parts of the store.
- *"Method using extra output orders"* which is rather like inserting print statements today. They suggest using these extra output orders in the original program, rather than waiting until a bug is found.
- Subroutines for checking programs, where the original program is effectively interpreted or emulated by a debugging routine wrapped around it.

For another perspective of early debugging experience, here is Brooks, looking back on a long career, recalling:

> *Early machines had relatively poor input-output equipment and long input-output delays. Typically, the machine read and wrote paper tape or magnetic tape and off-line facilities were used for tape preparation and printing. This made tape input-output intolerably awkward for debugging, so the console was used instead. Thus debugging was designed to allow as many trials as possible per machine session.*
>
> *The programmer carefully designed his debugging procedure – planning where to stop, what memory locations to examine, what to find there, and what to do if he didn't. This meticulous programming of himself as a debugging machine might well take half as long as writing the computer program to be debugged.* [*Brooks, 1995*]

The advent of on-line debugging (where the programmer can interact with the debugger during the debugging process in a conversational style) came with larger and more powerful computers in the 1960s. Evans and Darley describe the similarities and differences:

> *What's so new about on-line debugging? Nothing really; current on-line debugging techniques are the result of a gradual development from the days when debugging at the computer console was the norm, as it has remained for small computers over the years. Debugging methods based on single-stepping through parts of a program and*

> *on examination and modification of memory registers by means of console lights and switches were the natural precursors of today's more sophisticated techniques, and there is no sharp dividing line at any stage of the progression. Perhaps the critical step was the replacement of console lights and switches by some typewriter-like device as the principal means of communication between user and machine.* [*Evans and Darley, 1966*]

The description of the debugging system FLIT (FLexowriter Interrogation Tape) for the early 1960s TX-O system at MIT provides an insight into early online debugging:

> *With FLIT, for the first time, it was possible for the user to examine and modify his program in terms of the symbols used in his source program and, in fact, to examine and change the contents of registers in a form almost identical to that used in the corresponding assembly language.* [*Evans and Darley, 1966*]

The typical experience of offline and online debugging with such a system is described in the Programming Manual for the DEC PDP-8 'DDT' debugger:

> *Users of most computers, especially large-scale ones, are familiar with the procedure of submitting a new program for a computer run, waiting for it to be processed (which may take anywhere from a few hours to several days), and finally receiving the compilation and/or assembly listings, a list or dump of the contents of each core memory cell at the time the run was terminated, and perhaps a storage map giving the addresses of the symbols used in the programs. The user may get a few remarks from the computer operator regarding the failure of the program to run properly. If the user is present in the machine room when his program is processed, he may get additional information from the console lights, motion of tapes, etc., but his correcting must be done away from the computer. Getting a program to work under these conditions takes considerable time.*
>
> *DDT (DEC Debugging Tape) helps shorten this debugging time by allowing the user to work on his program at the computer, to control its execution, and to make corrections to the program or its data. For example, tracking down a subtle error in a complex section of coding is a laborious and frustrating job by hand; but with the breakpoint facility of DDT-8, the user can interrupt the operation of his program at any point and examine the state of the machine. In this way, sources of trouble can be located quickly.* [*Digital Equipment Corporation, 1967*]

Note the disadvantages of offline debugging and the advantages of online debugging given here. Balzer, writing in 1969 after a decade of compiled, high-level languages, explains the complications created by this shift – the need to map back to the source from the executable.

> *The debugging systems for higher-level languages are much more complex than those for assembly code. They must locate the symbol table, find the beginning and end of source-level statements, and determine some way to extract the dynamic information –*

> *needed for debugging – about the program's behavior, which is now hidden in a sequence of machine instructions rather than being the obvious result of one machine instruction.* [*Balzer, 1969*]

Some voices, by the 1960s, saw that debugging would become ever more important, as systems became larger and more complicated:

> *Oedipus is based on the premise that debugging is a central problem, perhaps* the *central problem, in the implementation and in the operation of any large programming system. Furthermore, conventional debugging aids* (*octal snaps and post mortems*) *are hopelessly inadequate in the face of dynamic storage allocation and recursion.* [*Brown, 1965*]

Dynamic storage allocation and recursion are two important aspects of typical functional programming systems, incidentally. Let us move on now, to think about why so much of this difficulty in debugging persists.

2.1.3 The difficulty of debugging

Is debugging difficult, or is it just that programming is difficult, and debugging is really just programming? The consensus in the literature is that debugging can, for the most part, be thought of as a separate activity, especially when it occurs after the program is thought to be complete. As early as the 1960s this observation is common: *Programmers seem to spend more time trying to find out why the programs they have written do not work than all other activities put together* [*Gaines, 1969*]. Or, more pithily, from "The Elements of Programming Style": *Debugging is twice as hard as writing the program in the first place* [*Kernighan and Plauger, 1978*]. These sentiments are still widespread in recent times, showing that all the improvements in programming languages do not seem to have translated into a fundamental shift in the difficulty of debugging.

> *Debugging is notoriously difficult and extremely time consuming.* [*Parnin and Orso, 2011*]

> *Nobody would claim that debugging software is easy: all too often it proceeds by trial-and-error experiments in which programmers examine the behaviour if the system and form hypotheses that could explain what they see.* [*Harris, 2002*]

In his book on debugging "Why programs fail: a guide to systematic debugging" Zeller explains that whilst many things have changed, the difficulty of debugging has been ever present:

> *Improved programming languages and tools can supplant, but not eliminate debugging, by statically identifying errors and by dynamically detecting invariant violations.* [*Zeller, 2009b*]

Harris points out another set of modern challenges, that of debugging communicating, distributed systems:

> *These problems* [of debugging] *are exacerbated when developing distributed, peer-to-peer or multi-processor applications, or when unreliable network links form part of the system under test. Environments for pervasive computing take this to an extreme, allowing user-supplied code to run or migrate within and around the network.* [*Harris, 2002*]

There has been an enormous increase in the scale of computer programs we now routinely write using modern, structured languages, so to have expected otherwise may have been too much. Plainly the improvements to programming language toolchains such as structured programming and type safety have lead to better programs and better programming, but improvements to debugging seem at best to keep pace – the war is one of attrition. So why are programming and debugging intrinsically hard? In his landmark software management retrospective "The Mythical Man-month", Brooks explains:

> *First, one must perform perfectly. The computer resembles the magic of legend in this respect, too. If one character, one pause of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.* [*Brooks, 1995*]

With regard to debugging in particular, it is a task which is unpleasant to most people as well as difficult. When a bug is found, the programmer is apt to be annoyed rather than intrigued, since their failure has been exposed. If someone else has written the program one has to debug, understanding it is another unpleasant task. Katz, in his analysis of bug-location strategies, writes: *Programmers often report that instead of debugging someone else's program, they would rather write their own version* [*Katz and Anderson, 1987*]. In his 1979 book "The Art of Software Testing" – which is mostly about testing but a lot about debugging too – Myers writes: *Debugging appears to be the single part of the software-production process that programmers seem to abhor the most* [*Myers, 1979*]. Brooks explains why:

> *The next woe is that designing grand concepts is fun; finding nitty little bugs is just work. With any creative activity come dreary hours of tedious, painstaking labor, and programming is no exception.* [*Brooks, 1995*]

But why specifically is debugging difficult? A common theme amongst explanations in the literature is that the abstractions we associate with good programming practice – and so modularity and what Dijkstra called "separation of concerns" – break down under the requirements of debugging: *Often, the code relevant to the task is scattered across many modules, increasing the difficulty of the task* [*Robillard et al., 2004*]. This includes, of course, not just the present state of the code, but the project's history. From the LaToza et al. study of the work habits of programmers:

> *Developers must know or obtain a variety of information to successfully understand and edit code – what code to change, how design decisions are scattered across code, the rationale or history behind decisions, the slice affecting a variable's value, the owner*

> *responsible for editing the code, other developers currently editing it, which changes will break code elsewhere, and which changes elsewhere affect it.* [*LaToza et al., 2006*]

> *When debugging, programmers view programs in ways that need not conform to the programs' textual or modular structures. In particular, the statements in a slice may be scattered throughout the code of the larger program and yet experienced programmers routinely extract the slices from a program.* [*Weiser, 1982*]

The term 'slicing' refers to calculating the part of the program which can lead to the alteration of a value at a given storage location:

> *Computer programmers break apart large programs into smaller coherent pieces. Each of these pieces: functions, subroutines, modules, or abstract data types, is usually a contiguous piece of program text. The experiment reported here shows that programmers also routinely break programs into one kind of coherent piece which is not contiguous. When debugging unfamiliar programs programmers use program pieces called slices which are sets of statements related by their flow of data. The statements in a slice are not necessarily textually contiguous, but may be scattered through a program.* [*Weiser, 1982*]

There are many kinds of slices, categorized by Penney:

> *The forward slice consists of all program statements affected by the slicing criterion, whereas the backward slice represents those statements upon which the criterion depends. Static slices are based on the static analysis of source code. Dynamic slices take account of program input and particular test cases.* [*Penney, 2000*]

In his paper "Some psychological evidence on how people debug computer programs", in which he develops what he calls "a gross theory of debugging", Gould writes:

> *Debugging computer programs is difficult for several reasons. The programmer must simultaneously keep track of several aspects of the program's detailed procedure specification, but the ability to do this is severely restricted* (*e.g. Yntema 1963* [A study on human memory recall]). *Second, the variety within all examples of many other systems that are regularly diagnosed* (*e.g. automotive, human anatomical, or plumbing systems*). *Third, debugging* (*as well as writing*) *a program requires a degree of precision because computer systems are unrelenting in their demands for accuracy.* [*Gould, 1975*]

Looking more closely at the debugging process, Ko et al. discuss debugging as a concretion, beginning with the abstract notion of the bug and finding concrete debugging actions to take to clarify and debug it.

> *What makes debugging difficult in general is that programmers typically begin the process with a 'why' question about their program's behavior, but must translate this question into a series of actions and queries using low-level tools such as breakpoints and print statements.* [*Ko et al., 2011*]

Gaines again, on the spectrum of difficulty, bug-to-bug: is it perhaps this heterogeneity which makes debugging a task where repeatability is limited?

> *The degree of difficulty the programmer experiences in isolating a bug once he has noticed an error depends on the nature of the bug and the ease with which he can obtain additional information about intermediate states in the computation.* [*Gaines, 1969*]

The gulf we have already mentioned, between the amount of effort needed to find a bug and the amount of effort needed to fix it, is explained by Metzger in a way which also validates our instinct that, whilst programming and debugging are often intertwined, they are separate in nature:

> *Coding, designing, analyzing, and testing are all constructive activities. They each produce a tangible result. Coding produces source code. Designing produces design documents. Analysis produces a variety of documents, depending on the methodology used. Testing produces test cases and reports on the success or failure of tests.*
>
> *In contrast, debugging is primarily a cognitive activity. The end result is knowledge of why there is a problem and what must be done to correct it. There will be a source change, but it may only involve adding or deleting a single character or word. The constructive output of debugging is often disproportionate to the effort expended.* [*Metzger, 2004*]

There are fundamental problems to these kinds of analyses and conversations, in terms of how we talk to each other about our processes.

> *An attempt to find out what people do when they are debugging by asking a number of programmers has only served to demonstrate the complexity of the problem and the general inability, well-known to psychologists, of people to describe what they do when they are involved in complex mental activity.* [*Gaines, 1969*]

But this should not prevent us attempting such analyses. We often know it when we see it. Myers suggests some reasons for the unpleasantness of debugging (and such unpleasantness and difficulty are closely linked, for most people):

Psychologically difficult *"...because it is an indication that they* [the programmer] *are less than perfect"*.

Mentally taxing Both intrinsically, and because of external pressures (getting a release to a customer, *"self-induced pressure"* or other stressors).

Location unclear *"The location of the error is potentially any statement in the program"*. Myers contrasts this with fault-location in another area – physical systems such as vehicle maintenance – where it is often easy to know which subsystem is likely at fault given the symptom observed.

Lack of resources *"…comparatively little research, literature, and formal instruction exists on the process of debugging."* [Myers, 1979]

So debugging is difficult, but we have been working on tools to aid debugging for decades, so how do we explain the persistence of the debugging problem?

2.1.4 Our lack of progress

As early as 1965, surprise was being 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", 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.* [*Halpern, 1965*]

Balzer explains that this surprise was widespread in the industry, as became clear 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 realized. 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.* [*Balzer, 1969*]

Hamlet suggests a possible reason – the lack of equivalent progress in debugging paradigms, suggesting that high-level languages might require different kinds of debugging tools rather than mere analogs of low-level 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.* [*Hamlet, 1983*]

It is fascinating to see Halpern writing on the same topic again, forty years later:

> *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.* [*Halpern, 2005*]

Thirty years later, debuggers were still not widely used, 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* [*Regelson and Anderson, 1994*]. Still later, in recent years, debugging in industry is sporadic. Parnin and Orso, 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.* [*Parnin and Orso, 2011*]

This "reliance on a strong set of assumptions" as the key to understanding why people do not use debuggers is a theme we shall return to again and again in this thesis. Hailpern and Santhanam tie this into the wider issue of immaturity with regard to software development practices:

> *. . . we observe that due to the informal nature of software development as a whole, the prevalent practices in the industry are still immature, even in areas where improved technology exists.* [*Hailpern and Santhanam, 2002*]

> *Designing a debugger which will deal with the whole debugging knowledge is still a challenge.* [*Ducassé and Emde, 1988*]

Most working programmers today, reading the five "maturity levels" from IBM's Capability Maturity Model [Humphrey, 1988] would recognise their own workplaces as being only at level one or two – the more haphazard and less mature:

> *1. Initial*
> *The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.*
>
> *2. Repeatable*
> *Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.*
>
> *3. Defined*
> *The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.*
>
> *4. Managed*
> *Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.*
>
> *5. Optimizing*
> *Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.* [*Humphrey, 1988*]

In academia, too, the intangibility of the debugging problem is recognised:

> *Even today, debugging remains very much an art. Much of the computer science community has largely ignored the debugging problem. Eisenstadt studied 59 anecdotal debugging experiences and his conclusions were as follows: Just over 50 per cent of the problems resulted from the time and space chasm between symptom and root cause or inadequate debugging tools.* [*Hailpern and Santhanam, 2002*]

Although, the proliferation of debugging papers does not necessarily address these practical issues: *... only 3 out of 111 papers on the technique of program slicing based techniques* [*Weiser, 1982*] *have considered issues with the use of the techniques in practice* [*Parnin and Orso, 2011*]. We shall confront this in our work too, having to evaluate our own new tool, even though it is unlikely to be in widespread use.

We shall address debugging in teaching in detail a little later, but to illustrate that the same challenges persist there, we can look at two recent pieces of work. McCauley et al., in a review of debugging literature from an education perspective, write about the lack of progress in that area too:

> *Debugging is an important skill that continues to be both difficult for novice programmers to learn and challenging for computer science educators to teach. These challenges persist despite a wealth of important research on the subject dating back as far as the mid 1970s. Although the tools and languages novices use for writing programs today are notably different from those employed decades earlier, the basic problem-solving and pragmatic skills necessary to debug them effectively are largely similar.* [*McCauley et al., 2008*]

And, in many cases, such education is simply omitted. It is not clear, in fact, that debugging is even really possible to teach (or, if it is, that, just like learning to drive, most of the education happens after the formal learning process is over). Siegmund et al. describe the debugging knowledge of subjects in their study:

> *Only half of our participants mentioned receiving debugging education, indicating that educators still assume that debugging is a minor part of software development or that students will learn it by themselves. However, there also was no significant difference between participants with and without education, indicating that the existing courses and trainings* [sic] *are indeed not more effective in teaching important debugging skills than self-learning.* [*Siegmund et al., 2014*]

Finally, we must quote at length the 1997 rallying cry of Henry Lieberman in his introduction to a special issue of the Communications of the ACM "The Debugging Scandal and What to Do About It":

> *Debugging is the dirty little secret of computer science. Despite all the progress we have made in the last thirty years: faster computers, networking, easy-to-use graphical interfaces, and everything else, we still face some embarrassing facts. First, all too often, computer programs don't work as they should. This makes software development costly. Too much buggy software reaches end users, leading to needless expense and*

> *frustration. That's unfortunate, but what is surprising is the fact that when something does go wrong, the people who write these programs still have no good ways of figuring out exactly what went wrong. Debugging is still, as it was thirty years ago, largely a matter of trial and error. What borders on scandal is the fact that the computer science community as a whole has largely ignored the debugging problem. This is inexcusable, considering the vast economic cost of debugging and emotional toll buggy software takes on users and programmers. Today's commercial programming environments provide debugging tools that are little better than the tools that came with programming environments thirty years ago. It is a sad commentary on the state of the art that many programmers name "inserting print statements" as their debugging technique of choice.* [*Lieberman, 1997*]

Does this lack of progress in debugging over the years reflect a genuine problem which has not yet been attacked in the right way, or simply that debugging is fundamentally not very tractable? We hope that this thesis will add some weight to one of those piles of evidence.

2.1.5 Teaching debugging

Debugging is a common difficulty among those learning to program [Badiozamany and Wang, 2010]. It is not often taught as a separate skill in university Computer Science courses, nor touched upon more than tangentially in most programming textbooks. Again, this is a longstanding situation: *It is interesting to note than* [sic] *an examination of 17 introductory programming texts revealed only 4 in which the subject of debugging received more than a few sentences* [*Gaines, 1969*].

In Ahmadzadeh et al's "Analysis of patterns in debugging among novice computer science students", the authors find that it is possible (indeed common) to learn a decent amount about programming without one's ability at debugging keeping up. This is perhaps another facet of the programming/debugging dichotomy to which we have already alluded.

> *We discovered that many students with a good understanding of programming do not acquire the skills to debug programs effectively, and this is a major impediment to their producing working code of any complexity.* [*Ahmadzadeh et al., 2005*]

Students seem to find debugging to be a job of roughly the same character and difficulty as industrial programmers, even though they are typically working on much smaller codebases. For example, in their study "Debugging from the student perspective", Fitzgerald et al. find:

> *Students in this multi-institutional study report that finding bugs is harder than fixing them. … Hypothesizing about the cause of bugs is an underdeveloped skill.* [*Fitzgerald et al., 2010*]

In particular, the study asks students to break down the debugging task in their minds into stages, and tabulates the difficulties encountered:

Troubleshooting stage	Subjects who found this stage most difficult
Understanding the code	4 (19%)
Testing	3 (14%)
Finding the problem	12 (57%)
Fixing the problem	2 (10%)

Murphy et al., in a qualitative study of novice debugging strategies, say debugging is fundamentally a hard skill in the initial stages of learning: *Similar to new drivers who must learn to steer, accelerate, brake, etc. all at once, novice debuggers must apply many new skills simultaneously* [*Murphy et al., 2008*]. Pea, in "Language-independent conceptual 'bugs' in novice programming" agrees. *The novice programmer works intuitively and pursues many blind alleys in learning the formal skill of programming* [*Pea, 1986*]. Vessey breaks down this difference between experts and novices, observing:

1. (*a*) *Experts use breadth-first approaches to debugging and, at the same time, adopt a system view of the problem area.*
   (*b*) *Experts are proficient at chunking programs and hence display smooth-flowing approaches to debugging.*
2. (*a*) *Novices use breadth-first approaches to debugging but are deficient in their ability to think in system terms.*
   (*b*) *Novices use depth-first approaches to debugging.*
   (*c*) *Novices are less proficient at chunking programs and hence display erratic approaches to debugging.*
   [*Vessey, 1985*]

Writing about end-user programming (for example, programming a set-top box to record a TV program), which is another form of novice programming, Ko et al. look at the informality of debugging strategies employed by such users:

> *...because end users often prioritize their external goals over software reliability, debugging strategies often involve "quick and dirty" solutions, such as modifying their code until it appears to work. In the process of remedying existing errors, such strategies often lead to additional errors.* [*Ko et al., 2011*]

Regelson and Anderson give another reason, which may be more related to the difficulty of teaching rather than learning, but this is much the same problem in practice: *Different people exhibit very different levels of debugging skill* [*Regelson and Anderson, 1994*]. Pea makes an interesting argument for the idea that misconceptions about how a computer operates need 'un-learning' before one can program and debug effectively:

> *This article argues for the existence of persistent conceptual 'bugs' in how novices program and understand programs. These bugs are not specific to how a given programming language, but appear to be language-independent. ... It is suggested that these classes of conceptual bugs are rooted in a 'superbug', the default strategy that*

> *there is a hidden mind somewhere in the programming language that has intelligent interpretive powers.* [*Pea, 1986*]

There are clearly many similarities between the difficulties industrial programmers of all abilities have in performing the task of debugging and the problems students only beginning their programming journey encounter. We might expect such similarities if the problem of debugging is fundamental or intrinsic, rather than incidental.

2.1.6 What makes a good debugger?

Many of the extracts from the literature we have already shown allude to what makes a good debugger, if only by implication. But it is useful to look to see if there are any more explicit commentary on the matter. Satterthwaite, in developing "A Philosophy of Design for Debugging" says: *Since debugging, as usually understood, is more a practical than a theoretical problem, proposed solutions must be evaluated within a framework of practical constraints* [*Satterthwaite, 1972*]. Brady, in a paper about a debugging tool for experienced users, explains that prior systems tried too hard to be approachable for novices, and the loquacity 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* [*Brady, 1968*]. Evans and Darley, in their 1966 survey of online debugging systems, concur, explaining that when designing the interface to a breakpoint-based debugger: *Here, as in other aspects of on-line work, convenience is critical* [*Evans and Darley, 1966*]. Eisenstadt lists three principles:

- *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.* [*Eisenstadt, 1997*]

Grishman echoes the first principle, suggesting that a debugger is at its best when it is at its most widely applicable, situationally:

> *. . . to facilitate maintenance, the same program was to be useable in both batch and interactive modes. Second, to facilitate distribution, the system had to be useable without any modification to the operating system.* [*Grishman, 1970*]

Eisenstadt's last theme of granularity is touched upon by several other authors. Hamlet notes the chasm between the separated and abstracted structured program written by the programmer, and the typical view of an executable-based debugger: *What the designer has divided and conquered, the debugger sees as an overwhelming monolith* [*Hamlet, 1983*]. Scowen sees the same lack of granularity in early flowchart-based debuggers:

> *Flowcharters have the disadvantage that they present all parts of the program with the same degree of emphasis, rather like a map that shows footpaths and motorways in the same way.* [*Scowen, 1972*]

The structure of the program has been lost. Since we expect the programmer to think about debugging using the same basic thought-structures with which they think about programming, this disconnect is troubling. Evans and Darley argue that this fineness of control is possible only with online rather than batch debugging:

> *...a very selective and close control over the execution of portions of one's program and for the examination of intermediate results, together with the possibility of making on-the-spot changes based on them, as desired.* [*Evans and Darley, 1966*]

Halpern draws the distinction between high-level debuggers which use the source language and those which operate only on executables or dumps:

> *But the general principles to be observed in implementing the XRAY (as I have dubbed it) are clear: show the programmer the system, not the machine, and do so in his language, not octal or hexadecimal.* [*Halpern, 1965*]

Grishman explains the fundamental choice between source debuggers (such as a source-level interpreter) and object debuggers:

> *The most important decision in designing a debugging system is whether to process the source language directly (by adding debugging statements to a compiler, or interpreting the source text) or to work from the object code.* [*Grishman, 1970*]

Grishman also gives an argument for object debuggers operating by simulation rather than by executing the object code directly:

> *Simulation provides a far richer set of traces and checks than could a system which executes the object code; in particular, it provides a simple solution to what appears to be the most common plight of the desperate user, "What part of my program stored that?".* [*Grishman, 1970*]

Zeller imagines the attributes of the ideal debugger, including a conception of the debugging process on a particular bug as something which can be packaged up and passed around:

> *I think this debugging process is the most important factor in debugging. Such a process should be systematic; that it should encourage people to explicitly state their hypotheses, predictions, observations, and outcomes; that it should allow interrupting and resuming sessions; and that it should allow moving a debugging session over to some co-worker at any point.* [*Zeller, 2009a*]

Of course, not everyone agrees on the usefulness of having a debugger at all, preferring the scattering of print statements and ad hoc test harnesses. In some ways, this is a useful standard against which a debugger writer may measure themselves. A debugger, if it is any good, should be so obviously useful as to render such dissent idiosyncratic. Knuth, in "The Art of Computer Programming", says:

> *The most effective debugging techniques seem to be those which are designed and built into the program itself – many of today's best programmers will devote nearly half of their programs to facilitating the debugging process on the other half; the first half, which usually consists of fairly straight-forward routines, will eventually be thrown away, but the net result is a surprising gain in productivity.* [*Knuth, 1997*]

Some go further and claim that the use of debuggers as a sort of crutch is counterproductive and should be discouraged:

> *I happen to believe that not having a kernel debugger forces people to think about their problem on a different level than with a debugger. I think that without a debugger, you don't get into that mindset where you know how it behaves, and then you fix it from there. Without a debugger, you tend to think about problems another way. You want to understand things on a different level.*
>
> *It's partly "source vs binary", but it's more than that. It's not that you have to look at the sources (of course you have to – and any good debugger will make that easy). It's that you have to look at the level above sources. At the meaning of things. Without a debugger, you basically have to go the next step: understand what the program does. Not just that particular line.* [*Torvalds, 2000*]

2.1.7 Human factors

If we want to understand why debugging is hard or to look for reasons why debuggers are not as widely used as we might expect, it is important to address not only the technical factors, but the human factors which give rise to this situation. Debugging is as personal, or more so, than programming itself. Gaines (in his thesis "The Debugging of Computer Programs", which we have quoted extensively, and which is well worth reading in full) summarises: *Computer programming is a creative art, best done by individuals, and debugging is the most highly individual aspect of programming* [*Gaines, 1969*]. Zeller captures the dual nature of debugging as a human experience of highs and lows:

> *…debugging can be an enjoyable activity that shares the thrill of the hunt and chase found in a good detective novel or video game. On the other hand a protracted, unsuccessful search for a bug in your code quickly loses its charm, particularly when your boss is asking repeatedly about your (lack of) progress.* [*Zeller, 2009b*]

The debugging process begins with an understanding that something is wrong (some wrong result or behaviour), and as the debugging process continues, we hold in our minds an ever-narrowing list of possibilities for the source or location of the bug. *The programmer will at any time have some notion of the probability that any given part of the formulation or implementation of the program is in error* [*Gaines, 1969*]. Sometimes this happens quickly, because of positive information narrowing down the scope, sometimes slowly because we are just ticking one item off the list of things which might have caused the bug (but which did not). Ko et al. believe this natural testing of the programmer's theories about the source of the bug is almost universal:

> *Although the process of debugging can involve a variety of strategies, studies have shown across a range of populations that debugging is fundamentally a hypothesis-driven diagnostic activity.* [*Ko et al., 2011*]

Inexperienced debuggers still use this approach, even if they do not know it, implying that perhaps there is something fundamental or innate to it.

> *Although all followed a standard approach that can be seen as a simplified scientific method, none of them was aware of this or able to explain his approach without resorting to demonstration.* [*Siegmund et al., 2014*]

It appears that there is a wide range of talent levels for debugging, even among equally talented or experienced programmers, another piece of evidence to suggest that debugging and programming have fundamental differences as activities.

> *There is a small number of people who are unusually good at debugging. These people can walk into almost any setting and quickly offer very helpful insights, suggestions, and solutions. … Years of experience seem to have some impact on people's debugging skill level. However, there are examples of good debuggers who are relatively new to the field and very experienced engineers who are not adept debuggers* [*Regelson and Anderson, 1994*]

Some of this is undoubtedly not just technical skill but emotional skill – debugging can be a profoundly humbling experience. Several authors note that the impedance mismatch between the modular, abstracted nature of a program and the unstructured, unknown conception of the bug at the beginning of the bug-hunting process might be one cause of some of the difficulties of debugging. For example, here are Perera et al. on this topic:

> *…in debugging we aim to understand why some erroneous result was computed by a program. This goal of understanding and explaining computations and their results often runs against our desire to treat a computation as a black box that maps inputs to outputs … opening the box exposes a great deal of useful information to the programmer but also presents several implementation and user-interface challenges.* [*Perera et al., 2012*]

Ducassé sees another part of this contrast, this haziness, the question whether a behaviour is a misbehaviour at all:

> *People sometimes talk about code errors as if they were absolute and well defined concepts whereas they are, in general, only relative and ill-defined … For example, depending on the point of view, a particular behavior of a program can be considered an error or a feature. Another example is a piece of code which breaks some company's convention, it may then be considered an error in that company while being considered correct in another place.* [*Ducassé, 1993*]

It is clear that the difficulty of debugging stems not only from the intrinsic difficulty of the technical problem, but also the many wider human factors which cannot help but affect the working programmer.

2.1.8 Classifications

So much for debugging in the abstract. We cannot go much further without considering at least a little, a survey of the concrete kinds of debugging ideas, approaches, and tools which have been pervasive in the literature over the last few decades. Our principal sources will be a Ducassé's debugging survey [Ducassé and Emde, 1988], Eisenstadt's article on programmers' debugging "war stories" [Eisenstadt, 1997], and the debugging (not testing) parts of Glenford J. Meyers classic book "The Art of Software Testing" [Myers, 1979].

One crude but effective method of extracting classifications from a text in the literature is simply to list the section headings, trusting the author's innate need to classify in order to understand. Consider, for example, some of Gaines' suggestions on how we might classify bugs [Gaines, 1969]:

Point of origin in the programming process Did the bug originate when the programmer was formulating the program in their mind, or whilst the task of converting these thoughts to a concrete program took place?

Control and computation bugs Distinguishes bugs which affect only the values of variables without affecting further control flow, from those which affect both.

Bugs resulting from lack of knowledge … of operating environment *"The lack of knowledge may be the result of the programmer's forgetting something he once knew, or getting an incorrect idea about some aspect of his operating environment, as well as being caused by something he was never informed of."*

Fatal and non-fatal This distinguishes buggy programs which terminate abnormally, due to an exception or assertion or segmentation fault from those which terminate normally but with incorrect results.

The point at which a bug may be detected During automatically during compilation or loading or execution (say an assertion) of the program, or detected manually at execution by seeing a bad output.

For an overview of practical kinds of debugging, we turn to the chapter concerning debugging in Myers' book on Testing "The art of software testing" [Myers, 1979]. He lists:

Debugging by brute force The most common technique, Myers says, and split into three: a) debugging with a storage dump; b) debugging by *"scattering print statements throughout your program"*; and c) using the debugging tools of the programming environment, such as breakpoints. *"The general problem with these brute-force methods is that they ignore the process of* thinking*."*

Debugging by induction This is a hypothesis-testing method, often conducted without the use of the computer, consisting of four steps: *"Locate the Pertinent data"*, *"Organize the data"* (to allow the observing of patterns), *"Devise a hypothesis"* and *"Prove the hypothesis"* (by further testing, before jumping in to fix the problem).

Debugging by deduction By contrast to induction, this method corrals a list of possible causes of the bug, then narrows them down with reasoning. The steps given are

"Enumerate the possible causes or hypotheses" (the hypotheses *"need not be complete explanations"*), *"Use the data to eliminate possible causes"*, *"Refine the remaining hypothesis"*, and *"Prove the remaining hypothesis"* (same as the last step in Induction above).

Debugging by backtracking Effective only for small programs, or very self-contained parts of larger programs, this method works backward through the program logic from the first evidence of failure, to work out what antecedent circumstances (states of variables, for example) may or must have caused that failure.

Debugging by testing Despite the distinction between testing and debugging being drawn in much of the literature (and indeed in Myers), in this section it is explained that there are, in fact, uses for testing procedures in the wider context of the debugging process. These are 'slim' tests, *"...attempts to cover only a single condition of a few conditions in each test case"*, rather than the 'fat' tests *"...attempts to cover many conditions in a few test cases"* used for testing more generally.

In their analysis of debugging in novice computer science students, Ahmadzadeh et al. list four major categories of knowledge used by their subjects when trying to debug faulty programs, written by others, which were presented to them in the study. These are knowledge of:

1. *The intended program*
2. *The actual program*
3. *The use of debugging methods*
4. *The error itself* [*Ahmadzadeh et al., 2005*]

Ducassé et al., in their review of automated debugging systems [Ducassé and Emde, 1988], give a classification of debugging knowledge, which echoes in part the classification of Ahmadzadeh above:

1. *Knowledge of the intended program* (*program I/O, behaviour and implementation*)
2. *Knowledge of the actual program* (*program I/O, behaviour and implementation*)
3. *Understanding of the programming language*
4. *General programming expertise*
5. *Knowledge of the application domain*
6. *Knowledge of bugs*
7. *Knowledge of debugging methods*

Eisenstadt conducted a survey of programmers, in an attempt to harness the *"generally overlooked ... potential benefit of self-reports by programmers that reflect the phenomenology of debugging"*. He then uses this data to classify sources of difficulty in bugs, methods by which they were round, and their root causes. This classification encompasses much of what is alluded to by others we have already quoted. However, the material is all together

in one single study, so we look at selected parts of the classification in some detail (names of headings Eisenstadt's from [Eisenstadt, 1997], descriptions ours in precis, except when in quotation marks):

- Dimension 1 (why difficult)

  Cause/effect chasm This describes what happens when the observed misbehaviour is distant in running time or source location from the root cause of the bug.

  Tools inapplicable or hampered Bugs which refuse to show themselves when debugged, in which the bug itself erases its own cause, or when *"some configuration or memory constraints make it impractical or impossible to use the debugging tool"*.

  Faulty assumption/model Also known as *"misdirected blame"*. Some basic misunderstanding sends the programmer off on the wrong course, when in fact the bug is elsewhere.

  Spaghetti (unstructured) code *"Informants sometimes complain about 'ugly' code invariably written by 'someone else'."* This is broader than the usual use of the phrase to mean simply "code with GOTOs" – one may create incomprehensible code in any language, no matter how structured.

- Dimension 2 (how found)

  Gather data When a programmer decides *"to find out more"*, in order to locate the bug.

    Step and study Single-stepping through the execution of the program, looking at the changes in variable values.

    Wrap and profile Wrapping a suspect function in a custom piece of code to print out some data before and after each invocation of the function.

    Print and peruse Scatter print statements at various suspect points in the code to print out function arguments, data structures, or both.

    Dump and diff Core dumps or extensive logging off a successful and failing run, and then compare them, say with the Unix tool `diff`.

    Conditional break Insert breakpoints conditional on some behaviour or value, then inspect what information the debugger provides.

    Specialist profiling tool Tools such as `valgrind` which detect illegal memory accesses.

  Inspeculation *"A hybrid of 'inspection', 'simulation', and 'speculation'."* *"In other words, either they go away and think about something else for a while, or they spend a lot of time reading through code and thinking about it."* This is recognisable as the catch-all approach when nothing else is working.

  Expert recognised clichés When a programmer asks for help from a colleague, and the colleague is able to offer an immediate diagnosis, based on some sense of similar bugs seen in the past. This sense may be conscious or unconscious.

  Controlled experiments When a clear idea about the root cause of a bug has been obtained, such tests or experiments are used to confirm. Compare with Myers' remarks on testing as a part of debugging above.

- Dimension 3 (root cause categories)

  Memory In low-level software in languages without type safety, or in the case of compiler errors, overwritten memory can cause bugs which are not immediately apparent. In earlier times, before operating system support for separate text and data segments, these were even more common. Even in modern programming languages, memory used up (in particular stack space used up) can lead to such hard-to-diagnose errors.

  Vendor Compilers or toolchains or operating systems with bugs. Or, bugs in libraries or APIs provided by others.

  Design logic The algorithm worked correctly, but was misconceived. A common cause is some missing case which was not considered.

  Initialization Incorrect starting state for a program, data structure or function, rather than incorrect logic per se.

  Variable Wrong variable used, either a design logic error (see above) or a simple typing blunder (see Lexical below). But self-reports made it hard to distinguish the two, so Eisenstadt used a separate category.

  Lexical Simple typos, or wrong understanding of operator precedence and other similar faults.

  Unsolved *"Some informants never solved their problems."*

  Language Semantic ambiguity or misunderstanding. *"In one case, an informant reported he thought 256K meant 256,000."*

  Behaviour *"The end user's or programmer's subtle behaviour ... In one case, the bug was caused by an end user's mysteriously pressing several keys at once; in another case, the cause was mischievous code."*

Another interesting classification is that of Knuth, drawn from a diary of ten years of bugs found and fixed in the TEX typesetter [Knuth, 1989]. Some 867 bugs are enumerated and classified according to an ad hoc categorisation of 15 kinds. For example:

> *354 Avoid infinite loop when recovering from `$$` in restricted horizontal mode.*
> *§1138 R* [*Knuth, 1989*]

This is bug number 354, at section 1138 in the code, categorised as type R "Robustness".

When evaluating our debugger, in chapter 8, we shall return to some of these lists and classifications, as well as to the commentary in this chapter, to see if our solution has the potential to improve on the state of the art with regard to the fundamental characteristics and difficulties of debugging that we have illuminated through this look at the historical literature.

2.2 Debugging in functional programming today

We plan to build a new debugger for the functional language OCaml [Leroy et al., 2018], 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 take a tour of existing debuggers for functional languages and examine to what extent they are usable and used. In the next section, we will take a brief look at modern debuggers for imperative languages, and debuggers which use visualization (we will look at the literature in the field of software visualization later in this thesis in section 3.6.) There is, regrettably, not enough time or space to look in detail neither at debuggers for imperative languages nor at visualization, so we must specialise early.

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 often appear, for example. Such debuggers come in several flavours. Some work by extending low-level executable debuggers such as GDB [Stallman et al., 2002] or LLDB [Lachwani and Srinivasan, 2015] 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.

Almost all the literature we have thus far reviewed is concerned with the debugging of imperative programs. How does debugging for functional programs differ? One of the great claims of functional programming is that type systems – and, in particular, type inference – remove whole classes of bugs. However, it is important to separate somewhat this claim from the question whether type errors coming from the type inference engine are a form of debugging as such. Often we think of debugging as something which happens once a program can be run – a program which fails the typechecker cannot be run. Nevertheless, we can consider type inference errors a sort of debugging support. Since the executable is not produced, however, many standard debugging techniques cannot be applied.

Most functional programming languages have a Read-Eval-Print Loop, which is used not only for learning and programming-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 functional 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. We shall see that these sorts of practical concerns have an outsize influence upon the utility of a debugging tool. Penney does not consider REPLs an adequate replacement for a real debugger, however:

> *Unfortunately, this approach can be clumsy and inadequate. Suppose we are debugging a compiler, and we find that a function call* `compile program` *fails. Moreover, we suspect the problem lies in a sub-expression in the definition of* `compile`*:*
>
> `transform table expression`
>
> *We want to supply different test cases to* `transform`*. However, it could be extremely difficult to type suitable arguments in by hand: the corresponding expressions may be*

> *long and complicated and have consistency requirements that make it difficult to write correct examples.*
>
> *A tracing tool can supply the information needed much more easily. By placing a breakpoint on the definition of the* `transform` *function, we can pick out every call to this directly from the trace of a failing execution.* [*Penney, 2000*]

The problem of not being able to produce (or edit) complex data for test cases is a scenario which appears again and again in debugging and testing.

Let us now examine contemporary debugging tools in popular functional programming languages.

2.2.1 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 quote Milner 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.* [*Hall and O'Donnell, 1985*] [*Milner, 1983*]

In fact, there was a primitive multi-typed value printer in early versions of ML, though it was not robust or portable, and does not appear in Standard ML:

> *Two multi-typed functions are included as quick debugging aids. The function* `print : ty -> ty` *is an identity function, which as a side-effect prints its argument exactly as it would be printed at top-level. The printing caused by* `print(exp)` *will depend upon the type ascribed to this particular occurrence of exp; thus print is not a normal polymorphic function. The function* `makestring : ty -> string` *is similar, but instead of printing it returns as a string what print would produce on the screen. Since top-level printing is not fully specified, programs using these two functions should not be ported between implementations.* [*Harper et al., 1986*]

The history of debugging tools for Standard ML has not always followed this pattern. Wadler [Wadler, 1998] records the story of Tolmach and Appel's debugger [Tolmach and Appel, 1995], which was deeply intertwined with the compiler and runtime of SML/NJ Standard ML. As the compiler implementation evolved, the debugger fell out of step, and is no longer available. Standard ML developers *"must return to older, more manual debugging methods"* [Wadler, 1998]. 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
debug > ^CCompilation interrupted         exit the debugger

Exception- Interrupt raised

> clearIn "rev";                          disable the debugger for our function
val it = (): unit
```

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. Poly/ML also includes a tracer:

```
> trace true;                             set tracing on
val it = (): unit
> rev [1, 2, 3, 4];                       run our function
 rev [?, ?, ?, ?]
  rev [?, ?, ?]
   rev [?, ?]
    rev [?]
     rev []
     rev () = []
    rev () = [?]
   rev () = [?, ?]
  rev () = [?, ?, ?]
 rev () = [?, ?, ?, ?]
val it = [4, 3, 2, 1]: int list
```

Polymorphism again defeats it, even in our toy scenario.

2.2.2 F#

Microsoft's F# [Syme, 2012] 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 [Meijer and Miller, 2012]. 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# [F Sharp, 2016] 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 =.* [*F Sharp, 2016*]

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

2.2.3 OCaml

We surveyed OCaml users informally to ask whether they routinely use debuggers, and if not, why not. The overwhelming result was that debuggers are not widely used. The Haskell community has found the same [Marlow et al., 2007]. There was plenty of general assent:

> *For sure, a simpler and more robust way to visualise/follow the execution of a program would be a great help to debug OCaml programs.*

More interestingly, several respondents whittled this down to a theme:

> *I use tools that I am familiar with when debugging because I don't want to focus on two things (learning a new tool and tracking down/fixing a bug).*

One coined this the *lack-of-use vicious circle*:

> *When you really need a debugger, you're not willing to learn a new tool. When you're willing to learn a new tool, you don't really want to learn a debugger.*

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.

Exception backtraces An OCaml program, when appropriately compiled with debugging information, can print out a useful trace of the stack of function calls and exception raises which led to an uncaught exception reaching the top-level and thus leading to the termination of the program:

```
$ OCAMLRUNPARAM=b ./a.out
Fatal error: exception Failure("tl")
Raised at file "pervasives.ml", line 30, characters 22-33
Called from file "example.ml", line 2, characters 24-33
Called from file "example.ml", line 5, characters 2-6
```

The system is very much a best-effort service, which can be fooled by OCaml's optimizer, and results vary from architecture to architecture. In addition, it is not possible to build a custom REPL which continues on after the error and stack trace to allow further interactive interrogation of the program's state.

Debugging with print statements Inserting print statements is a popular method of informal debugging and logging across multiple languages and 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 print 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 community still recommends it (perhaps an indication of the paucity of debugging tools):

> *"In fact, for complex programs, it is likely the case that the programmer will use explicit printing to find the bugs, since this methodology allows the reduction of the trace material: only useful data are printed and special purpose formats are more suited to get the relevant information, than what can be output automatically by the generic prettyprinter used by the trace mechanism"* [*ocaml.org debugging, 2019*]

OCaml tracing 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 2;;                             invoke the function
f <-- 0
f --> <fun>
f* <-- 2
f <-- 1
f --> <fun>
```

```
f* <-- 1
f <-- 2
f --> <fun>
f* <-- 0
f* --> 2
f* --> 2
f* --> 2
- : int = 2
```

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>`, a significant obstacle to usability.

OCamldebug The OCamldebug program is supplied with OCaml. It operates only on compiled and linked bytecode executables, not on native code executables nor on source code. The program must have been compiled with debug information. In addition, one's 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;
- 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 `ocaml --version`:

```
$ ocamldebug ocaml --version
        OCaml Debugger

(ocd) run
Loading program... done.
The OCaml toplevel, version 4.06.1
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_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 [Stallman, 1981] 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. Again, though, this places a restriction on the programmer's environment if they want the best from the tool. 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."*

OCaml and GDB 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 to GDB [Shinwell, 2014] in development, which allows for limited printing out of OCaml values using the type annotation files left behind during compilation. A similar system [Le Fessant and Chambart, 2015] is available for the LLVM debugger LLDB. But they operate very much a best-effort service. The advantage, of course, is that they work on native code executables, and can be attached in situ to processes as and when required.

2.2.4 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."* [Marlow et al., 2007]. 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."* [Marlow et al., 2007], so we choose those to look at before examining the GHC debugger.

Hat The Hat debugger [Chitil et al., 2002] 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 use 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.

Hood The Hood debugger [Gill, 2000] 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 built-in `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 show how this method fits particularly well with the point-free style of functional programming, the observation point acting as a sort of 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 GHC debugger The 2007 GHC debugger [Marlow et al., 2007] 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 suitable for use on all programs, or being limited to one compiler, or requiring re-compilation of libraries, or not being able to work 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 similar observations about what we call 'accessibility' as the cornerstone of usability in our review of the literature in chapter 2.

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.

2015 Survey An email survey [Contorer, 2015] commissioned by a commercial Haskell contractor, targeting 16000 Haskell users (with 1240 replies), asked respondents to finish the following sentence, choosing from a list of words: *"Debugging and profiling: improvements in this would be..."*. Here are the results:

crucial	29%
important	30%
helpful	23%
slight help	9%
no impact	4%

(Total crucial or important 59%). A free response field was also provided. Here are a selection of responses:

> *I see this as one of the major blockers to Haskell development. Even with understanding of the language, it is sometimes very difficult to discover why programs behave in certain ways.*
>
> *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.*

> *This is of key importance. When* [the] *compiler eliminates ... most of the basic problems, the most troublesome and complicated issues with logical structure are still to be debugged away.*
>
> *Students complain about the difficulty of debugging Haskell programs* (*laziness, no printf*).
>
> *To be honest I'm a bit 'afraid' of this part of Haskell.*
>
> *The backbone of development lies in debugging. This shouldn't even be a question.*
>
> *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 pioneers of computing.

2.2.5 Lisp

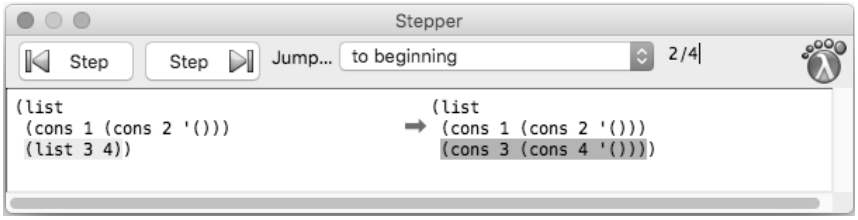
Common Lisp [Steele, 1984] has a tracing function similar to the OCaml one we looked at earlier, although there are more sophisticated facilities: the user can ask for certain values to be printed at each step, or for tracing to begin or end only when a certain predicate related to the code holds. The tracer is itself implemented as a Lisp macro.

```
[1]> (defun rev (l)
           (cond
             ((null l) '())
             (T (append (rev (cdr l)) (list (car l))))))
REV
[2]> (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 '(3 4))
4. Trace: (REV '(4))
5. Trace: (REV 'NIL)
5. Trace: REV ==> NIL
4. Trace: REV ==> (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 body of the function.

The Froglet debugger [Watt, 1994] is a modern source-level debugger for Lisp. Its author explains that, at the time it was written, Lisp had been overtaken in terms of debugging facilities by the advent of source-level debuggers for traditional imperative languages. Previously, Lisp had relatively good facilities, due to the pervasive use of S-expressions at a time when debuggers for other languages worked at the machine level. However, this very abstraction of S-expressions was in turn inferior to more modern debuggers which operated at source level.

Racket [Felleisen et al., 2015], 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 [Clements et al., 2001], 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. 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.

Some other dynamic languages, for example Smalltalk, have interesting debuggers, but these ideas do not usually translate to the kind of statically-typed batch-compiled environment which is dominant among today's general-purpose languages.

2.3 Modern debuggers for imperative programming

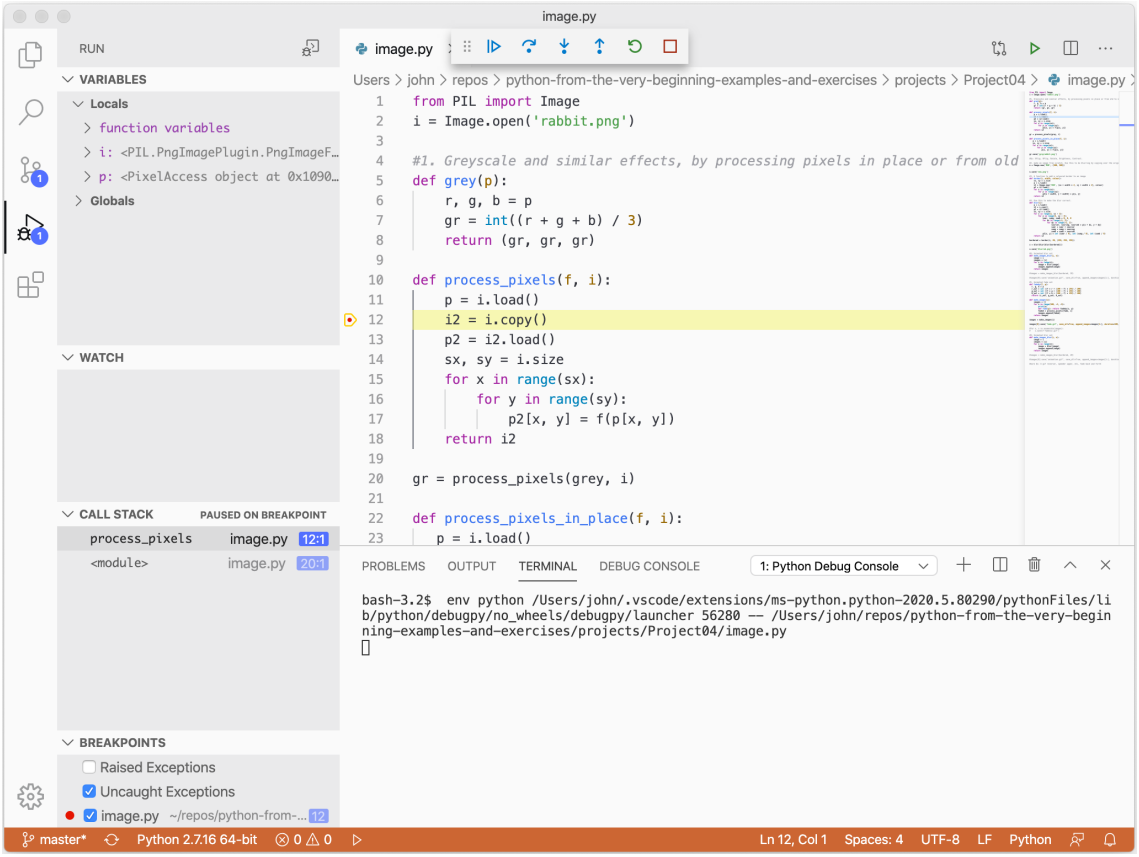
There are two areas of the debugging literature and practice which we have not thus far covered in any detail: visual interfaces to general debugging (that is to say, debugging without the command line); and specialised visual debugging for particular domains (for

example, visual debugging of concurrency issues). We address these briefly now, and explain why they do not form a main plank of our thesis.

Modern debuggers which operate in a GUI follow the pattern of DDD [Zeller et al., 2000, Zeller and Lütkehaus, 1996], a graphical interface to the ubiquitous GDB debugger. As the authors explain:

> *Besides "usual" features such as viewing source texts and breakpoints, DDD provides a graphical data display, where data structures are displayed as graphs. A simple mouse click dereferences pointers or reveals structure contents. Complex data structures can be explored incrementally and interactively, using automatic layout if preferred. Each time the program stops, the data display reflects the current variable values.* [*Zeller and Lütkehaus, 1996*]

Here is a modern example, the Microsoft Visual Code editor [Microsoft, 2020], debugging another imperative language, Python:



Debuggers for object-oriented languages, for example the HotWire debugger for C++ [Laffra and Malhotra, 1994], often have visualization methods specifically for that idiom. As the authors say:

> *What is required, are filtering techniques that enable us to define and regulate our focus while we are trying to understand why a system behaves as it apparently does. Additionally, the enormous amount of information encountered while inspecting*

> *running applications should be represented in a fashion that allows programmers to actually interpret and use it.* [*Laffra and Malhotra, 1994*]

Visual debugging in the sense of drawing charts and graphs and flow diagrams is exemplified by the JIVE debugger [Gestwicki and Jayaraman, 2004, Gestwicki and Jayaraman, 2002] and Alsallakh's tracing debugger [Alsallakh et al., 2012], both for Java. The authors of JIVE summarise:

> *JIVE provides visualizations of object structures and execution histories. The notation is generally applicable to object-oriented programs but has been customized for Java. Our methodology highlights the fact that objects are environments of program execution, with method activations nested within their proper object contexts. We provide intuitive visualizations of such Java features as objects, static contexts, inner classes, threads, and exceptions.* [*Alsallakh et al., 2012*]

We can see an interesting example of how quickly these more tightly-integrated approaches can age:

> *Programs with Swing or AWT interfaces are fully supported in JIVE; the GUIs coexist with the visualization of the program's execution.* [*Alsallakh et al., 2012*]

Neither of these GUI toolkits is in significant use today. The requirement for a debugger to "move with the times" in this fashion is avoided by DDD in its dependence only on the well established and well-supported GDB debugger. We should like to avoid it too.

Another kind of visualization, for debugging and educational purposes is touched upon in a separate literature review on interpretation in section 3.6.

2.4 Summary

We have taken lessons from the history of debugging since the beginning of the computer era, and surveyed recent systems in our area of interest, functional programming. Now, having done this, it is time to tease out the lessons from this reading, and decide how to build a debugger which might advance the state of the art. Being restricted by time and space, we have been unable to give full attention to the range of modern imperative debuggers, but reading the literature on debugging for functional languages makes a convincing case that they are essentially different and, in any event, our historical review leads us to understand that the issues at stake are fundamental enough that they should be well represented there, despite the passage of time.

We shall next decide upon the lessons to take from this literature review – both positive and negative, and delineate our scope so as to focus the project appropriately.

Chapter 3

Approach

I want everything explained to me or nothing.

— Camus, *The Myth of Sisyphus*

What can we learn from our review of the theory and practice of debugging? Having identified the attributes of a usable debugger, can we see how to apply that to the functional world, building a new debugger which represents a genuine step forward, and avoiding the mistakes of the past?

In this chapter, we develop the concept for our new debugger, justifying its overall design by reference to the literature review. We include examples of the use of our debugger as it was imagined at the time this approach was developed. We shall see later that the favoured interface to our final debugger turned out rather differently, but still in the spirit of the approach we advocate in this chapter.

3.1 Concept

We intend to tackle the problem of debugging by directly interpreting the program, showing the intermediate steps of evaluation. 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 bytecode or not. We make this clarification because the REPL is often referred to as an interpreter though it does not work by interpretation but by compilation of each entered phrase to bytecode.

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 the reduction of 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.

Thus, we are writing a kind of tracing debugger, but because of the interpretation method we shall have all information available at all times. So the traces will be unusually

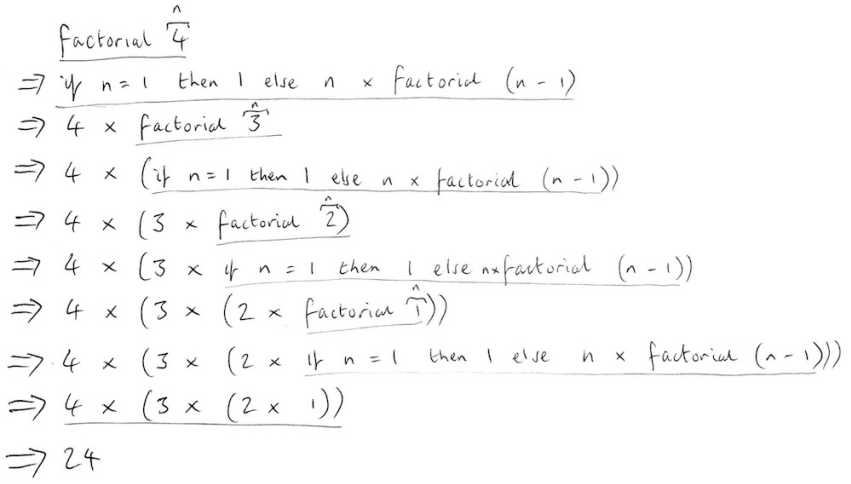


Figure 2. Handwritten diagram for the factorial function.

complete. Here is Penney describing tracing debuggers:

> *Traditionally, tracing the execution of a program means displaying an outline of the sequence of evaluation steps taking an initial program state to the final result. It is more sophisticated than just revealing what point in a program has been reached, usually highlighting the expression under evaluation and giving the user a degree of control with facilities for single-stepping and breakpoints.* [*Penney, 2000*]

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

3.2 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
```

In figure 2 we show how we might write the evaluation of this program on paper. Note that the definition of the function is not shown, and that we have used underlining and annotations for clarity. Now let us consider the same visualization produced by machine. The upper portion of figure 3 shows a naive computer-generated 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, 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 source program 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 3, showing the output of one of our prototype systems. The following changes have been made:

```
    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
=>  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 3. A naive rendering of the evaluation of `factorial 4` showing each step of the evaluation, followed by an automatically abridged 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, even if the next line is elided.

- We removed the definition of the `factorial` function itself. As 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 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.
- 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 [Taylor, 1996] and Pajera-Flores [Pareja-Flores et al., 2007]. A practical solution 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.

Figure 4 shows another example of handwritten evaluation, this time with more radical elisions. This is insertion sort, showing just the top-level `insert` and `sort` functions, not their internal workings. This kind of abstraction is perfectly natural when writing such diagrams by hand, but we shall have to carefully instruct the computer how to do it. It is not clear yet, of course, what the best interface to such a visualization system might be, in particular whether such elision may be interactively variable.

3.3 Rationale

Existing debuggers suffer, to a lesser or greater extent, from a lack of what Marlow calls *accessibility* [Marlow et al., 2007]. They provide only for a subset of the language, or require changes to be made to build environments, or do not scale well. So there is often one or more fundamental impediments to their use – they are not *accessible*. A debugger must be

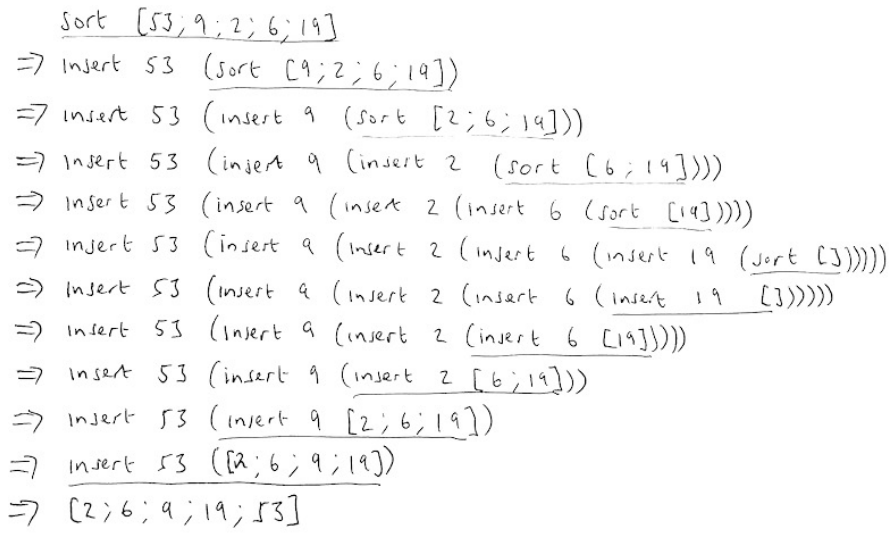


Figure 4. Handwritten diagram for insertion sort.

as accessible as a compiler. Marlow claims that the most complete Haskell trace debugger, Hat [Chitil et al., 2002], remains largely unused due to a lack of such accessibility – for example, it must be modified to support new third-party libraries. We intend, then, to bake in the correct design decisions to support widespread applicability (and thus adoption) from the beginning, even if it is at the expense of other desirable characteristics, such as speed. We aim for our system to a) be able to support the whole language by design; b) be suitable for any build environment where OCaml programs can already be built; and c) be abstracted from the compiler, and thus be robust to advances in the language and run-time environment. Thus, instead of imagining the perfect debugger, writing a toy system, and worrying about how to extend it to a practical one later, we will make design decisions based on the practicalities, and work backward from our goal. Even if our system is initially a toy in the sense that it does not support the full language, it is not a toy in terms of its integration with the language and runtime, and so extending it to the full language should be technologically straightforward (though a sizeable piece of work). Can our system be used to debug any OCaml program where source is available, even if it uses external libraries? Can our system support development of a complex system such as the OCaml compiler itself? Most importantly, of course, do people actually choose to use it?

The extremist approach

We choose to build a debugger which puts this 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 to the much more limited accessibility of various high-level

debuggers. We need to bridge this gap. So now we define precisely what trade-offs we are willing to accept.

3.4 Scope

It is important to demarcate our area of interest. Whilst we are adamant that our debugger design must be applicable in all reasonable circumstances in order to be usable and used by programmers, that does not mean that it will cover everything one may mean by the term debugging, used in its widest sense. For example, programs which statically analyse memory leaks or audit code against security threats such as timing attacks might be considered ways to identify (and therefore remove) bugs from a program, but they are not typically called debuggers. It is with the more everyday sense of the word debugger that we shall operate. Let us list some of our principal aims here. Our eventual debuggers should:

- be useable with any build system;
- work with mixed C/OCaml code;
- be able to debug libraries, not just the programmer's own code;
- be easy to keep in sync with the OCaml toolchain, so a new version can be released with each OCaml version;
- require no patches to the target toolchain – an independent development;
- be suitable for debugging the development of the OCaml compiler itself, and similar complex code.

What restrictions will we insist upon? What latitude will we give ourselves? Just like GDB, we will insist upon the executable being compiled with a special flag. In fact, the requirement is slightly stronger than GDB, since GDB is of some use on an executable not so compiled. 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, only the OCaml portion). 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.

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 (or become apparent) when switching to the interpreted version of OCaml. This might damage reproducibility when debugging certain kinds of low-level problems.

In the previous chapter, we looked at debuggers with graphical interfaces and a little at those which used visualization tools such as charting to show, for example, the memory behaviour of programs. We will stick to text-based tools for our initial implementation

for several reasons. First, it is possible to add GUI tools later, just as DDD builds upon GDB. Second, we know from our literature review that it has been widely observed that programmers prefer to think in the source language. Third, as we go on to discuss in section 3.6, the very idea of software visualization is considered dubious by some of its researchers.

We believe that this is the sensible approach for a high-level debugger. A debugger which sits within these parameters ought to embody the spirit of accessibility and so fulfil our requirements.

3.5 Correctness and maintenance

How will we know that the step-by-step interpreter is correct? Correctness is the major concern in any implementation of a programming language, but we have a special extra concern. We must make sure that the interpreter matches the semantics of the 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 will share the front end (parser, typechecker, etc.) with the OCaml bytecode and native code compilers, so we need not worry about correctness there – its behaviour should match exactly their 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 (save for a subset [Owens, 2008]) to prove adherence to and, in any event, without also formally proving the OCaml compiler, we could not show the two are semantically equivalent.

The OCaml distribution does come, however, with a large test suite, which we could use to test the interpreter. We would choose a subset of its tests removing, for example, any which depend upon code generation, and automatically test them against the expected results. Such a modification to the OCaml test suite, which has support for regression testing too, would be relatively simple and have the advantage of making a connection between the OCaml source tree and our interpreter. We will also test the interpreter manually against the corpus of simple programs in the author's textbook [Whitington, 2013]. In the future this could be automated. Another, smaller corpus of programs is included in Appendix B – this is the set we will use to test the speed of the interpreter, as described in section 5.9.

How do we know that our debugger will continue to work when the OCaml compiler is updated? Programming languages change, and new language features are added. Of course, the best solution would be to eventually have our step-by-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, may require no changes to the interpreter.

The maintenance load is composed also of the complexity of the interpreter itself. There is a marked difference in complexity between writing a simple evaluator for an abstract syntax tree and its step-by-step counterpart. Recent work on an interpreter for a subset of OCaml [Cong and Asai, 2016, Furukawa et al., 2019] suggests a scheme whereby one may *"implement a stepper concisely by writing an evaluator that is close to a standard big-step interpreter"*, using the same continuation mark techniques as Racket – so there is reason to believe that this difficulty may also one day be overcome.

3.6 Interpretation in the literature

Given the long history of debuggers and debugging which we have already explored, and our stated intention to create a debugger which works by the direct interpretation of the program source, it is legitimate to ask: "If it is such a good idea, why hasn't anyone done it before?" As part of our work to address this, we take a quick tour of some of the few naively-interpreting programming language implementations, and the use of interpretation in debugging tools past and present.

Early computers were so slow (and machine time so precious) that little consideration seems to have been given to interpreted high-level languages – the benefits of writing even a simplistic compiler were clear. The loss of debugging information which compilation embodies was an unfortunate side effect. Interpretation as part of a language was perhaps most embodied by Lisp, which provides an interesting case study in interactive debugging – a retrospective by Sandewall [Sandewall, 1978] covers this in detail. So often, though, the many idiosyncrasies of Lisp set it apart from the mainstream, and so it can be difficult to apply the same techniques to more conventional languages.

There is some interesting work by Hoffman and O'Donnel on the automatic generation of interpreters for functional programs (or mathematical expressions) whose syntactic structures are tree-like [Hoffmann and O'Donnell, 1979, Hoffmann and O'Donnell, 1982]. However, we expect that modern automatic techniques for pattern matching [Maranget, 2008] will allow these efficiencies to be automatically applied to our interpreter by the OCaml compiler itself.

Of direct interpreters in use today, there is little mainstream evidence. One interesting memory is that of a commercial product of the 1980s, the C interpreter Saber-C. The personal recollections [Ranum, 2007] of Marcus Ranum, a pioneer of network security, contain these rather tantalising passages:

> *For me, using Saber-C was an eye-opener. It gave me a whole new approach to development, since I could use the interpreter to directly call functions from a command line, without having to write a test harness with a* `main()` *routine and controlled inputs and outputs. ...without having to go through a compile/link/debug cycle, my code-creation sped up dramatically and I was catching bugs in 'real-time' as I wrote*

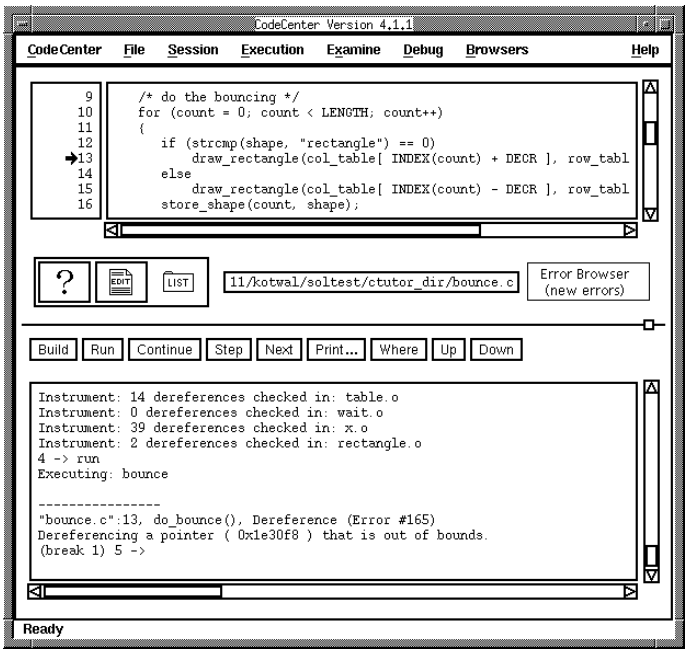


Figure 5. The CodeCenter C Interpreter.

> *each block of code. After a little while, I can safely say that the quality of my code skyrocketed. … I used to use Saber-C as my secret weapon to convince my friends I had sold my soul to the Devil: whenever they were dealing with a weird memory leak or a wild pointer that was making their programs crash with a corrupted stack or mangled free list. Usually Saber-C could pinpoint the problem in a single pass.* [*Ranum, 2007*]

This software is still apparently available under the name CodeCenter. However, our attempts to contact the company failed. The Software Preservation Group have made some resources available [Software Preservation Group, 2017] including the screenshot shown in figure 5.

There are two interesting extant systems for interpretation of C/C++. A commercial project intended for beginning students, with an emphasis on engineering and numerical methods is SoftIntegrations's CH system [Cheng, 2003, Cheng, 2010]. It is provided as a sort of 'platform', bundled with numerical and graph-plotting facilities. It does not support all modern C/C++, requires extra work to interface with any new library and is not derived from any other C compiler. The result of these things, taken together, is that it appears unsuitable for the working programmer.

Another educational system is Thetis [Freund and Roberts, 1996], designed for a first-year course at Stanford. The authors say *"On further examination, however, we made an important discovery: student frustration was less a function of the language than of the programming environment."* and *"Interactive debuggers typically require students to understand advanced concepts before they are ready to assimilate them"*. The Thetis system provides run-time error detection, a set of enhanced syntactic restrictions for beginners to prevent common confusion (e.g. `if (i = 0)` vs `if (i == 0)`), and debugging and visualization tools. It supports, however, only the subset of the language required for the course.

The Cling interpreter [Vasilev et al., 2012] is a C++ interpreter, built on Clang and LLVM, providing a REPL for interactive C++ development. Its aim is to provide a development environment for the data processing library ROOT used in the field of high-energy physics. The use of the modular structure of modern compiler components like Clang and LLVM (compared with Cling's predecessor Cint) allows Cling to keep up to date with modern language developments:

> *One can see that using Clang and LLVM as libraries helps us to build a complete interactive C++ interpreter with far less effort than building it from scratch. ... This allows us to maintain only that part of the code and delegate the rest of the maintenance to the LLVM community.* [*Vasilev et al., 2012*]

A similar REPL, for C#, is available [CSharp, 2012], based on the Mono .NET platform.

In chapter 6, we shall address the problem of speed of execution in interpreted languages by using selective interpretation. That is to say we shall have most of the program running natively, at full speed, while only the part which is being debugged will be interpreted. This has been written about before [Chase and Hood, 1987, Gough et al., 1994] though, as we shall see, a number of factors combine to make our system somewhat simpler in its implementation.

In addition to this work in practical systems, there is also plenty of interest in the field of software visualization, which seems to have its own, largely separate literature. Two useful surveys [Urquiza-Fuentes and Velázquez-Iturbide, 2004, Urquiza-Fuentes and Velázquez-Iturbide, 2009] 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 [Petre and de Quincey, 2006] gives background, and Sorva et al. give a comprehensive survey [Sorva et al., 2013] of education systems for program visualization. We pick out a few recent systems for further discussion.

The WinHIPE system [Pareja-Flores et al., 2007] is a recent incarnation of these ideas for the HOPE [Burstall et al., 1980] 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 good prettyprinted program representations. The Visual Miranda Machine [Auguston and Reinfields, 1994] 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. Touretsky describes a Lisp-based system [Touretzky, 1989] 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 [Ungar et al., 1997] 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* [Reeves et al., 1994, Goldson, 1993], showing how it pertains to various logical systems with a mathematical basis, not just functional programs. Prospero [Taylor, 1996] 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. The MLExplain system [Le Bon and Schmitt, 2018] is a recent interpreter for a subset of OCaml, based upon the JavaScript "double debugger" JSExplain [Charguéraud et al., 2018].

These systems are mostly concerned with program visualization for teaching; we wish to bias ourselves towards the task of general debugging, hoping that some educational uses will be subsumed by it. The authors of DrScheme [Finder et al., 2002], a precursor to Racket, urge caution here, choosing instead to build a 'tower' of syntactically restrictive variants of Scheme specifically for educational purposes. 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 prettyprinting. 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 – whether interactively or not. 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). A useful discussion [Patel et al., 1997] of this topic of text-vs-graphics in visualization systems for programming education shows support for our choice, amid a mixed picture. Finally on this theme, a refreshing read, in which the author tears apart the failure of his own endeavours and that of his field, in order to do better in the future, is the paper "The Paradox of Software Visualization":

> *Software visualization seems like such a logical and helpful concept with obvious benefits and advantages. But after decades of work, it has yet to be successful in any mainstream development environment.* [*Reiss, 2005*]

Some of Reiss' complaints echo those we found in our review of the debugging literature, about the ability to use these tools in any situation:

> *Some tools require extensive configuration to get a program into an environment and get it understood by the environment. ... Some require that the user work with specific languages or subsets or convert portions of the system for compatibility.* [*Reiss, 2005*]

So we can feel reasonably confident in our conservative safety-first choice of text-based 'visualization', at least for our first attempt on the problem of debugging.

3.7 Summary

We have introduced our central, rather unintuitive idea for debugging – to interpret our programs naively – based upon our previous review of the literature on debugging. We

have given a few examples, and a literature review of areas relevant to the chosen design, to add to our historical literature review in chapter 2. We have decided on an appropriate motive, rationale and scope for our first attempt at implementation, but we shall only be able to evaluate its success or lack of success later in this document. There are, of course, other choices which could have been made, and ours is certainly a bold one. This boldness increases the risk in execution, of course, but makes the prize greater.

Now it is time to build a prototype and find out if the idea can work.

Chapter 4

An interpreter for OCaml

You campaign in poetry. You govern in prose.

— Mario Cuomo

This chapter presents the technical development and practical use of our prototype OCamli interpreter for OCaml. 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 little OCaml programs. Later, we shall introduce a system which is more widely applicable.

This interpreter was written in an experimental manner, not an overtly planned one. The aim is to build from tiny programs up to larger ones, using more and more parts of the OCaml language until we can load the Standard Library and run many programs. We have not attempted to implement OCaml's object-oriented subset of features, since they do not appear in the Standard Library. We have implemented only such parts of the module language as are strictly required to load the Standard Library. But we have implemented the vast majority of the core language. As we shall learn in the next chapter, this implementation was superseded, and so did not reach the stage where formal testing for correctness, as described in section 3.5, was appropriate.

A primer on OCaml for the uninitiated can be found in appendix A.

4.1 Architecture

We should like to have a command `ocamli` so we may write:

```
$ ocamli test.ml
```

The program whose source code is `test.ml` 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, and `-show-all` to show all the stages of computation:

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

```
$ 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.

4.2 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 compiler-libs, the OCaml toolchain's library form. However, the data type is far from 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 data type (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 = 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 arguments 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 call it TinyOCaml. We shall now exhibit the bulk of the main type, skipping about a third for brevity (for example, the module system), together with some of the types to which it refers. Unlike the OCaml

parse tree data type, which is a complicated set of mutually-recursive data type definitions, we have only a few here. Let us begin with very simple enumerations for mathematical operators, comparison operators, and kinds of **`for`** loop:

```
type op = Add | Sub | Mul | Div

type cmp = LT | EQ | GT | EQLT | EQGT | NEQ

type forkind = UpTo | DownTo
```

Now, a definition for patterns in OCaml, used with **`match`**:

```
type pattern =
  PatAny                                          the pattern _ which matches anything
| PatVar of string                                                          atomic patterns
| PatInt of int
| PatBool of bool
| PatChar of char
| PatString of string
| PatUnit
| PatTuple of pattern list                                                compound patterns
| PatArray of pattern array
| PatNil                                                                    list patterns
| PatCons of pattern * pattern
⋮
```

Pattern match cases and bindings:

```
and case = pattern * t option * t                                  pattern, guard, right-hand side

and binding = pattern * t
```

We shall discuss environment items (which are used for actual evaluation) later, but we show them here for completeness:

```
and envitem =
  EnvBinding of bool * binding list ref                                                    value
| EnvType of (bool * Parsetree.type_declaration list)                                type declaration

and env = envitem list
```

And so we come to the main type:

```
type t =
  Unit                                                                          atomic types
| Int of int
| Bool of bool
| Float of float
| Char of char
| String of string
| 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
| And of (t * t)                                            boolean operator
| Or of (t * t)
| 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
| Field of (t * string)                                    access and set record field
| SetField of (t * string * t)
| 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
```

Here is what is going on inside OCamli – much simpler than directly manipulating the OCaml parse tree 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, easier to manipulate type for OCaml programs. Consider the following extract of the `of_real_ocaml` reader for converting an OCaml parse tree into its 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).

4.3 Evaluating expressions

For this proof of concept, a very simple step-by-step 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.

4.3.1 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 (redex), following the OCaml order of evaluation. 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 interpretive 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 representation of 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.

4.3.2 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? Like anything else in OCaml it is simply a kind of expression, so the natural visualization, omitting the internals of `print_int`, is this:

```
$ ocamli -e 'for y = 0 + 1 to 6 - 1 do print_int y done' -fast-for -show-all
   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                    see previous definition of forkind
```

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"))
```

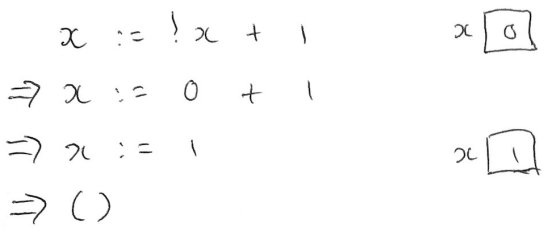
We need two copies of the body, so that one may be evaluated step-by-step, and then, when it has been reduced to a value, the spare copy can be moved into place, and we go round again. Here are all the cases needed for step-by-step evaluation of the **`for`** construct:

```
| 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 next step of evaluation. 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 possible visualization of 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. On paper, we would probably represent the reference cell in a graphical way rather than writing `{contents = ...}`:



4.3.3 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`:

```
$ ocamli -e '(fun x y -> x + y) 4 5' -show-all
    (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 such places where similar abstract syntax forms are not distinguished in the concrete syntax, 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 another option is added to the command line, the arguments will be applied at once:

```
$ ocamli -e '(fun x y -> x + y) 4 5' -show-all -fast-curry
    (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. In fact, combined with another option which pulls out let bindings to the side, we get an evaluation which is better still:

```
$ ocamli -e '(fun x y -> x + y) 4 5' -show-all -fast-curry -side-lets
    (fun x y -> x + y) 4 5
x = 4 y = 5 =>  x + y
      y = 5 =>  4 + y
            =>  4 + 5
            =>  9
```

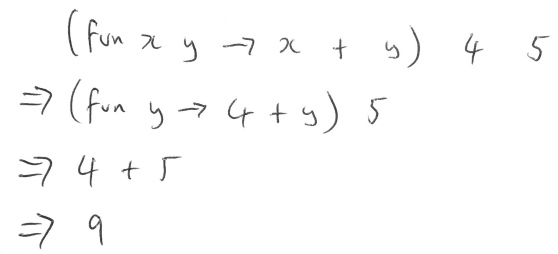
If we could go further, and do away with the step-by-step lookup of variables, we can imagine the optimal visualisation:

```
    (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. Here is our default paper visualization when teaching beginning students:



Again we see that our intuition in writing paper diagrams is to what corresponds to quite radical elision for the computer-generated diagram.

4.3.4 Exceptions

Exceptions are thought of as a sort of break in the evaluation of a program, causing evaluation to 'jump' from one place to another. And so exceptions can cause a little trouble when it comes to the visualization of a program's execution, especially if some parts of the program are elided. To exacerbate the issue, OCaml programmers often use exceptions for control flow, rather than only for truly exceptional situations. This is possible because OCaml exceptions are very fast compared with most languages. When an exception is caught, one expression (the one which raised the exception) is replaced by another (the right-hand side of the catching case), and it can seem to appear from nowhere. For some background, let us look at the paper visualization of the execution of a program which uses exceptions for control flow. Here is the example program:

```
type 'a tree =
  Lf
| Br of 'a tree * 'a * 'a tree

exception E

let rec path = function
  Lf -> raise E
| Br (l, v, r) ->
    try
      if v = 7 then [] else 1 :: path l
    with
      E -> 2 :: path r
```

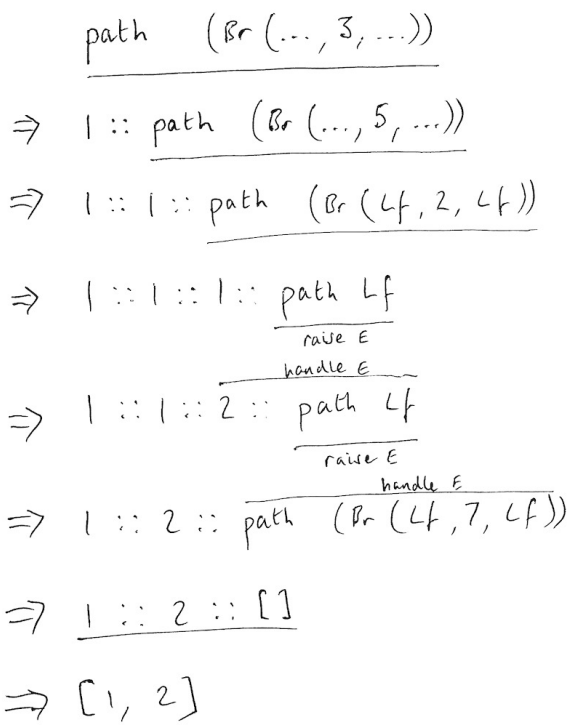
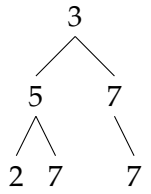


Figure 6. Handwritten diagram for a function using exceptions for control flow.

Our example binary tree will be this:



This program was adapted from a Standard ML exam question set by Lawrence C. Paulson at Cambridge. The function `path` finds the first 7 it can in a binary tree, giving a path (1 = left, 2 = right) to its position. Figure 6 shows the pen and paper visualization of its operation on the given tree. Notice that we annotate both raising and catching of exceptions here. The visualization is very stripped-down, showing almost no code, just the accumulation of the result list. This is a good example of how radical such elisions often are when we do things on paper, with our human intuition about the specific case at hand.

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 exceptions 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 in the interpreter. 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 * t option
```

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

```
| Raise of string * 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.

The problem of visualizing exceptions is discussed by Shah et al. in an interesting paper [Shah et al., 2008] showing several mechanisms for exceptions in the Java programming language.

4.3.5 Opening modules

To use a function from another module, one must either type its full path e.g. `Char.code` (function `code` in module `Char`) or use the **open** keyword to bring the name into scope. OCaml has so called local opens – writing `Char.(code 'x' + code 'y')` is the same as writing `Char.code 'x' + Char.code 'y'`. These are relatively easy to handle. We introduce a new `LocalOpen` **of** `string * t` node in the TinyOCaml data type, and upon encountering it, open the module up and bring its names into the environment at top-level, for example adding the name `hd` bound to the definition of `List.hd`:

```
$ ocamli -e 'List.(hd [1; 2; 3])' -show-all
    List.(hd [1; 2; 3])
=>  List.((function a::_ -> a ) [1; 2; 3])
=>  List.(let a = 1 in a)
=>  List.(1)
```

We could remove the local open when there are no longer any uses of its symbols, and improve the underlining:

```
    List.(hd [1; 2; 3])
=>  (function a::_ -> a ) [1; 2; 3]
=>  let a = 1 in a
=>  1
```

We cannot really use the same approach for the normal (non-local) **open** keyword, which brings all definitions of a module up to top-level. Here is an example, with four structure items (a structure item is OCaml parse tree parlance for a top-level definition in a source file):

```
open Sys

let () = Printf.printf "%s" argv.[0]
```

We are using the function `Sys.argv` by just typing `argv` after opening the `Sys` module with the **`open`** keyword. In our data type, we would have to represent this as a parenthesised structure `Open(x, ... Open(y, ...))`, which would be awkward to work with. So instead, we keep **`open`** as a structure item, dealing with it at every step. For example:

```
$ ocamli -e 'open List let x = hd [1; 2; 3]' -show-all
    open List

    let x = hd [1; 2; 3]

=>  open List

    let x = (function a::l -> a) [1; 2; 3]

=>  open List

    let x = let a = 1 in a

=>  open List

    let x = 1
```

Of course, there is no need to print it out at each step. We might prefer:

```
    open List

    let x = hd [1; 2; 3]

=>  let x = (function a::l -> a) [1; 2; 3]
=>  let x = let a = 1 in a
=>  let x = 1
```

This is shorter. However each step is no longer necessarily a valid program, since the **`open`** has been removed.

4.3.6 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:

```
$ ocamli -e 'match 1 + 2 with 4 -> 0 | 3 -> 1 + 2 | _ -> 1' -show-all
    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. We can imagine

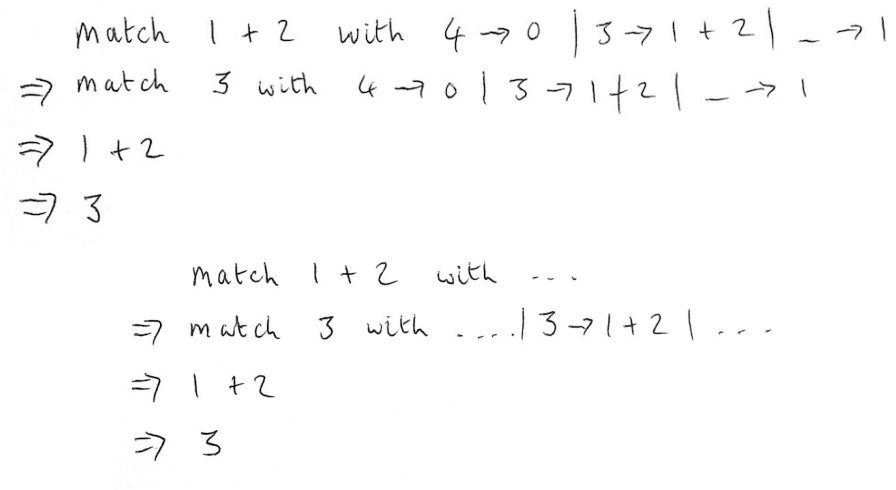


Figure 7. Two handwritten diagrams of a pattern-match

wanting to skip this process, and show just the case that matched. Figure 7 shows two possible on-paper evaluations of this program snippet. In the second case, the evaluation would have to be explained orally alongside the full snippet. This is often the case in teaching situations.

The evaluator, as one would expect, being implemented in the language it is interpreting, exhibits a certain self-similarity. For example, figure 8 shows part of the implementation of pattern-matching defined, of course, using pattern matching.

4.3.7 Functor application

We show this as an example of the more complicated work that OCamli must do at run-time. In this case, it is something which OCaml does at run-time too: applying a functor. Consider the following example, culled from the popular textbook "Real World OCaml" [Minsky et al., 2013]:

```
module type X_int = sig val x : int end                          type

module Three : X_int = struct let x = 3 end                      module

let y = Three.x                                                  test

module Increment (M : X_int) : X_int =                           define functor
  struct
    let x = M.x + 1
  end

module Four = Increment(Three)                                   apply functor

let z = Four.x                                                   test
```

Functor definitions, like type definitions, must be kept in the environment, to be looked up when a functor is applied. So, each time OCamli encounters a functor definition, it adds

```
let rec matches expr pattern rhs =
  let yes = Some rhs in
  let no = None in
    match expr, pattern with
      _, PatAny -> yes
    | x, PatConstraint (p, _) -> matches x p rhs
    | Unit, PatUnit -> yes
    | Bool b, PatBool b' when b = b' -> yes
    | Int i, PatInt i' when i = i' -> yes
    | Int32 i, PatInt32 i' when i = i' -> yes
    | Int64 i, PatInt64 i' when i = i' -> yes
    | NativeInt i, PatNativeInt i' when i = i' -> yes
    | String s, PatString s' when s = s' -> yes
    | Char c, PatChar c' when c = c' -> yes
    | Char x, PatCharRange (c, c') when x >= c && x <= c' -> yes
    | e, PatVar v -> Some (Let (false, [(PatVar v, e)], rhs))
    | Nil, PatNil -> yes
    | Cons (h, t), PatCons (ph, pt) ->
        begin match matches h ph rhs with
        | Some rhs' -> matches t pt rhs'
        | None -> no
        end
    | Tuple es, PatTuple ps ->
        match_many_binders es ps rhs
    | Record es, PatRecord (_, ps) ->
        match_many_binders (List.map (!) (List.map snd es)) (List.map snd ps) rhs
    | Array es, PatArray ps ->
        match_many_binders (Array.to_list es) (Array.to_list ps) rhs
    | e, PatAlias (a, p) ->
        matches e p (Let (false, [(PatVar a, e)], rhs))
    | e, PatOr (a, b) ->
        begin match matches e a rhs with
          Some _ -> yes
        | _ -> matches e b rhs
        end
    | Constr (_, y, None), PatConstr (x, None) when x = y -> yes
    | Constr (_, y, Some yp), PatConstr (x, Some xp)
        when x = y -> matches yp xp rhs
    | _ -> no

and match_many_binders es ps rhs =
  match es, ps with
    [], [] -> Some rhs
  | eh::et, ph::pt ->
      begin match matches eh ph rhs with
        None -> None
      | Some rhs' -> match_many_binders et pt rhs'
      end
  | _ -> None
```

Figure 8. Pattern matching by pattern matching

it to the environment, and moves on, trying to find the next redex. The functor application itself is done by what is effectively textual substitution. Here is the final state:

```
module Three : X_int =
struct
  let x = 3
end

let y = 3

module Increment =
functor (M : X_int) : X_int ->
  struct
    let x = M.x + 1
  end

module Four =
struct
  let x = 4
end

let z = 4
```

The functor application has occurred, and `z` has been fully evaluated. We could, of course, have removed the functor definition now it is no longer needed.

4.3.8 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 prettyprinter, each step may be printed out.

4.4 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 4.8. So, what remains? Three things: 1) the showing or eliding of whole steps for things like simple arithmetic and variable lookup; 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).

This topic has attracted some attention in the literature. Foubister's Spatial and temporal filters [Foubister, 1995], and Penney's 'reactive' and 'shadow' zooms and "hierarchical closing", a tree-based mechanism similar to folding in text editors [Penney, 2000]. There is, inevitably, overlap between searching and elision, especially in systems which operate on-line. For offline trace browsers, the division is more clear-cut: elision or filtering is suppressing trace output, searching is picking out pieces of the trace file to examine.

We have developed simple mechanisms for elision. There is much more to be done, but even the simplest elision mechanisms can be expected to suppress plenty of unneeded trace material, so it makes sense to conquer those first, and then consider what remains. For an 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 elisions the OCaml 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 might want to see. Similar conditions have been devised for the other kinds of elision listed in the `last_op` type. Of course, as we have mentioned, such elision may be interactive in many reasonable interfaces to such a system.

Eliding within a step

Consider the following example with multiple structure items (remember, 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.

4.5 The Standard Library

OCaml comes with a small but useful library of routines. These fall broadly into three categories:

1. 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`.
2. 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, such as the `Arg` module for handling command line arguments. When a programming language is in its infancy, the general programmer and the compiler author are one.
3. 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. For example, the function `output` which writes to standard output, or the value `Sys.word_size` which is the word size of the machine upon which the program is running.

Categories 1 and 2 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 can simply load the source code for the OCaml Standard Library before loading the main program's source. The `OCamli` interpreter also 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 the third category above which requires special treatment. Functions which are external to OCaml are introduced like this in a source file:

```
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` source 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 data type already exhibited contains the constructor `CallBuiltIn`:

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

This inclusion of a native OCaml function into the TinyOCaml 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 arguments, calls some external native function and returns a TinyOCaml 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 run-time, 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 `("%word_size", x)` in the table of primitives 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 representation of an exception by the function `exception_of_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, which we describe in section 4.9.3, 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.

OCamli ordinarily emulates the low-level primitives used to implement some of OCaml's basic language features. Recall our references example:

```
    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 ()
=>  ()
```

The `ref` function is emulated, rather than taken from the Standard Library. If we were instead to show the low-level primitives involved in the use of references, we would see a much longer visualization:

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

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

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. This sort of elision is often called "trusted code". Trusted in our sense refers not to security, but that we trust (or assume) it does not contain the bug. Bugs in standard libraries are less common than in code we are in the process of writing and debugging, and compiler bugs even less common. Our first investigation should involve suspecting our code and trusting the third-party components it uses.

4.6 An oddity: polymorphic comparison

The OCaml runtime deals with memory allocation and garbage collection, and handling signals and threads and other low-level concerns. However, there are some surprising jobs which one might expect to be handled in the language, but in fact require the connivance of the runtime. One example is polymorphic comparison, the ability to compare for equality or ordering two items of like type, where the type is not known at compile time. For example, a sorting function capable of sorting items of any type might use the polymorphic comparison operator < to order them.

OCaml's polymorphic comparison operator is implemented in C, and simply traverses the heap representations of the two items it is given, checking that their structure and contents are equal or comparing them by order. It may raise an exception if a closure is found, since functions may not be compared for equality.

We must replicate this system, for the data type which represents OCaml data as part of OCaml programs in OCaml*i*. For simple types, using the OCaml comparison operator on the data types representing the values in the interpreter works for many types, but we have not shown that this is always the case (and do not expect it to be so). Similar work is required to emulate OCaml's standard hash functions, which again require support from the runtime. We are providing, effectively, a partial alternative runtime. One thing we do not need to provide, happily, is a garbage collector: the OCaml garbage collector will happily collect the garbage of the interpreted program in just the same fashion as it will collect the interpreter's garbage. However, the space-efficiency of the running program may be different, or its garbage may be longer-lived if, for example, the storage of environments in the interpreter is inefficient.

Later, we shall see how to avoid bringing this complication into our interpreter at all.

4.7 Prettyprinting

We have not yet discussed how to print out the individual steps of evaluation of a program. Each step of evaluation is a valid OCaml program itself. The printed representation, of

course, may elide much of this for brevity, but the task is essentially the same: convert the TinyOCaml tree representing the current step of evaluation to a string. At first glance, we might decide to convert the TinyOCaml representation back into a valid OCaml parse tree, and print that using the prettyprinter provided by the OCaml toolchain. Many modern language toolchains include such functionality. The advantage of not having to write our own prettyprinter is compelling, but the straitjacket is ultimately too tight: we cannot control the line-breaking adequately, add bold or underlining, and so on. Thankfully, writing a prettyprinter for TinyOCaml is relatively easy, given the table of precedences and associativities from the OCaml language manual. We have a prototype prettyprinter which can underline the redex, use bold for keywords and highlight search results. This is enough for our experimentation purposes for now.

It is worth discussing one issue: to what extent should the prettyprinter ape the actual layout of the code in the source file? Programmers are very attached to the layout they choose for their code (although some systems have automatic formatters [Go Format, 2019] [OCaml Autoformatter, 2019]). It would reduce the impedance mismatch between source and debugger output if this formatting could be kept. On the other hand, printing each stage of the evaluation in the original format would take up many many lines, making it perhaps harder to follow, and as the expression evaluates, the original formatting would be somewhat lost anyway. So, for now, we make no attempt to follow source formatting.

The techniques the prettyprinter uses are decades old, the line breaking is aided by the OCaml Standard Library's comprehensive Format module, and the basics of printing are no real consequence to our core aims, so the implementation is not discussed in detail here.

4.8 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 [Paul and Prakash, 1994] uses an extended form of the programming language's own grammar, an approach from which we shall draw inspiration. Devanbu's GENOA [Devanbu, 1999] also reuses the language's parser in the context of source code analysis. Crew's ASTLOG [Crew et al., 1997] 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 [Griswold et al., 1996]. Specifically searching through program traces (rather than program source code) is also known in the literature, for example Watson and Salzman's work on the offline processing of traces of the evaluation

of lazy functional programs [Watson and Salzman, 1997] which allows not just single-stepping but the ability to search for given variables, successful pattern matches or return values.

Taylor distinguishes [Taylor, 1996] between filtering (preventing information appearing at all) and searching (moving through an evaluation to discover things). What we shall be calling searching is roughly what he calls filtering. He provides temporal filters, allowing information to be displayed only when a starting condition is met, and to be suppressed when an ending condition is met. We shall incorporate a similar facility: it allows the removal of swathes of information from a trace at a single stroke – the remaining trace can then be reduced using ordinary searching mechanisms.

Penney applies the well-known concepts of temporal and spatial immediacy (sometimes called temporal and spatial locality-of-reference) to the interface of an interactive tracing debugger. He says *"Temporal immediacy implies that a user should expect to find certain core facilities that support efficient reasoning by means of conceptual step sizes that bring the user quickly to the points of interest in the trace."* and *"Spatial immediacy concerns the manner of presentation. Items that are conceptually linked should clearly be seen to be linked, perhaps by placing them near one another."* [Penney, 2000]. He links this to Eisenstadt's diagnosis, which we have already discussed, that the hardest bugs to fix are those which *"were made hard because of a large temporal or spatial distance between their root cause and observable effect"* [Eisenstadt, 1997].

We have already discussed the elision of information in our interpreter. Here we provide a searching mechanism which learns the basic lessons of earlier systems. It is no grand design, but for experimentation. Until we know the final form of the interface to our debugger, it is unwise to commit too deeply. If we are to provide simple 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 three lines up to each result line*

For example, consider the evaluation:

```
$ ocamli -e 'List.map (fun x -> x + 1) [1; 2; 3]' -search '4::' -remove-rec-all
=>  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 for only those steps of evaluation which contain lists of length three with `1` as the first element:

```
$ ocamli -e 'List.map (fun x -> x + 1) [1; 2; 3]' -search '[1; _; _]' -remove-rec-all
    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]
=>  (function a::l -> let f x = x + 1 in let r = f a in r::map f l ) [1; 2; 3]
```

The search results may be highlighted with `-highlight`:

```
$ ocamli -e 'List.map (fun x -> x + 1) [1; 2; 3]' -search '[1; _; _]'
  -highlight -remove-rec-all
    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]
=>  (function a::l -> let f x = x + 1 in let r = f a in r::map f l ) [1; 2; 3]
```

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:

```
$ ocamli -e 'List.map (fun x -> x + 1) [1; 2; 3]' -after '3 + 1' -until '2::3::4'
  -remove-rec-all
=>  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
      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 ) []
```

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 have said that an essential element of the notion of accessibility is that our interpreter should be usable with any project: no matter the build system, no matter what external libraries it uses. We now have an OCaml interpreter, OCamli. Assuming that OCamli were to be extended to support the full language, such that it can run any pure OCaml code, is that good enough? No. We still cannot deal with OCaml code which calls into C, we still cannot deal with arbitrary build systems (what happens if a build system has a preprocessor for some of its code?), in short we cannot cope with any project which is much more than a directory of plain `.ml` source files. A programmer will, in such circumstances, not turn to OCamli for debugging anything other than toy programs.

We shall need to interface with code written in C through the OCaml Foreign Function Interface (FFI).

4.9 From interpreted to native and back again

Most language toolchains have a *Foreign Function Interface* (FFI) allowing interfacing with C and thence to any other language. This involves calling functions in C from our language, calling functions in our language from C, and doing any work required to convert data types. Sometimes C is the host language, with the other language embedded, sometimes the other language controls the compilation and links in the C parts. The result of the compilation might be an executable or a static or dynamic shared library.

Let us look in turn at three parts of the OCaml toolchain which we will need to understand building this new, more accessible interpreter: the FFI itself, the memory model, and OCaml's built-in preprocessing mechanism PPX.

4.9.1 The OCaml/C FFI

The OCaml/C FFI is somewhat complicated, and difficult to write correct bindings for, though there is now an automatic system [Yallop et al., 2017]. Happily, our job is only to use bindings which have already been written – they are part of any C code linked into an OCaml program, including the C parts of the Standard Library.

Let us look at a very simple example. Here, in a `.ml` file, we use the **`external`** keyword to define a function `f` of type **int** → **float** which will not be defined in the same file, but is provided elsewhere and will exist at link-time:

```
external f : int -> float
```

The implementation in C is defined thus:

```
#include <mlvalues.h>                                    load OCaml's special macros

CAMlprim value f(value x)                        the function takes and returns an OCaml value
{
  CAMLparam(x);                                                  OCaml macros to mark values
  CAMLvalue(result);
  result = Val_float(ceil(foi Int_val(x)));    OCaml value to integer, process, back to OCaml as float
  CAMLreturn(result);                                          another macro to return the result
}
```

The `value` type is used in C to describe OCaml values as they are stored in the OCaml heap. Our function in C does not really take an integer and return a floating-point value – it takes a `value` which represents an OCaml **int** and returns a `value` which represents an OCaml **float**. The `CAMLprim`, `CAMLparam`, `CAMLvalue` and `CAMLreturn` macros take care of the other requirements of the FFI, such as maintaining the fidelity of the heap (and thus allowing garbage collection, which involves walking the heap).

What of the other direction – how can we call back from C into OCaml? This is done by registering a function in the OCaml part at run-time using the Standard Library function `Callback.register`:

```
let g x = floor x                                    g has type float → int

let () = Callback.register "g" g
```

It will be the programmer's responsibility to make sure the types match and the whole program remains type-safe. Now, we look up the function from the C side, at run-time also, and call the function:

```
double c_g (int x)
{
  double d = Double_val(caml_callback(*caml_named_value("g"), Val_int(x)));
  return d;
}
```

Again, we need to ensure the rules of the FFI (especially with regard to the `value` type) are respected. The OCaml/C FFI is, of course, more complicated than these simple examples show.

4.9.2 Modelling OCaml heap values

We shall have values in our interpreter (of type `Tinyocaml.t`) such as `Int 5` or `Cons(Int 5, Cons(Int 6, Nil))`. When we call into a C function we shall need to make sure these values are in the OCaml heap, and represented in the correct fashion. Conversely, when reading the results of such a call, we will have OCaml heap values which we need to read back into the interpreted world.

OCaml, at run-time, has an untyped but tagged representation of data, providing just enough information to allow the garbage collector to traverse the heap, finding what is no longer used, and moving memory around from the minor heap to the major heap and to compact the heap. To give some examples:

- An integer is represented unboxed, but shifted left one bit, with a tag bit of 1 in the lowest bit. This distinguishes such immediate integers from boxed representations, since there can never be a 1 in the lowest bit of a memory address.
- The booleans `false` and `true` are represented in the same way (like the integers 0 and 1 would be) so they are indistinguishable from integers. But, again, the garbage collector and runtime do not need to know the type.
- The tuple $(1, 3, (4, 5))$ is represented by a heap block with tag 0, followed by two immediate integers representing 1 and 3, and a pointer to another block, tagged 0, with immediate integers 4 and 5.
- Floating-point numbers are stored boxed, but there is a special case for arrays of floating-point numbers, which are stored in one block without further boxing for each array element.
- Strings are stored as a valid C string in a block.

The following data type can be used to represent such OCaml heap values:

```
type untyped_ocaml_value =
  UInt of int
| UBlock of int * untyped_ocaml_value array
| UString of string
| UFloat of float
| UFloatArray of float array
```

User-defined data types are also represented in the same way on the OCaml heap. Consider, for example, the type:

```
type colour =
  Red
| Green
| Blue
| RGB of int * int * int
| Transparent
```

Nullary constructors are represented as integers 0,1…and non-nullary ones as blocks tagged 0,1… So, in our example, we have:

```
type colour =
  Red                                   integer 0
| Green                                 integer 1
| Blue                                  integer 2
| RGB of int * int * int                block with tag 0
| Transparent                           integer 3
```

We shall retain these tag numbers when reading the program into a TinyOCaml one. We require a function to convert any TinyOCaml value to an OCaml heap value:

```
external to_ocaml_value : t -> 'a = "to_ocaml_value"
```

The implementation must be in C, since we cannot construct the heap value using OCaml operations, but only using the C macros provided by the OCaml FFI. This is why the output type of `to_ocaml_value` is the polymorphic `'a`.

The type definitions of any user-defined data types are required for this function to work, so that it knows the tag numbers for each constructor. We read these upon initial conversion from the OCaml parse tree to the TinyOCaml representation and store the tag with each data constructor. Other than this complication, this function, at least for immutable values, is straightforward.

The inverse operation has two parts: first we want to read the heap value into the OCaml data type we created for untyped heap values, then we need to convert that to a TinyOCaml representation. The first part is simple, and again must be written in C:

```
external untyped_of_ocaml_value :
  'a -> untyped_ocaml_value = "untyped_of_ocaml_value"
```

We need to convert, say, the heap value `UInt 0` to the interpreter value `Int 0` if we expect an integer, but to the interpreter value `Bool false` if we expect a boolean. Thus, we need the expected type available if we are to do the conversion. So we provide a function which takes the type and the value of type `Tinyocaml.untyped_ocaml_value` and yields a Tinyocaml result.

4.9.3 An introduction to OCaml PPX: ppx_auto

We said in section 4.5 that, to generate a binding to a C function, we can now write:

```
[%%auto external word_size : unit -> int = "%word_size"]
```

This will be converted automatically by our preprocessor ppx_auto into:

```
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), []))
```

Such a system minimises the size of the part of OCamli which is special, that is to say the part which must be updated carefully each time the OCaml version number is increased.

The conversion from the first to the second piece of code above is done by the use of what is called a PPX (PreProcessor eXtension) – this is a preprocessing mechanism built directly into the OCaml toolchain. It operates not source-text-to-source-text but AST-to-AST. This occurs after parsing but prior to typechecking. Thus, only syntactically valid programs can arrive at and leave the preprocessor, and the result will be typechecked as usual. The PPX invocation is denoted by a so-called extension node `[%%<name> <contents>]` where the contents in our case is the **`external`** declaration. To compile this, we just add `-ppx <ppx name>` to the compiler command line, or use whatever facility our favourite build system provides. The AST will be passed through the PPX processor, which recognises the name `auto` and processes only that part of the AST, returning the rest unchanged. Thus, one can have several PPX processors operating in turn on a given file. The only requirement is that, at the end of all of them, there should be no `[%% ]` extension nodes left.

Let us look at the example above in detail. What information does `ppx_auto` have? It knows the name of the function provided (`word_size`), the name of the symbol which is expected to be available at link-time (`%word_size`), and the types of the input to the function (**`unit`**) and the output from the function (**`int`**). Our PPX processor builds a copy of the external declaration – it is still needed – and a **`string`** × `Tinyocaml.t` pair consisting of the symbol name and a generated `CallBuiltIn` to actually call the **`external`**. This `CallBuiltIn` knows to take the argument `Unit`, call the function, check for an exception (and process it if need be) and construct the `Int` return value.

So now we can generate bindings to C functions for our interpreter automatically, so long as a binding for native OCaml exists.

4.10 Summary

We have exhibited a prototype interpreter for OCaml, showing that it is plausible. The interpreter can load the OCaml Standard Library (with the exception of the complicated `Printf` and `Scanf` modules), and we have manually verified that it can execute all the programs in the textbook *OCaml from the Very Beginning* [Whitington, 2013]. Further suggestions on possible future testing mechanisms are given in section 3.5.

This work has succeeded in important ways: first, of course, it shows that such an interpreter can be reasonably written using the API provided by the OCaml compiler's public components. Second, we can see that the work required to support many of OCaml's complex language concepts in interpretation is much less difficult that compiling them. This leads us to glimpse a future, working tool for the first time: we have something with which to experiment in practice, rather than merely think about in theory.

However, as a debugger for the working programmer (as opposed to the student) it is less compelling, failing to meet many of our concerns about accessibility. In addition, its interface was designed ad hoc as part of the prototyping process, rather than in advance as a cohesive whole. We cannot, therefore, consider it as anything other than a partial success.

Nonetheless, building such an experimental interpreter is an essential part of the process, allowing us to discover the boulders which are in our way, before moving on to design a better interface for our debugger. The state of the implementation of this work is given in section 8.6. As we shall see in the next two chapters, lessons learned from this work lead to a better implementation.

Chapter 5

An improved interpreter

> *The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. ... Hence plan to throw one away; you will, anyhow.*
>
> — Fred Brooks *The Mythical Man Month*

As we have built OCamli we have glossed over a number of concerns about time and space usage, both in terms of algorithmic complexity, and just raw speed. We have also learned of several significant technical problems with the previous approach, which a new design may fix. We concentrated on building an interpreter which could cope with most of the language, and allow real examples to be tried, loading the Standard Library and other modules. This was the right approach at the time, but now we must put it to one side, and begin again. This time, we shall implement only a small subset of OCaml, but make sure that every other practical aspect is catered to; that is to say, this time, we shall be deep and narrow, rather than broad and shallow. This is not a repudiation of the previous approach, but a complement to it. Together, we think they lay the foundation for an eventual, full implementation of our debugging concept – an implementation which, sadly, we leave for future work.

In this chapter we enumerate the problems of both efficiency and technical design which were encountered during the work described in the previous chapter. We shall discuss each in some detail, and develop a design for a more practical interpreter, OCamli2.

5.1 Problems

Let us now review the process of interpretation as it stands in OCamli, identify the most serious technical and efficiency concerns, and then address them in turn. Some we have workable solutions for, some are discussed more briefly and left for future work. We do not wish to fall into the well-known trap of premature optimisation, but some inefficiencies really are too glaring to ignore: without fixing them, we could not have a usable tool, and without a usable tool, our ability to evaluate our work will be hampered. Here is the present process:

- A program is read and converted into an internal representation, TinyOCaml. This happens only once per program.
- The program is evaluated step by step. For each step, we must first identify the reducible expression (redex). This involves traversing the tree representing the TinyOCaml program. This tree can be deep, especially in the case of code which is not tail-recursive. In fact, it is easy to see how such a process can add a factor of $O(n)$ to the complexity of running some programs. It is presently not possible to find the redex in any other way than starting at the top of the program tree.
- As the redex is located, moving down the tree representing the program, an environment of bindings is built up. This environment is then used when evaluating the expression a single step. It is thrown away before evaluating the next step, even if the next redex is likely to have an identical environment.
- The evaluation is advanced one step. The new program tree has now been built. Some of these operations will have no efficiency concerns – adding two numbers is $O(1)$ just like in a compiled program, but some may be much more complex than the equivalent compiled form, for example functor application.
- Now that the next step is ready for printing, we decide whether to print it. These considerations are not typically computationally complex. If we decide to print it, however, we must clean it up by, for example, removing unused let bindings. This involves multiple passes over the tree, which is expensive.
- We now print the step itself. This is complicated compared with the interpretation of the actual step of computation (which might be as simple as adding two numbers). However, it is less complicated than finding the reducible expression and building up the environment. In addition, the work is largely unavoidable, and prettyprinting techniques are well understood.
- Some programs require the use of the Foreign Function Interface involving the copying of arbitrary amounts of data between the OCaml heap and the TinyOCaml program representation. We should like to avoid this, for simplicity, for correctness, as well as for performance.

Why it is important to fix these problems? They prevent our having a system which is (a) tractable in implementation, due to complexity of the previous approach; (b) usable by programmers in terms of the functionality it can support (for example, high quality elision); and (c) usable by programmers due to being fast enough. The alternative, to plough on with the original OCamli, looks doomed to slow suffocation.

Let us now look at some of these problems in more detail, and how OCamli2 approaches them.

5.2 A better data structure for programs

In writing our first interpreter, we chose a data structure for representing programs and their data which, as it turned out, was somewhat flawed. Here are some difficulties with

the TinyOCaml data structure used in OCamli:

- Being based upon the untyped OCaml parse tree, not the typed OCaml tree, types for each subexpression are not always available, complicating C/OCaml interaction and the provision of external features such as the Standard Library. Types are also required for printing, so any lack of availability of type information is a concern. This was probably the key mistake of our first implementation. [see section 4.5]
- Values (for example data) in the type were not represented in the same way as a compiled OCaml program would represent them, making easy interaction with the OCaml runtime and external functions accessed through the FFI difficult (and, we suspect, probably not even correct in all circumstances). [see section 4.9.1]
- Pattern matching was difficult when one or more let bindings were interspersed between structures one wished to match. [see section 5.8]
- Elision of information for printing was a largely ad hoc affair, with little direct support in the data structure. [see section 4.4]

Some of these problems could only have been discovered through the process of writing OCamli and some were simply errors which might have been foreseen. In any event, we now exhibit a new data type for OCamli2 which remedies them.

The first change is to represent values as integers or pointers into the heap, exactly as a compiled OCaml program would. That is to say instead of `Int 5` or `Cons(Int 1, Nil)` in OCamli, we just have the `Value` constructor to represent all values. This means that interaction between C and interpreted OCaml and between interpreted OCaml and compiled OCaml becomes much simpler. Data need no longer be copied. The two modes of execution (native and interpreted) simply live in the same executable, using the same heap. We shall see later how the existence of types in our new design allows such heap values to be printed out even though the OCaml heap does not contain types itself.

The second change is to decorate each node of the program tree with a record containing ancillary information: most importantly for our new design, the type of that node from the typed tree. We shall also see how it allows an elegant solution to the problem of pattern matching over let bindings.

Here is the main part of the new type for representing OCaml programs. Notice that a pair of mutually recursive data types is used to have a record decorate each node of the tree.

```
type t' =
  Value of Obj.t                          values in OCaml native representation
| Function of case list * env                                        functions
| Apply of t * t list                                       function application
| Var of string                                                       variables
| Cons of t * t                  cons part of list literal which is not yet a value e.g. [1 + 2; 3]
| Append of t * t                                                   append lists
| IntOp of op * t * t                                          integer operations
| FOp of op * t * t                                    floating-point operations
| Compare of cmpop * t * t                                 polymorphic comparison
```

```
| BoolOp of boolop * t * t                          boolean operations
| Let of bool * binding * t                         let bindings
| Struct of t list                                  structures
| LetDef of bool * binding                          top-level let bindings

and t =
  {typ : Types.type_expr;                           type from OCaml typed tree
   e : t';                                          expression
   lets : env;                                      let bindings with value right-hand-sides
   peek : peekinfo option;                          info for peeking support
   printas : string option}                         substitute printed representation
```

Consider the following program:

```
let x = 3 in
  let y = 1 + 2 in
    x + y
```

The representation of this program in our new type might be:

```
{typ = <OCaml compiler type representation of int>;
 e = Let (false,
       ("y", IntOp (Add,
                    {typ = <OCaml compiler type representation of int>;
                      e = Value <OCaml heap representation of 1>;
                      lets = []; peek = None; printas = None},
                    {typ = <OCaml compiler type representation of int>;
                      e = Value <OCaml heap representation of 2>;
                      lets = []; peek = None; printas = None}));
        {typ = <OCaml compiler type representation of int>;
         e = IntOp (Add,
                    {typ = <OCaml compiler type representation of int>;
                      e = Var "x"; lets = []; peek = None; printas = None},
                    {typ = <OCaml compiler type representation of int>;
                      e = Var "y"; lets = []; peek = None; printas = None});
         lets = [];
         peek = None;
         printas = None
        });
 lets = [("x", Value <OCaml heap representation of 3>)];
 peek = None;
 printas = None}
```

We have not been explicit thus far about why this new data structure fixes the problems we have identified: in the next few sections we shall explain the reasons that this particular structure improves evaluation, printing, and interaction with compiled code.

5.3 Values in the new representation

How do we convert a program represented by the type-checked tree provided by the OCaml compiler into our representation? It is not quite so straightforward as before. We

must identify items which "should be values" in our system and convert them to the OCaml compiled memory representation to build a `Value`. For example, a constant integer represented in the typed tree must be converted to a real integer in memory. Similarly, a cascade of `Cons` parse tree nodes where each element of the list should be a value, must itself be converted into a heap representation of such a list, along with its elements. Tediously, we must of course also have constructors in our type to represent the relatively rare case where a list literal should not be a value, for example `[f a; f b; f c]`. When a non-value array such as `[|1; 2 + 3|]` becomes the value `[|1; 5|]` during evaluation, it is added to the OCaml heap, and converted to a `Value`. Consider the following program:

```
([1; 2; 3], List.map (fun x -> x + 1) [1; 2 + 3])
```

It is a tuple of two items: the first is the list `[1; 2; 3]`, which is considered to be value in OCamli2. Heap constructs representing this list will be built upon loading the program, and it will be represented in OCamli2's program representation as a pointer into the heap, i.e `Value` in the type given above. The second part of the tuple is plainly not a value, being an unevaluated function application. The list `[1; 2 + 3]` is not a value, either. And so, it must be represented in OCamli2 as `Cons(Value ... IntOp(Add, Value ..., Value ...))` where the three values represent 1, 2, and 3 respectively in OCaml's native representation. Now we may proceed to evaluate one step:

```
([1; 2; 3], List.map (fun x -> x + 1) [1; 5])
```

The list is now `[1; 5]`. This is a value, and is immediately converted to a `Value` on the heap before the end of this step of interpretation. Thus the interpreter, at the next step, may assume any such value-normalisation has been done, and the program is in this normal form, just as it may do on the first step, such normalisation having been done upon loading the program. A few steps of interpretation later, we have reduced the program fully to a value:

```
([1; 2; 3], [2; 6])
```

Now, the entire tuple becomes a heap object represented by a `Value` constructor. Since the program is now a value, no more evaluation is possible, and interpretation ends.

Reading the type given above, and considering our definition of what should constitute values, one might wonder why the `Function` constructor exists at all: are not functions also values? We shall certainly have to have a way of building native OCaml functions for higher order use – for example, to evaluate `List.map (fun x -> x + 1) [1; 2; 3]` when `List.map` is a native function, we shall have to produce a native version of the function `fun x -> x + 1` for `List.map` to use. The reason is that functions in OCaml are used for pattern-matching, and we need to see those pattern matches when single-stepping interpreted code. So, in the present design, functions are only converted to values when being applied to a native function such as `List.map`. Otherwise, they are considered values only when required: for example, when deciding whether to end execution with the expression `[(fun x -> x + 1); (fun x -> x + 2)]`, it is considered a value; when deciding whether to convert it to an OCaml heap object upon reading the program, it is not.

Our previous solution for dealing with code written in C and accessed through the C/OCaml FFI is flawed. The copying back and forth of heap data between the interpreted and compiled worlds is cumbersome and inefficient. It may not even be possible for it to work in all cases, or for it to alter program behaviour. Consider the case where a memory location is written to both from the OCaml program and the C program (e.g. a shared mutable array), or where the program is operating in parallel execution. In the new scheme, all values are opaque under the `Value` constructor, and only unevaluated parts of the program are represented in the data type proper. This ensures that no conversion is needed when transferring values to and from the OCaml heap and the interpreted representation – the same values are simply shared. We are, in effect, following the OCaml memory model wherever possible, and the model of OCaml's longstanding FFI. That is to say the model OCaml uses itself to interface with its runtime, with external code, and with dynamic loading. This is the correct level of abstraction to work at, with regard to not only correctness but future compatibility: it ties us into a union with the compiler reducing or removing the need to update this part of the codebase with each successive OCaml version.

We have removed the possibility of a fragile interpreter-native interface. We know that such a hybrid system is workable since the OCamlJIT project [Meurer, 2010], which provides a JIT system for compiling parts of OCaml bytecode programs into executable code at run-time, is a working system with similar requirements.

5.4 Printing

The reader might wonder why we need types at run-time at all. After all, compiled OCaml programs have no types available. We need them because we are printing the steps of execution, and so we need to print values, and to print them we need their types – there is simply not enough information in the OCaml heap structure to reconstruct the type, it being designed for traversal by the runtime and in particular the garbage collector only. In the original OCamli, we represented data in programs in our own way, where the types were mostly clear: it is obvious how to print `Cons (Int 5, Nil)` as `[5]`, for example. In OCamli2, the data is simply `Value x` where `x` is an immediate integer or heap pointer in the standard OCaml manner. We cannot print it without knowledge of the type. This is where the type annotation attached to each node comes in. We already know how to build a structured value from a heap value given its type, as we showed in section 4.9.2.

Our starting point, we remember, is the typed tree produced by the OCaml compiler after parsing and typechecking. It might, then, be tempting to assume that every node has a defined type which can be used, as we have noted, for printing. However, polymorphism means this is not necessarily true: α **list** is no good for printing a list of items inside the evaluation of a polymorphic function unless we know what α is. We know only at run-time, when the function is called. Again, an evaluator which did not have to print the steps would not have to care what the type of the list elements is – that is the very definition of what it means for such a function to be polymorphic. If we fail to get this right, we may end up with having to print `<poly>` or `_` in lieu of such values, something we earlier

noticed as a fault of many other debugging approaches for functional programming.

When we look up a variable denoting a polymorphic function, we pull its type apart, and substitute in the known types of the arguments. If the polymorphic function is used again, at a different call site, different types will be substituted in. Consider the following simple program making use of polymorphism:

```
let hd = function h::_ -> h

let a = hd [1; 2; 3]

let b = hd [[1]; [2]; [3]]
```

The `hd` function has type α **list** $\rightarrow \alpha$ and is use here twice, once with an input of type **int list** and once with an input of type **int list list**. That is to say, with **int** and **int list** as concrete values for α.

For a user-defined type, printing a value of that type may need not just the type itself, but the type definition, for example to find constructor names. Such information must be loaded from compiler artefacts. It would be more pleasant if the OCaml typed tree could include such information directly.

The other printing innovation in OCamli2 is the use of the decorating record in our new data structure to store an annotation giving an alternative printing of the node in question. The evaluator may set the `printas` entry to a string which is printed instead of attempting to print the expression or value itself. This simple little hack should subsume a number of the complicated elision methods described in chapter 4, leading to shorter and better traces with little alteration or complication of the evaluator itself.

The evaluator sets up `printas` information in two ways. First, when a variable lookup results in a function (for example when a function is being passed to a higher-order function) the `printas` is set to simply print the name of the variable. This substitution is silent. We thus avoid printing the whole function body inline in subsequent steps of evaluation, a significant source of visual noise in the original OCamli, even on small programs. Second, on partial evaluation of a function passed to another. For example in the program `List.map (( + ) 2) [1; 2; 3]` applying the argument `2` to the addition operator is a step of evaluation, but its result is best printed as `(( + ) 2)` since that is what the reader will understand.

Presently, the evaluator never overwrites an existing `printas` instruction. As with several aspects of the experimental OCamli2 it is not yet clear what the optimal scheme will be. For this and other reasons, example programs produced by OCamli2 often have traces less than half the length of their OCamli versions, and these traces are much more readable. This is despite much less time having been spent on the implementation of OCamli2 – it is simply the result of learning the lessons of earlier design mistakes. An example is given at the end of this chapter.

5.5 Making native functions from interpreted ones

When the program `let y = 1 in List.map (fun x -> x + y) [1; 2; 3]` is encountered we have need to pass the function `fun x -> x + y` to the (native) `List.map` function. This is a problem because `fun x -> x + y` is code we wish to interpret. What is required is to convert this function to a natively-callable version which `List.map` can use but which, internally, will call the interpretive evaluator so the steps of evaluation can be shown, returning the resultant value back to the native `List.map`. Notice that such a function must also have access to the environment from the interpreter, to access the value of `y`.

Writing this `make_native` function is actually relatively simple, though we must be careful to give the correct result if the application of it is partial: the result will be another (interpreted) function which itself must be made native by a recursive call to `make_native`. Here it is:

```
let rec make_native impl_lets funexpr =          make native function given implicit lets and definition
  match funexpr with
  | {e = Function ([PatVar vname, None, rhs], fenv);              match expression and its type
     typ = {desc = Tarrow (_, alpha, beta, _)}} ->
      let fn =                                        what will become the natively-callable function
        let expr =                                                  build the function application
          {rhs with lets = impl_lets; e = Apply (funexpr, [rhs with e = Var vname])}
        in
          fun x ->
            let newlet = (false, ref [(vname, {expr with e = Value x; typ = beta})]) in
              match ((eval_full (newlet::fenv)) expr).e with
                Value ret -> ret                                    full application, return the result
              | Function (f', fenv') ->       this was a partial application, keep binding for next time
                  make_native
                    (newlet::impl_lets)
                    {expr with typ = beta; e = (Function (f', fenv'))}
              | _ -> failwith "didn't make a value"
      in
        (Obj.magic fn : Obj.t)                                                  build the heap object
  | _ -> failwith "make_native: not a function"
```

5.6 The Standard Library

Our new representation does away with almost all the complications of interacting with native functions. Recall that we had to write the following boilerplate to bridge between the interpreted and native worlds for a single function:

```
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), []))
```

We wrote a PPX extension to automate this in most circumstances, writing instead:

```
[%%auto external word_size : unit -> int = "%word_size"]
```

We had to give the whole **external** declaration, including specifying the type. Such a type may not always be available (if we are accessing pre-compiled code, for example). In OCamli2, we may write simply:

```
addfun "List.map" List.map
```

The addfun function is pure OCaml, and no PPX is needed. This works for any OCaml function of any type and, of course, for external functions too. The advantages for this design decision are first that it makes the code base much more maintainable due to the reduction in new work required when the OCaml version changes; and second, that it reduces traces in the ordinary case. Small traces which can then be iteratively explored by the user are, as we shall see in the next chapter, a good interface model.

5.7 Finding the redex and building the environment

Consider, by way of example, the following:

```
let x = 1 in
  let y = 2 + x in
    x + y
```

The next step of evaluation is this:

```
let x = 1 in
  let y = 2 + 1 in
    x + y
```

To find the reducible expression, as OCamli is presently conceived, we begin at the top of the program tree. First, we encounter the binding **let** x = 1 whose right-hand side is already a value. Thus, we need a function to test if an expression is a value. Of course, this function might have to recurse a great deal to find out if a big structured piece of data really is a value. It being so, we have the environment $\{x = 1\}$ and move inside the let binding, but retaining it so we can build the expression back up when we have evaluated one step. The next binding **let** y = 2 + x has a non-value right-hand side, so the redex is in its right-hand side, and we go inside the addition operation, to find that its right-hand side is the non-value x. We look this up in the environment, and our step is done. All the information we learned about the environment and the position of the redex is lost: we must begin at the top of the expression for the next step.

Can we do better? Since the right-hand side of the addition is now a value, we know exactly what the next reducible expression is – it is the addition `2 + 1` itself. Such situations, in which the redex is in a known place, are relatively frequent. We could return the next step, together with a continuation allowing easy calculation of the step after that. If the next redex cannot be determined easily, we default back to beginning at the top of the expression. It would be interesting to see if this mechanism can be achieved without undue complication to the code, and to measure its effect – that is to say, statistically, how many reducible expression sequences can be dealt with in this way, and how much do they speed things up?

At this point the reader might wonder whether our methods are just too juvenile, given, of course, the well-known literature on abstract machines for the evaluation of the lambda calculus such as the SECD machine [Danvy, 2004, Ager et al., 2003] and its many successors and variants: they make sure that the next redex is always known. Why do we not just modify such an abstract machine to hold enough extra information such that expressions may always be reconstructed step-by-step? In chapter 7 we will show one way this might be done (by modifying an OCaml-style bytecode), but we believe that it is important to get our interpreter and debugger design right, and working correctly, before jumping to such a scheme. Once we make the choice to use an abstract machine, it may be that various things we should like to do become technically difficult; sticking with inefficient, simple interpretation is best for now, however theoretically suboptimal it seems. We have already discussed, in section 3.5, Cong, Asai and Furkawa's mechanism [Cong and Asai, 2016, Furukawa et al., 2019] for writing a step-by-step interpreter in terms of a big-step one using a variant of Racket's continuation marks. This may provide a less disruptive way to make such an improvement.

5.8 Let bindings

One of the difficulties of the implementation of the original OCamli is that matching on the program data structure to find the redex is impeded by there being a non-zero number of let-bindings between layers of the data structure. So, we should like to change the data structure to make this easier. For example, when evaluating the following program, there are lets surrounding the function we wish to apply:

```
(let x = 1 in let y = 2 in (fun p -> p + x + y)) 3
```

The first problem is that this expression is not of the form `App (<fun>, x)` but `App (Let (Let (_, _), <fun>), x)` so is hard to pattern-match. We do not want to do lexical substitution, because there may be much bigger values than 1 and 2 and this would make the visualization awkward.

The connection to our work on the efficiency of the step-by-step interpreter is that, when a program is being executed, names disappear, and the let bindings become orphaned, the name they define being unused in the program. So the second problem is that such names must be removed efficiently when printing the program. To search through the

whole program each step to remove unused lets (as OCamli does) is plainly a source of inefficiency. Can we devise a mechanism which prevents this?

The first problem is solved by introducing a new type which wraps each expression (and all its subexpressions) in a record containing what we call "implicit lets". This is a list of bindings where the right-hand sides are values. These are ones whose right-hand sides require no further evaluation. When the program is read (from a real OCaml one), let bindings which are already values are put in the implicit lets. When the program is being evaluated, a let binding's right-hand side will be evaluated step by step. When its right-hand side is a value, it ceases to be a `Let` but is placed in the implicit lets surrounding its expression. Thus, all lets with fully-evaluated right-hand sides are always in the list of implicit lets, and all ones with unevaluated right-hand sides are explicit in the data type. Now, when we want to, say, find the redex by matching, there are no longer interleaved `Lets` in the way – the implicit ones cannot contain the redex since they are fully evaluated by definition. And we will never need to evaluate underneath an explicit `Let`, since the redex will always be in the right-hand side of the outermost such `Let`.

During evaluation, the implicit lets are appended to the environment whenever they are encountered. Similarly, during prettyprinting both the implicit lets and those still in the program text can be printed.

The second problem, as we mentioned, is how to remove an implicit let which is no longer used in the enclosed expression, so it is not printed. The current mechanism (in OCamli) traverses the tree, removing any lets whose names appear in the enclosed program text – save for shadowing. For example, consider the following partly-evaluated program:

```
let x = 5 in
  let y = 6 in
    5 + y
```

The name `x` no longer appears, but the let binding is not removed. In particular, the prettyprinter will not discover that the name is unused until the let binding has been printed. So, removal must be a separate process before the expression is printed. This is not just an issue of printed output either – a let binding which is not removed constitutes a space leak. A more efficient solution might be to keep a mutable flag with each implicit let, and have the printer set the flag for each name encountered when printing. Then, implicit lets with flags not so set are removed from the expression, and evaluation continues. This removes the unused let binding at the next stage.

5.9 Speed

We know that interpretation is slow, but how slow? And is it the computation or the printing out of steps which is slow? What is the cost of doing the interpretation step by step even if we only print out the pertinent steps? What are the costs of choices in data structure representation?

We might pause to remind ourselves why speed matters. It is for two reasons. First, in some cases a slow debugging tool might make debugging a particular program (or a

program for a particular input) simply intractable. Second, in the case that it is tractable, it makes debugging simply more pleasant. As we saw in our literature review in chapter 2, human factors like this can be critical in the uptake of a debugging tool.

First, we look at some early performance results from the beginning of the project. Then, we shall benchmark OCamli against OCaml using a series of test programs, with regard to time and space usage. The following table shows some time benchmark results for a tiny program (calculating the addition of two numbers) run using the OCaml compilers, OCamli, and various early experimental precursors to OCamli, each more naive than OCamli. The times given are relative to the time for the OCaml native code compiler. The discussion follows the table.

Letter	Time	Description
A	1	OCaml native code compiler
B	7	OCaml bytecode code compiler
C	1960	OCamli interpreter
D	2055	Substitution, not step-by-step, TinyOCaml tree
E	5754	Substitution, step-by-step, TinyOCaml tree
F	36,627	Substitution, step-by-step, OCaml parse tree
G	27,264,567	Substitution, step-by-step, OCaml parse tree, printing to `/dev/null`

Let us take these in turn. Letter A is the OCaml native code compiler, whose evaluation time is defined as unity. Letter B is the OCaml bytecode compiler, which is seven times slower on our benchmark. Then, at Letter C, our OCamli interpreter which is about 2000 times slower than native code. This is without prettyprinting the steps of execution – just interpreting the code silently, step by step. The latter four are timings taken from very early precursors to OCamli, in the initial experimentation stages of this project. The first, letter D, is not a step-by-step interpreter but reduces the expression to a value all at once. It uses the same TinyOCaml representation of programs which eventually became the basis of OCamli and a very naive model of execution which textually substitutes instead of looking things up in an environment of bindings. Compare this with letter E, which is the same but operates step by step. The slowdown is about three times. Letter F is the same, but is of an even earlier vintage when we were still using the usual OCaml parse tree as our main data structure. The cost of this is about six times. Finally, we add prettyprinting of each step. The cost is about 750 times. Overall, Letter G is more than 27,000,000 times slower than Letter A, demonstrating the cumulative effect of all the inefficiencies in A…G. Printing itself, unsurprisingly, slows the running time by about 750 times.

Time and space usage of some test programs

A small corpus of test programs has been prepared; they are listed in Appendix B. Due to the great disparity in time between native code compiler OCaml and OCamli, testing for time involves:

1. Picking input sizes (values of n below) which generate a run time not too long (so we do not wait for hours) and not too short (so the time measured can be considered accurate).

2. Testing each program three times and taking the mean.

3. Calculating the startup time for OCaml and Ocamli programs, and removing it from the measured time.

4. Dividing through to calculate a ratio showing how much slower each OCamli program is that its OCaml equivalent.

Here are the results:

Program	OCaml n	OCaml t	OCamli n	OCamli t	Ratio
`reference_swap.ml`	1,000,000,000	0.045	1000	0.986	21911111
`table.ml`	25,000,000	3.603	10000	22.471	15591
`factorial.ml`	5,000,000	0.006	4000	84.468	17597500
`factorialacc.ml`	5,000,000,000	7.793	4000	3.628	581932
`helloworld.ml`	50,000,000	0.743	5000	4.22	56796
`exception.ml`	5,000,000	0.124	50000	0.653	526
`tree.ml`	10000	0.104	1	4.171	382596

The first observation is, of course, that the ratios are enormous. In particular, for the `reference_swap.ml` and recursive `factorial.ml` programs. In the case of the program `reference_swap.ml` this is due to the simple fact that the execution of the compiled version reduces to very simple memory accesses, where the OCamli interpreted execution is creating and modifying compound data structures (those representing the references). It is worth noting also the contrast between the recursive and iterative versions of the factorial calculation: the recursive one generates a large intermediate expression which not only affects memory usage (as it would in the compiled version) but also affects speed since OCamli's step-by-step nature means finding the redex becomes quadratic.

Moving on to space usage, then, may give us further insight. In this test, we again have to remove a baseline, the space usage of the `donothing.ml` program in both OCaml and OCamli. This time, however, we can use equal values of n for both OCaml and OCamli. There is a complication: the measurement is maximum memory usage, which of course depends on how the garbage collector runs. In the future we might like to find a way of measuring the number of bytes allocated by the program, rather than just its highest memory usage. Here are the results, all space numbers in bytes:

Program	n	OCaml excess	OCamli excess
`reference_swap.ml`	1000	0	10,600,448
`table.ml`	10000	212,992	337,518,592
`factorial.ml`	4000	45,056	920,727,552
`factorialacc.ml`	4000	0	13,910,016
`helloworld.ml`	5000	65,536	50,286,592
`exception.ml`	50000	0	8,986,624
`tree.ml`	1	20,480	54,091,776

We can see that there are several programs which do not appear to allocate any memory in the compiled case (in fact, this just means that they did not need to allocate a page, they may allocate a token amount of memory). In each of these three cases (`reference_swap.ml`, `factorialacc.ml`, and `exception.ml`), OCamli needs significant extra memory. The worst offenders, however, are the recursive factorial program, and the times-table printer. In the case of the times-table printer this is to do with interpreting the insides of the Standard Library `print_string` function. As we have mentioned, a future OCamli2 need not do this: it will be able to use the function directly. We have illustrated the evaluation of `print_string` in Figure 9. In the case of the recursive factorial, the space used is for the expression itself, and for its repeated modification by the step-by-step interpreter.

Growth in time and space

We have considered the time and space requirements of programs interpreted with OCamli and OCaml. However, this misses out an important part of the story: the question whether the performance of a given program with regard to time and space scales in the same way when interpreted with OCamli rather than compiled with OCaml. Take, for example, our recursive and iterative `factorial.ml` and `factorialacc.ml` programs. When compiled with OCaml, they are both linear in time. The recursive `factorial.ml` is linear also in space, but `factorialacc.ml` is constant in space. In Figure 10, the top and middle graphs show the time growth of the two programs in OCamli: the time behaviour of `factorial.ml` becomes polynomial, due to the repeated traversing of an increasingly large expression to find the redex at each step and other similar processing. In the case of `factorialacc.ml` the linear behaviour is however retained. The bottom graph shows a comparison of the space usage of `factorial.ml` and `factorialacc.ml` in OCamli. The recursive case is clearly polynomial, compared with the linear behaviour of the same program compiled with OCaml. It remains to be seen how tractable the improvement of these space and time behaviours will be as we improve our implementation.

```
   print_string "Hello, World\n"
=> let string_length a = «%string_length x» in let unsafe_output_string a b c d = «caml_ml_output x» in
     let output_string oc s = unsafe_output_string oc s 0 (string_length s) in let s = "Hello, World!"\n" in output_string <out_channel> s
=> let string_length a = «%string_length x» in let unsafe_output_string a b c d = «caml_ml_output x» in
     let output_string oc s = unsafe_output_string oc s 0 (string_length s) in let s = "Hello, World!"\n" in output_string <out_channel> s
=> let s = "Hello, World!\n" in (let string_length a = «%string_length x» in
     let unsafe_output_string a b c d = «caml_ml_output x» in let oc = <out_channel> in fun s -> unsafe_output_string oc s 0 (string_length s)) s
=> let s = "Hello, World!\n" in (let string_length a = «%string_length x» in
     let unsafe_output_string a b c d = «caml_ml_output x» in fun s -> let oc = <out_channel> in unsafe_output_string oc s 0 (string_length s)) s
=> let s = "Hello, World!\n" in (let string_length a = «%string_length x» in
     fun s -> let unsafe_output_string a b c d = «caml_ml_output x» in let oc = <out_channel> in unsafe_output_string oc s 0 (string_length s)) s
=> let s = "Hello, World!\n" in (fun s -> let string_length a = «%string_length x» in
     let unsafe_output_string a b c d = «caml_ml_output x» in let oc = <out_channel> in unsafe_output_string oc s 0 (string_length s)) s
=> (fun s -> let string_length a = «%string_length x» in let unsafe_output_string a b c d = «caml_ml_output x» in
     let oc = <out_channel> in unsafe_output_string oc s 0 (string_length s)) "Hello, World!\n"
=> let s = "Hello, World!\n" in let string_length a = «%string_length x» in
     let unsafe_output_string a b c d = «caml_ml_output x» in let oc = <out_channel> in unsafe_output_string oc s 0 (string_length s)
=> let s = "Hello, World!\n" in let string_length a = «%string_length x» in
     let unsafe_output_string a b c d = «caml_ml_output x» in unsafe_output_string <out_channel> s 0 (string_length s)
=> let s = "Hello, World!\n" in let string_length a = «%string_length x» in (let a = <out_channel> in fun b c d -> «caml_ml_output x») s 0 (string_length s)
=> let s = "Hello, World!\n" in let string_length a = «%string_length x» in
     (fun b -> let a = <out_channel> in fun c d -> «caml_ml_output x») "Hello, World!\n" 0 (string_length s)
=> let s = "Hello, World!\n" in let string_length a = «%string_length x» in
     (let b = "Hello, World!\n" in let a = <out_channel> in fun c d -> «caml_ml_output x») 0 (string_length s)
=> let s = "Hello, World!\n" in let string_length a = «%string_length x» in
     (let c = 0 in let b = "Hello, World!\n" in let a = <out_channel> in fun d -> «caml_ml_output x») (string_length s)
=> let string_length a = «%string_length x» in
     (fun d -> let c = 0 in let b = "Hello, World!\n" in let a = <out_channel> in «caml_ml_output x») (string_length "Hello, World!\n")
=> (fun d -> let c = 0 in let b = "Hello, World!\n" in let a = <out_channel>in «caml_ml_output x») (let a = "Hello, World!\n" in «%string_length x»)
=> (fun d -> let c = 0 in let b = "Hello, World!\n" in let a = <out_channel> in «caml_ml_output x») 14
=> let d = 14 in let c = 0 in let b = "Hello, World!\n" in let a = <out_channel> in «caml_ml_output x»
Hello, World!
=> ()
```

Figure 9. A simple program to print to the screen, which results in a lengthy trace, due to OCaml's habit of interpreting inside Standard Library functions. Such intricacy is almost never needed by the user.

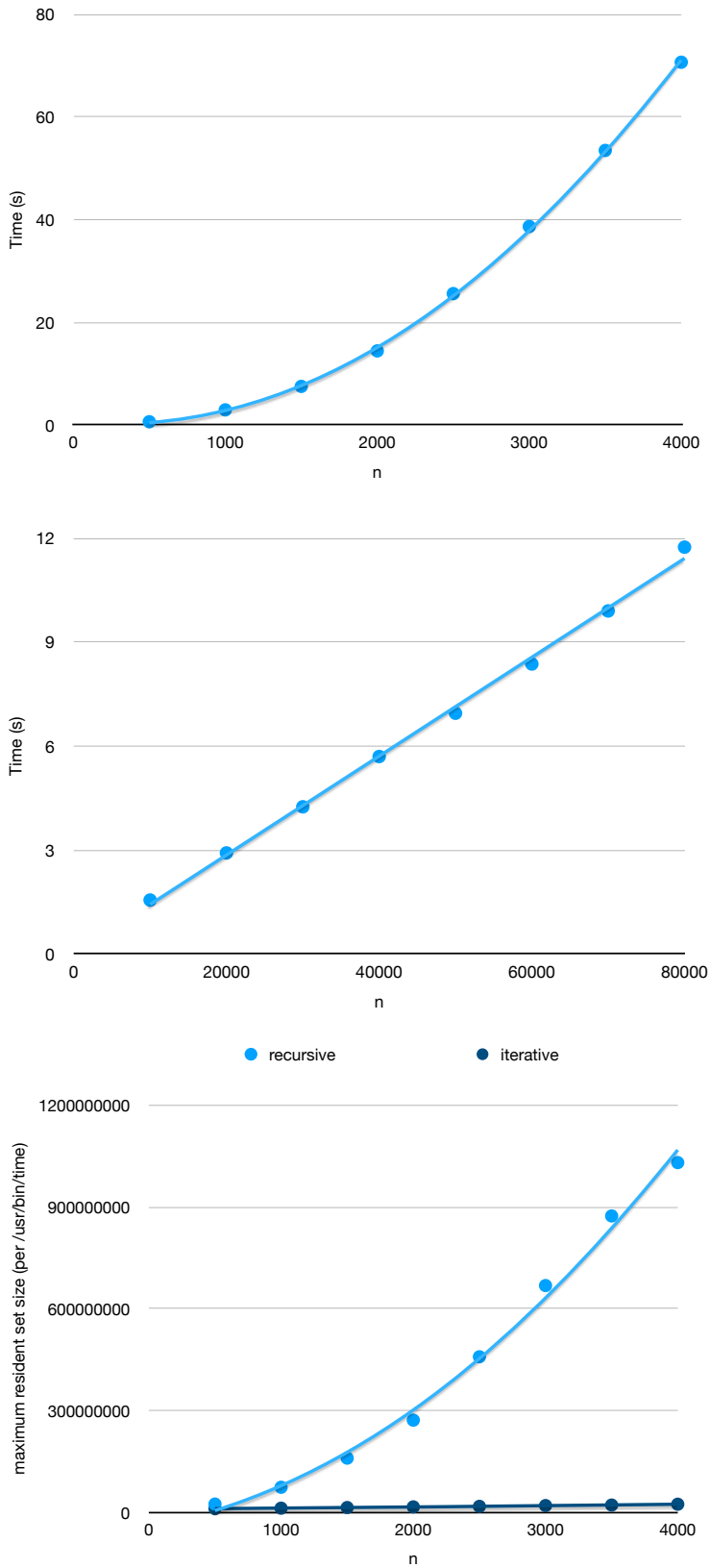


Figure 10. Time and space growth. Top: the growth of time usage in `factorial.ml`; middle: the growth of time usage in `factorialacc.ml`; bottom: space usage for the recursive `factorial.ml` and iterative `factorialacc.ml` compared.

We have mentioned that the cost of printing can be significant. There are two reasons: the job of creating the string to be printed for each step and preparing for such by eliding information; and the fact that the printed representation of the step can scale in size by a factor of $O(n)$ turning, for example, an algorithm of $O(n)$ into one of $O(n^2)$ when its steps are printed. Concretely, when testing the `factorial.ml` and `factorialacc.ml` programs, printing slows the former by about three times, and the latter by less than two times. As the interpreter becomes more efficient, we might expect this multiple to increase, since the printing will consume more of the overall time budget.

Discussion

These sorts of magnitudes are similar to those encountered elsewhere – in reviewing the Ch system [Cheng, 2003], which is a mature C interpreter (and which does not print out the steps of execution) Huber says *"In running several simple benchmark programs, code executes at about 1/1,000 the speed of compiled C code."* [Huber, 2010]. So this is perhaps the best we can hope for with pure interpretation. In fact, we might expect an OCaml interpreter to be slower due to the higher-level nature of the language.

We can consider the efficiency of a given step-by-step, prettyprinting interpreter in two fairly standard ways. Ideally, we should like an interpreter whose time and space behaviour is, modulo a constant factor, the same as that of the program compiled by the OCaml bytecode or native code compiler and executed in the usual fashion, that is when it prints none of the steps. A corollary of the space requirement is that such an 'efficient' interpreter must have the same general behaviour with regard to tail recursion (though it may cause a stack overflow earlier or later than a compiled version of the same program). In reality, some functionality of step-by-step evaluation (or, indeed the idea of step-by-step evaluation itself) may preclude this goal, and we may have to settle for some small increase in algorithmic complexity in some situations. Of course, when we start prettyprinting most or all the steps of the evaluation, time-efficiency becomes moot – the act of printing the line itself probably multiplies the complexity of the program in question by at least $O(n)$. And second, of course, we should simply like the interpreter to be as fast as possible in absolute terms.

There is an OCaml benchmarking suite in development [Sivaramakrishnan, 2020], which we would hope to modify to support OCamli, leading to a lower friction way of keeping time and space benchmarks a part of our development cycle.

5.10 Example

Many of the improvements of OCamli2 over OCamli discussed in this chapter are about making the implementation simpler and more robust, not about visualization itself. However, some of these technical decisions (such as not interpreting the Standard Library) have a bearing on the visualization itself. Consider the following output of the original OCamli on the program `List.map (function x -> x + 1) [1; 2]`:

```
$ ./ocamli -e 'List.map (fun x -> x + 1) [1; 2]' -show-all -remove-rec-all
    List.map (fun x -> x + 1) [1; 2]
```

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

Without the extra command line option `-remove-rec-all` it would be longer still. Here, in contrast, is the default output from OCamli2 on the same program.

```
$./ocamli2 -e 'List.map (fun x -> x + 1) [1; 2]'
   List.map (function x -> x + 1) [1; 2]
{entering List.map}
{entering function x -> x + 1}
=> (function x -> x + 1) x
=> (function x -> x + 1) 1
=> let x = 1 in x + 1
=> 1 + 1
=> 2
{leaving function x -> x + 1}
{entering function x -> x + 1}
=> (function x -> x + 1) x
=> (function x -> x + 1) 2
=> let x = 2 in x + 1
=> 2 + 1
=> 3
{leaving function x -> x + 1}
{leaving List.map}
=> [2; 3]
```

Much of the reduction comes from not tracing the insides of the Standard Library function `List.map`, just its callbacks to the interpreted function `function x -> x + 1`. Until we have a fuller implementation of our new, more efficient interpreter, we will not know where the balance lies between detail and conciseness in debugger output – this will require returning to our work on visualization itself.

5.11 Summary

Reflecting on our original implementation of OCamli, we have addressed a number of issues which were put aside during the rush to rapid prototyping. This further exploration gives us some confidence that a "real world" implementation of OCamli2 is tractable. The state of the implementation of this work is given in section 8.6.

In chapter 7, we pick over another kind of discard. But first, we shall discuss a practical interface for our debugger.

Chapter 6

An interface for debugging

> *The customer is never wrong.*
>
> — César Ritz

(Or, *"if people don't use your debugger, it's not their fault"*). In chapter 4 we described the OCaml PPX rewriting mechanism and used it to simplify the OCamli Standard Library. In this chapter, we suggest a possible interface for interpretive debugging based upon the same kind of PPX annotations. Then we show a failed attempt to build such a system for the original OCamli. It was the out-of-hand complexity leading to this failure which alerted us to the need to redesign from the ground up. We shall see how the new design of OCamli2 allows for a much simpler implementation of such a system, leading us to believe that our approach is now technically feasible.

The interface described in this chapter was not chosen as part of a formal process, comparing and ranking possible designs. It was simply an idea which occurred one day. And, as we shall see, it seems to be a very natural and almost beautiful concept. In the future, we would wish to consider other designs for and compare and contrast them. Let us look at it now.

6.1 Choosing what to interpret

We have already seen how annotations can be used to invoke PPX functionality in OCaml programs. What if we simply put an `[@interpret]` annotation on any part of the code we wish to have interpreted step by step, and write a PPX extension to facilitate this? All other (unannotated) code in the source file (and indeed the rest of the program) would be natively compiled as usual, running at full speed despite the annotations.

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) is motivated by our observation that a tool like OCamli on its own would not fulfil our usability or accessibility 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. Here is the debugger interface we envisage:

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 program. 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 of such debugging annotations. It will not be until we have implemented some of these (or until they have been used in anger) that we will be able to know if they are the best choices. But consider the following possibilities:

`[@interpret]` The piece of code annotated is interpreted, but functions it calls into are not. 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 argument `a` is an accumulator to make the function tail-recursive. There are two bugs in this program. First, the final case should read `pairs f (f h h'::a) (h'::t)`. Second, for the case of the single-item list, the result should be `rev a`, as for the empty list. We can add an `[@interpret]` annotation 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 = pairs ( + ) [] [1; 2; 3; 4] [@interpret]
```

Now, upon compiling the program the interpreter is embedded, and calls to `pairs` (but not the insides of `pairs`) are shown on screen. 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]
=> pairs ( + ) [3] [3; 4]
=> pairs ( + ) [7; 3] []
=> [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 as much detail as the output of OCamli we showed in chapter 4. We could annotate the body of the function `pairs` too, in order to show the step-by-step execution of the recursive parts. For now, we will just add a simpler `[@showmatch]` annotation to show which case matches each time, without interpretation:

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

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

This is a more pleasing output:

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

The bug is plain to see – the list is being reduced in size by two each time not one – so we correct it, replacing `t` with `(h'::t)` in the final case of the pattern match:

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

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

We compile the code again, with the annotations in the same places, 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 correct during evaluation, only for the output to 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 [@showmatch] with
    [] | [_] -> rev a
  | h::h'::t -> pairs f (f h h'::a) (h'::t)

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

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 may remove our annotations:

```
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. As a matter of style, we notice now, of course, the two match cases having been coalesced, their order may be reversed to reduce their number:

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

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

Now, the method can be summarised as "If there are two numbers left to process them, do so and remember the result, otherwise we can do no more, and we return the result."

`[@interpret-deep <level>]` The piece of code annotated is interpreted, and so is every function it calls, if in the same module. Recall our example program:

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

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

When using `[@interpret-deep <level>]`, calls to `rev` would also be treated as if they had an `[@interpret]` annotation attached to them, if the level is 1, and calls from `rev` to if the level is 2 and so on. For example, take a buggy definition of `rev`, using `[@interpret-deep 1]`, as follows:

```
let rec rev_inner a l =
  match l with
    [] -> []
  | h::t -> rev_inner (h::a) l

let rev = rev_inner []
```

We might see:

```
   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 [_]}
=> rev [7; 5; 3]
{return from rev}
=> []
{return from pairs}
=> []
```

This is not useful if the bug is in fact in `rev`, since `rev` immediately calls `rev_inner` which is one level deeper, so we increase the level by one and write `[@interpret-deep 2]`. We will then see this:

```
   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 [_]}
=> rev [7; 5; 3]
=> rev_inner [] [7; 5; 3]
{matches h::t}
=> rev_inner [7] [5; 3]
=> rev_inner [5; 7] [3]
=> rev_inner [3; 5; 7] []
{return from rev_inner}
=> []
{return from rev}
=> []
{return from pairs}
=> []
```

Now we can see the source of the bug. This sort of interactive deepening of the search

space for a bug allows us to begin with small manageable traces, and explore lower-level code only when required. It is especially useful in the case of well-used libraries, which are unlikely to be the source of a bug. If the trace becomes overwhelming, the annotation may be moved from the `pairs` function to the `rev` function, now that the programmer knows that it is `rev` which is at fault. We correct the code, check by running it again to yield:

```
   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 [_]}
=> rev [7; 5; 3]
=> rev_inner [] [7; 5; 3]
{matches h::t}
=> rev_inner [7] [5; 3]
=> rev_inner [5; 7] [3]
=> rev_inner [3; 5; 7] []
{return from rev_inner}
=> [3; 5; 7]
{return from rev}
=> [3; 5; 7]
{return from pairs}
=> [3; 5; 7]
```

The astute reader will notice that in this example, much of the effect is similar to simply inserting a print statement at each recursive call, and we are not really using the ability to show the fine-grained step-by-step execution of an expression. Nonetheless, the `[@interpret]` annotation does provide this functionality, as and when required.

`[@interpret-logto <filename>]` 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. For example, if we have a bug which seems to be caused by a `Not_found` exception, but the bug is not reproducible, occurring only on certain runs (for example a threaded program where the threads may run in a different order each time), we may leave a `[@interpret-logto]` annotation in place, and check the log later to find what data led to the exception being raised. Since each separate run appends rather than replaces the data, we can collect even from multiple runs of a command line tool. The logs need not be as enormous as one might think, either: the annotation mechanism reduces the trace, and a streaming compression algorithm may be used.

`[@interpret-env <variable>]` 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. This allows the shipping of

test executables which operate as quickly as one might expect, but which can, upon instruction, be used to generate debug information.

One can imagine, in fact, scenarios in which it might be sensible to leave multiple such annotations in code, each having a unique identifier, to be triggered by environment variables or command line flags. An executable with ppx_interpret embedded in it is about 4Mb larger than usual, so this sort of usage would not be suitable in all environments.

`[@interpret-sub]` Pause the program at the given point, printing the current expression and allowing the programmer to substitute their own. This can be used when we think that an intermediate result in a large program may be wrong, but do not yet know how to fix the code. We wish to stop the program, edit the data structure or program, and restart execution. Should the execution continue and produce correct output, we have evidence that the intermediate result was in fact wrong, and that we have narrowed down the source of the bug. There might, of course, be technical limitations on this substitution, for example a limitation on which symbols might be included.

`[@show-only]` Simply show a given expression, but run it natively. This has two purposes. First, for data structures, it is a way of adding a generic printer to OCaml (like Java's `toString` or Haskell's `show`). Second, it allows an extreme form of elision – we do not show any of the evaluation of the expression, just the expression itself. Now, in a recursive function, this is not so different a result as for an imperative one, but it is likely to be a good first step in debugging. For example, if we think some data is wrong somewhere, and wish to see it printed out.

`[@interpret-matching <search term>]` Give a search term (just like in OCamli) and show only those lines of the evaluation when interpreting. This allows output to be reduced from within the program, rather than having to do it by piping through another program, or in batch mode on the log file afterwards. This helps to ensure accessibility, because we may not always be able to rely upon the standard Unix tools such as `grep` being available – for example in embedded environments. In any event, our own search syntax is specialised to the task.

`[@interpret-n <n>]` / `[@exit-after <n>]` Show only `<n>` times through this code point. After that, be silent whilst continuing to run, or exit. The purpose is to produce a given, small, amount of debug output. Then, the output may be inspected manually with ease. In the case of a single debug annotation, this may not add much – we could just produce more output and inspect the beginning of it. However, when there are several annotations, it is important to make sure that a single frequently-reached annotation does not overwhelm the output. Only with experience will we be able to design this sort of functionality in detail.

`[@interpret-until <pattern>]` / `[@exit-when <pattern>]` As above, but decide when to finish interpreting or when to exit based not upon a number but upon matching the given pattern. We can use this when we already know which value indicates a bug

and are not interested in seeing output after the bug has already occurred. We wish to see only what led up to it.

`[@interpret-interactive <n>]` Upon reaching this program point, dump into an interface which acts as an interactive debugger, setting and clearing breakpoints and so on. So, we might see, in our `pairs` example:

```
   pairs ( + ) [] [1; 2; 3; 4]
?next
{matches h::h'::t}
=> pairs ( + ) [3] [2; 3; 4]
?next 5
{matches h::h'::t}
=> pairs ( + ) [5; 3] [3; 4]
{matches h::h'::t}
=> pairs ( + ) [7; 5; 3] [4]
{matches [_]}
=> rev [7; 5; 3]
=> rev_inner [] [7; 5; 3]
{matches h::t}
=> rev_inner [7] [5; 3]
?next
=> rev_inner [5; 7] [3]
?next
=> rev_inner [3; 5; 7] []
{return from rev_inner}
=> []
?exit
```

Here we step through the code interactively, then exit the process upon finding the bug. All sorts of standard debugging tools could be included here, such as breakpoints. This is an example of embedding a whole interface for debugging inside the ppx_interpret mechanism. Note that the whole debugger is part of the executable, just like the interpreter – so still no external tools are required, and the debugger remains accessible.

`[@expected <pattern>]` We expect the result of evaluating this expression to match the given pattern. If it does not, the suspect evaluation may be printed out. These "expect tests" are a common method of software testing. We could also, instead of a pattern, use a predicate to be run on the value to see if it matches. We might write the annotation `[@expected [3; 5; 7]]` into our `pairs` example, and should a bug be introduced in a future version of the code and running with debugging turned on (perhaps using the environment variable method already described), the mismatch would be reported.

`[@stoprepeat]` Stop after a duplicate expression is encountered. For detecting bugs caused by non-termination. Such bugs can cause huge amounts of output before execution can be interrupted, making it hard to see the steps leading to the non-termination. For example, we might see:

```
f [1; 2; 3]
```

```
f [1; 2]
f [1]
f [1]
f [1]
ppx_interpret: output ended on [@stoprepeat]
```

This concludes our tour of possible annotation types. 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. But the annotation-based one we have alighted upon seems promising, so we persevere with it for now.

6.2 An early failure

In this section, we discuss an attempted implementation of our scheme, using the original OCamli architecture from chapter 4. This sheds further light upon some technical difficulties which made the transition to the typed OCamli2 necessary. We shall go on to describe the successful implementation in due course.

Recall that we wish to debug a program, and think that the problem is in a certain module, or that the problem would be best identified by examining the innards of a certain module. Instead of inserting print statements, we could write the PPX annotation `[@interpret]` to show the steps of evaluation of all the code in a module as if it had been interpreted by OCamli:

```
[@interpret]

let f x = A.double x

let g y = A.double (f y) + 2

let h = g 1
```

Call this module `B`. It uses `A.double`. Suppose also that there is another module, `C`, which uses `B.f`. Then the module dependencies are `C` $\rightarrow$ `B` $\rightarrow$ `A` where $\rightarrow$ means "depends upon". We wish to be able to compile this program using whatever commands and build system we are used to, and for the result to be an ordinary executable. Then, when we run the executable, only the code in module `B` is interpreted and displayed on the screen in the manner of OCamli. The code in modules `A` and `C` is run natively and silently.

In fact, it would be better if we could interpret only exactly the expression which we are interested in. PPX annotations such as `[@interpret]` may be placed in any position which corresponds to a node of the parse tree (to which they then become attached). This is what we described in the previous section. So we might write:

```
let f x = A.double x

let g y = A.double (f y) [@interpret] + 2

let h = g 1
```

Now, only `A.double (f y)` will be interpreted, the rest of module `B` being natively executed, together with modules `A` and `C` as before. We have a very lightweight but powerful version of "adding print statements", with the added ability to show steps of execution. Of course, the annotations could be silently added by some other interface such as an IDE, rather than by the user manually typing and then removing `[@interpret]` annotations.

How is this achieved technically, making use of the tools we have already developed in OCamli for interfacing between the interpreted and native worlds? We have four tasks:

- Calling into native code from interpreted code, which happens if the part of the program annotated with `[@interpret]` calls into native code – in this instance the second call to `A.double` in our latter example.
- Calling into interpreted code from native code – for example if module `C` calls the interpreted function `B.g`.
- Calling into native code from interpreted code in the same module. For example, when `B.g` uses the value of `f y` in the second example.
- Calling into interpreted code from native code in the same module, for example when `h` calls `g` in the second example.

The job of our PPX extension ppx_interpret must be to take a parse tree representing an `.ml` file with one or more `[@interpret]` annotations, and transform it into one with exactly the same interface, which interprets code marked with `[@interpret]` and runs all other code natively. The shim produced by our system for the function `f` in our example is displayed in figure 9. The attempted implementation of this early version of ppx_interpret, however, only served to highlight further the faults of the untyped nature of OCamli, discussed in chapter 5. The interface between the native and interpreted parts of the running program is fragile, and the difficulty of the implementation, were it to be completed, would be overwhelming. Large parts of the OCaml front end would have to be re-implemented to make sure type definitions could be found, for example.

Now let us describe a feasible implementation of this annotation-based debugging interface, using our new interpreter OCamli2.

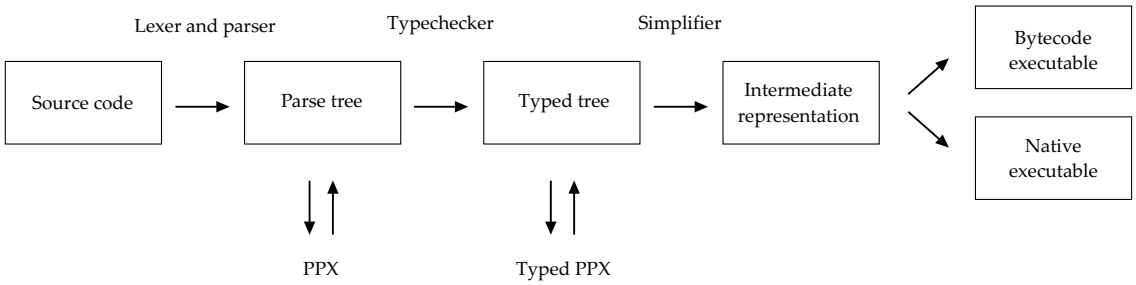
6.3 Typed ppx_interpret with OCamli2

We have written a patch to the OCaml compiler to provide a typed analog to PPX. This allows typed-tree-to-typed-tree rewriting in addition to the standard PPX mechanism of parse-tree-to-parse-tree rewriting. We did not want to have to patch the compiler, but this patch is very small and unobtrusive, and need not change between OCaml versions. It

```
let f x =
  let module A =                                          shim for calling into module A
    struct
      let double env =
        function
        | x::[] ->
            let heap_x = Tinyexternal.to_ocaml_value x in          shim for A.double
            let result = A.double heap_x in
            Tinyexternal.of_ocaml_value env result "int"
        | _ -> failwith "A.double: arity"
    end in
    let open Tinyocaml in
      let tiny_x = Tinyexternal.of_ocaml_value [] x "int" in    read argument from native world
      let (_,program) =
        Tinyocamlrw.of_string "let rec f x = A.double x in f x" in     interpreted version of f
      let env =
        [EnvBinding (false, (ref [((PatVar "x"), tiny_x)]))] @      environment for interpreter
          ([EnvBinding
              (false,
                (ref [((PatVar "A.double"), (mk "A.double" A.double))]))]
             @ [])
         in
      let tiny_result =
        Eval.eval_until_value true false (env @ (!Eval.lib)) program       interpret the code
      in
        (Tinyexternal.to_ocaml_value tiny_result : int)             result back to native world
```

Figure 11. A shim for the function `f`

may also prove worthwhile to other developers – there are many uses for a typed PPX. Here is the OCaml compiler architecture with PPX and typed PPX mechanisms in place:



Consider the following program, with an `[@interpret]` annotation:

```
let global x = x * 2

let x () =
  let y = 3 - 4 in
    (Random.int 50 + y + 5 + global 6 [@interpret]) + 7

let _ =
  Printf.printf "Result is %i\n" (x ())
```

The program will be compiled as normal, and run natively. However, the section marked by `[@interpret]` will be interpreted step by step. In this case, it is `Random.int 50 + y + 5`

`+ global 6`. Note that adding the `[@interpret]` annotation does not imply that the innards of `global` will be interpreted too. So we have a native function `x`, which natively calculates the value of `y`, and then has an interpreted section which needs that value of `y` and also uses the native Standard Library function `Random.int`. The result of `x` must be returned natively. We have both native code and interpreted code, and the interpreted code needs to be able to call native functions and use the results of natively calculated values.

It is the job of our new OCaml2-based, typed version of ppx_interpret to arrange all this. Altering the typed tree is a delicate affair, since there are so many invariants required for a valid tree, so we arrange for most of the work to be done by a plain PPX prior to typechecking. The typed PPX which operates after typechecking is limited to transporting the type of expressions from compile-time to run-time. Here is the output of the plain PPX on our example, before the typed PPX acts:

```
let env = ref Lib.stdlib                                          initial environment
let () = Tppxsupport.init ()                                      set up support code
let eval_full = Tppxsupport.eval_full env                         the interpreter

let global x = x * 2

let () = Tppxsupport.addenv env "global" global ""        make global available to the interpreter

let x () =
  let y = 3 - 4 in
    let () = Tppxsupport.addenv env "y" y "" in          make value of y available to the interpreter
      (Random.int 50 + y + 5 + global 6 [@interpret]) + 7

let () = Tppxsupport.addenv env "x" x ""                 make function x available to the interpreter

let _ =
  Printf.printf "Result is %i\n" (x ())
```

Two things have happened. First, a preamble has been added, for the use of the typed PPX, to keep it as simple as possible. Second, both local and global let bindings have been followed by calls to `Tppxsupport.addenv`, whose job is to put native values into an environment for the interpreted code to refer to. The empty string `""` will be replaced by the subsequent typed PPX with a marshalled representation of the type of the value. Marshalling is used to minimise interference in the typed tree. We need not build new nodes for the typed tree, just change one constant string for another.

Once typechecking has occurred, the typed PPX part of ppx_interpret runs. It has two jobs: to build the interpreted sections, and to fill in the types of the environment items. Here is the output:

```
let env = ref Lib.stdlib                                          initial environment
let () = Tppxsupport.init ()                                      set up support code
let eval_full = Tppxsupport.eval_full env                         the interpreter

let global x = x * 2
```

```
let () = Tppxsupport.addenv env "global" global "<marshalled type of function global>"

let x () =
  let y = 3 - 4 in
    let () = Tppxsupport.addenv env "y" y "<marshalled type of value y>" in
      eval_full env "<marshalled TinyOCaml representation of Random.int 50 + y + 5 + global 6>"
      + 7

let () = Tppxsupport.addenv env "x" x "<marshalled type of function x>"

let _ =
  Printf.printf "Result is %i\n" (x ())
```

Now, at run-time, not only are the types of the interpreted parts available (as they are in command-line OCamli2) but the types of environment elements from the natively-executed parts of the enclosing program are available too: the step-by-step interpreter has everything it needs to print out the steps of evaluation. Here is the output for a run of our example program:

```
$ ./example
   Random.int 50 + y + 5 + global 6
=> 44 + y + 5 + global 6
=> 44 + -1 + 5 + global 6
=> 43 + 5 + global 6
=> 48 + global 6
=> 48 + 12
=> 60
```

The file `example` is a native code executable, containing within it the main program, executed natively, the phrase `Random.int 50 + y + 5 + global 6` interpreted step by step, and the interpreter itself. The native code portion and the interpreted portion share the same OCaml runtime, and the same heap.

One other aspect of the implementation of this annotation-based interface to debugging is worth discussion. Can we now dispense with our special Standard Library? We should not need to load the Standard Library and interpret its module initialisation, since the interpreter is in the same process as the OCaml runtime in which the native code is running and in which the Standard Library has already been loaded and its initialisation run. So, we should be able to reduce the footprint of our changes greatly, simply by interfacing with the Standard Library as if it were any other piece of code. This means, of course, that we can no longer show the steps of interpretation inside the Standard Library (since it was not compiled with `[@interpret]`). This should not be too much of a problem, since it is unlikely that a programmer would like to inspect the inside of the Standard Library. In any case, we should be able to provide an alternative Standard Library package compiled with interpretation turned on, if required.

6.4 Use in the REPL

Another pleasant side effect of the way in which we have been able to use the PPX system to embed our step-by-step interpreter into bytecode and native code programs is that, of course, it could be extended to work with the normal OCaml REPL (which functions by compiling phrases to bytecode and then executing them). And so, to the REPL's playground is added a new toy. We may put `[@interpret]` annotations to see the steps of execution.

6.5 Summary

We have found a pleasant interface to our step-by-step interpreter which might make debugging a better experience, meeting many of the challenges of accessibility with which we have been most concerned. Along the way, we have confirmed that our new OCamli2 seems to fix the technical problems of our original OCamli implementation with regard to the interpreted/native interface. We have sketched an implementation to show that the idea is technically feasible (the state of the implementation of this work is given in section 8.6). Once we have a working implementation of both OCamli and this PPX-based interface, the debugger will be usable, and so may be evaluated more fully. Only when the debugger is in use, as we have repeatedly said, can we know for certain that it is useful. But we have been careful to design it to be so.

Chapter 7

Roads not taken

The first rule of intelligent tinkering is to save all the parts.

— Paul Ralph Ehrlich

Along the way, we turn down some blind alleys. One of the pleasant peculiarities of the thesis as a form, when compared with conference or journal papers, is the space to include what was observed in those alleys, before we were forced by failure or the constraints of time to return to the main path.

In this chapter we describe, in turn, an interesting way to improve error messages in an interpreter by allowing it to attempt to evaluate programs which are ill-typed, a mechanism for including arbitrary interpreted code in ordinary OCaml programs to be run at compile-time, and a way of visualising the execution of OCaml bytecode programs without needing interpretation at all.

7.1 Turning off the typechecker

As an aid to debugging OCamli itself, the command line option `-no-typecheck` was introduced. This option, when enabled, parses and interprets the code without running the OCaml typechecker. The interpreter, which normally relies on programs having been typechecked for correct operation, will then run an incorrect program as far as it can, concluding either with an incorrect result, or an exception raised from the interpreter.

Might this option have legitimate uses outside of debugging the interpreter itself? A program which will not compile is a kind of bug, in a sense. Especially so when other languages with weaker type systems would have compiled the same source code to a buggy executable without complaint. The question "Why doesn't my code compile?" is a variant on "Why does my code fail at run-time?" because they are both, really, "Why doesn't my code work?".

The usual mantra when debugging is that the failure should be as close to the source of the problem as possible, helping us to pinpoint the bug. However, we think that, in the case of recursive functions, it might be helpful sometimes to find errors slightly later – when a badly typed value is received by code which does not expect it, rather than when it might

be created (as a type inference engine would find the error). Turning the typechecker off allows such ill formed values to be created and propagated.

Presently, some ill-typed programs fail because no case is there to handle them in the interpreter. For example, take the expression `1 + false`, and try to evaluate it in the normal OCaml REPL:

```
# 1 + false;;
Error: This expression has type bool but an expression was expected of type int
```

In OCamli, the addition operator looks for two integers, or an unevaluated left or right side. For now, there is no case to handle the type mismatch, and so we see an unhelpful error message:

```
$ ocamli -e '1 + false' -no-typecheck -show-all
    1 + false
Error in Eval.next Failure("already a value or unimplemented: false")
```

The actual text of this error message is of no use except to the interpreter-writer. It indicates that, not being able to find two integers to add, the interpreter tried to evaluate the right-hand-side. Since the evaluator is never run on values, and `false` is a value, this error results. Let us look at another example. A classic beginner's mistake in functional languages is to confuse list consing and list concatenation:

```
# [1] :: [2] @ [3];;
Error: This expression has type int but an expression was expected of type int list
```

In this instance, OCamli with `-no-typecheck` runs the program with no problem at all:

```
$ ocamli -e '[1] :: [2] @ [3]' -no-typecheck -show-all
    [[1]; 2] @ [3]
=>  [[1]; 2; 3]
```

This is, of course, because both `::` and `@` are polymorphic. So this is an example where the interpreter cannot currently spot the type error at run-time.

We should like to alter OCamli to properly report type errors, such as adding an integer to a boolean, where the error message is currently unhelpful. In addition, we should like to add errors for things like the cons/concatenate confusion above. Do we detect this kind on value creation or value consumption? Presently, the code for arithmetic operators in our interpreter looks like this:

```
| Op (op, Int a, Int b) ->                                    we have two integers
    begin try Int (calc op a b) with
      Division_by_zero -> Raise ("Division_by_zero", None)
    end
| Op (op, Int a, b) -> Op (op, Int a, eval peek env b)        evaluate the right-hand side
| Op (op, a, b) -> Op (op, eval peek env a, b)                evaluate the left-hand side
```

The final cases are too generic in an ill-typed program. They are those cases where the left or right-hand side are not integers, and not compound expressions, but non-integer values. If `a` and `b` are both values, but not both integers, we can give the appropriate message:

```
| Op (op, Int a, Int b) ->                                    we have two integers
    begin try Int (calc op a b) with
      Division_by_zero -> Raise ("Division_by_zero", None)
    end
| Op (op, a, b) when is_value a && is_value b ->              values not both integers
    raise
      (RuntimeTypeError
         (Printf.sprintf
            "operation %s can operate only on integers"
            (string_of_op op)))
| Op (op, a, b) when is_value a -> Op (op, Int a, eval peek env b)      any value on left
| Op (op, a, b) -> Op (op, eval peek env a, b)
```

The cons/append example is harder. It is difficult to see how polymorphic operations can be typechecked at run-time. Do we need a type inference engine? Or just a "same-type" check to make sure that the new item being consed on to the front has the same type as the elements already in the list? As another example, how might we deal with generic comparison? We cannot actually check the types if we are using the real generic comparison operator:

```
$ ocaml -e '1 < 2 < 3' -no-typecheck -show-all
    1 < 2 < 3
=>  true < 3
=>  false
```

We can indeed develop a "same-type" approximation, applying to just values (excepting functions). Due to OCaml's order of evaluation, we can only find the run-time type error when the expression has been reduced to a value. This is a shame, because it means we see it quite late in the process. But, as we have discussed, this can be useful to the beginner too. So, for example, we now get a useful error:

```
$ ocaml -e '(fun x y -> x :: y) 2 ['a']' -no-typecheck -show-all
    (fun x y -> x::y) 2 ['a']
=>  (let x = 2 in fun y -> x::y) ['a']
=>  (fun y -> let x = 2 in x::y) ['a']
=>  let y = ['a'] in let x = 2 in x::y
=>  let y = ['a'] in 2::y
Run time type error:
  Cannot cons onto this list: differing element types
```

Now, for our generic comparison example on what is, to a beginner, an innocuous piece of code – because it is correct mathematics:

```
$ ocamli -e '1 < 2 < 3' -no-typecheck -show-all
    1 < 2 < 3
=>  true < 3
Run time type error:
  Comparison between values of differing types
```

This is better for the beginner than the static type error:

```
$ ocamli -e '1 < 2 < 3' -show-all
File "", line 1, characters 8-9:
Error: This expression has type int but an expression was expected of type bool
```

Consider another example. Here, the programmer has left out `f` in the recursive call to `map`

```
let g x = x + 1

let rec map f = function
  | [] -> []
  | h::t -> f h :: map t

let l = map g [1; 2; 3]
```

This leads to a type error in the OCaml REPL:

```
# let rec map f = function [] -> [] | h::t > f h :: map t;;
Error: This expression has type 'a list
       but an expression was expected of type 'a -> 'b
```

What we get in our modified OCamli with run-time type detection is:

```
=>  let g x = x + 1

    let rec map f = function [] -> [] | h::t -> f h :: map t

    let l = let rec map f = function [] -> [] | h::t -> f h :: map t in 2 :: map [2; 3]
Run time type error:
  Attempt to cons onto non-list
```

Whilst this is true (`map [2; 3]` is a function, and so a value), it is not quite the intuitive error message we would like. A much wider study of this phenomenon vis-a-vis common beginner errors would be needed to see if this approach of turning the typechecker off has wide applicability.

Another approach to improving OCaml error messages is to modify the typechecker itself [Charguéraud, 2015]. Trying to improve error messages in type inference has a long record of research – two recent examples are the localization approach [Pavlinovic et al., 2014] and the type-debugger [Lerner et al., 2006].

Our work, in any event, is not useable under OCamli2, where full typechecking is required – recall, from chapter 5, that types must be available at every node so that heap values can be converted to strings to be printed by the interpreter during step-by-step debugging. Hence its presence in this chapter.

7.2 Compile-time interpretation

Now that we have an interpreter for OCaml, what can we use it for, other than our intended purpose of debugging? Of course, OCamli can be built as a library, and so can be linked into other programs. For example, linking the OCaml REPL with OCamli allows one to evaluate arbitrary OCaml source at run-time:

```
# let s = Runeval.eval_string "List.split [(1, 2); (3, 4)]";;
val s : string = "([1; 3], [2; 4])"
```

Unfortunately, the result is a string. However, we have functions to deal with that already, from our work interfacing with C. We can convert this string to a real OCaml value, though the user must provide the type in the REPL, and it must be correct:

```
# let x : int list * int list =
    Tinyexternal.to_ocaml_value (snd (Tinyocamlrw.of_string s));;
val x : int list * int list = ([1; 3], [2; 4])
```

Now, we can build OCaml expressions as strings or ASTs at run-time and typecheck and evaluate them. This is fun, but a better metaprogamming solution for OCaml already exists in the form of MetaOCaml [Kiselyov, 2014] which provides guarantees of well-typedness – that is to say, a well-typed MetaOCaml program can generate only well-typed OCaml programs at run-time.

A more practical application of this kind of ad hoc code generation takes the form of another PPX extension, to allow arbitrary code to be executed at compile-time and the resultant value inserted in source code of the program under compilation. With the ppx_eval extension we can write the following:

```
let compiler_command = [%compiletime "Sys.argv.(0)"]
```

This code, in a normal compiled OCaml program with ppx_eval might generate this, the string representing the compiler command itself:

```
let compiler_command = "ocamlopt"
```

This can do jobs normally performed by preprocessing tools, but in a way which respects the OCaml grammar. Another example might be to include in the file the date of its compilation. In many codebases (including the OCaml compiler itself), there are one or more places where a preprocessor such as sed is used, and each time the behaviour of such a tool on a given source file must be checked, and re-checked when the file is significantly altered. Depending upon an external command-line tool like sed for preprocessing is fraught, since it can behave differently on each operating system. By bringing the functionality into OCaml, we reduce such concerns.

This is interesting work, which merits further study, but is not directly relevant to our core aim of debugging OCaml programs, so it was not taken further. It is merely a consequence of our having written an interpreter, not a prerequisite for our debugging work.

7.3 Debuggable bytecode by decompilation

Instead of writing an interpreter for OCaml, could we instead simply modify OCaml's own bytecode interpreter to print the current step of evaluation at any given moment, thus producing a trace of the program's execution without affecting execution speed unduly?

OCaml's bytecode system is based on the ZINC machine [Leroy, 1990]. We shall consider the simplified version designed by the same author for pedagogical purposes [Leroy, 2015], look at the compilation scheme it uses, and see if we can derive a decompilation scheme.

7.3.1 Programs

Our programs are defined using a tiny subset of OCaml. Variable accesses have been converted to deBruijn indices when the program was converted from the OCaml parse tree. Here is the type for programs:

```
type op = Add | Sub | Mul | Div

type prog =
  Int of int
| Bool of bool
| Var of int
| Eq of prog * prog
| Op of prog * op * prog
| Apply of prog * prog
| Lambda of prog
| Let of prog * prog
```

For example, the OCaml program

```
let x = 5 in if x = 4 then 1 else (fun x -> x + 1) 2
```

may be represented as:

```
Let (Int 5,
     If (Eq (Var 1, Int 4),
         Int 1,
         Apply (Lambda (Op (Var 1, Add, Int 1), Int 2))))
```

7.3.2 Compilation scheme

The abstract machine instructions follow [Leroy, 2015] but we add booleans, IF and the equality test EQ:

EMPTY	*program ends*
INT(integer)	*integer*
BOOL(boolean)	*boolean*
OP(op)	*arithmetic operators*

EQ	*test for equality*
ACCESS(integer)	*fetch value of name*
CLOSURE(instructions)	*closures*
LET	*let bindings*
ENDLET	
APPLY	*function application*
RETURN	*last instruction in closure*
IF	*if… then… else …*

Here is the compilation scheme $\mathcal{C}$, again extended from Leroy:

$$\begin{aligned}
\mathcal{C}(\texttt{Int}(i)) &= \text{INT}(i) \\
\mathcal{C}(\texttt{Bool}(b)) &= \text{BOOL}(b) \\
\mathcal{C}(\texttt{Op}(a, \oplus, b)) &= \mathcal{C}(a); \mathcal{C}(b); \text{OP}(\oplus) \\
\mathcal{C}(\texttt{Eq}(a, b)) &= \mathcal{C}(a); \mathcal{C}(b); \text{EQ} \\
\mathcal{C}(\texttt{Var}(n)) &= \text{ACCESS}(n) \\
\mathcal{C}(\texttt{Lambda}(a)) &= \text{CLOSURE}(\mathcal{C}(a); \text{RETURN}) \\
\mathcal{C}(\texttt{Let}(a, b)) &= \mathcal{C}(a); \text{LET}; \mathcal{C}(b); \text{ENDLET} \\
\mathcal{C}(\texttt{Apply}(a, b)) &= \mathcal{C}(a); \mathcal{C}(b); \text{APPLY} \\
\mathcal{C}(\texttt{If}(a, b, c)) &= \mathcal{C}(\texttt{Lambda}(b)); \mathcal{C}(\texttt{Lambda}(c)); \mathcal{C}(a); \text{IF}
\end{aligned}$$

So our example

```
let x = 5 in if x = 4 then 1 else (fun x -> x + 1) 2
```

compiles to:

```
INT 5
LET
CLOSURE
  INT 1
  RETURN
CLOSURE
  CLOSURE
    ACCESS 1
    INT 1
    OP +
    RETURN
  INT 2
  APPLY
  RETURN
ACCESS 1
INT 4
```

EQ
BRANCH
ENDLET
EMPTY

Notice the EMPTY inserted at the end. We have used indentation here to make it easier to see the structure of the CLOSUREs but it is, in reality, a simple list of instructions with no structure. The compilation process is illustrated in figure 10.

7.3.3 Evaluation scheme

Here is the evaluation scheme $\mathcal{E}$, again extended from [Leroy, 2015] with our new instructions.

Machine state before			Machine state after		
Code	Env	Stack	Code	Env	Stack
INT$(i); c$	e	s	c	e	$i.s$
BOOL$(b); c$	e	s	c	e	$b.s$
OP$(\oplus); c$	e	$i.i'.s$	c	e	$\oplus(i, i').s$
EQ$; c$	e	$i.i'.s$	c	e	$(i = i').s$
ACCESS$(n); c$	e	s	c	e	$e(n).s$
CLOSURE$(c'); c$	e	s	c	e	$c'[e].s$
LET$; c$	e	$v.s$	c	$v.e$	s
ENDLET$; c$	$v.e$	s	c	e	s
APPLY$; c$	e	$v.c'[e'].s$	c'	$v.e'$	$c.e.s$
RETURN$; c$	e	$v.c'.e'.s$	c'	e'	$v.s$
IF$; c$	e	$\mathsf{T}.c'[e'].c''[e''].s$	c'	e'	$c[e].s$
IF$; c$	e	$\mathsf{F}.c'[e'].c''[e''].s$	c''	e''	$c[e].s$

There are three components: a code pointer c (instructions yet to be reached), an environment e, and a stack s (intermediate results and pending function calls). The notation $c[e]$ is the closure of code c with environment e.

The final result is at the top of the stack when the code is EMPTY. The evaluation process is illustrated in figure 11.

7.3.4 Decompilation scheme

In order to print the evaluation step by step, we need to be able to decompile:

- Any program which has been compiled by the compilation scheme above.
- Certain incomplete evaluations under the evaluation scheme above. That is to say, given (c, s) we can decompile a program which represents the evaluation at that stage. We need not be able to decompile arbitrary (c, e, s) triples.

We add names to `Var`, `Lambda` and `Let`, since we shall need to recover variable names during decompilation:

```
type prog =
  Int of int
| Bool of bool
| Var of name * int
| Eq of prog * prog
| Op of prog * op * prog
| Apply of prog * prog
| Lambda of name * prog
| Let of name * prog * prog
```

So we must add names to the ACCESS, CLOSURE and LET instructions. These are not required for evaluation, of course, but only for decompilation:

EMPTY	*program ends*
INT(integer)	*integer*
BOOL(boolean)	*boolean*
OP(op)	*arithmetic operators*
EQ	*test for equality*
ACCESS(name, integer)	*fetch value of name*
CLOSURE(name, instructions)	*closures*
LET(name)	*let bindings*
ENDLET	
APPLY	*function application*
RETURN	*last instruction in closure*
IF	*if… then… else …*

Decompilation is performed by going through the instructions in order, holding a stack a little like the evaluation stack, but which may also contain decompiled program fragments – the empty stack is written $\{\}$. When we have gone through all the instructions, the final program is at the top of the stack. We do not need the environment, since we are not running the code, just decompiling it. Here is the decompilation scheme $\mathcal{D}$:

$$
\begin{aligned}
\mathcal{D}(\text{EMPTY}, v.s) &= v \\
\mathcal{D}(\text{INT}(i); c, s) &= \mathcal{D}(c, \texttt{Int}(i).s) \\
\mathcal{D}(\text{BOOL}(i); c, s) &= \mathcal{D}(c, \texttt{Bool}(b).s) \\
\mathcal{D}(\text{OP}(\oplus); c, i.i'.s) &= \mathcal{D}(c, \texttt{Op}(i, \oplus, i').s) \\
\mathcal{D}(\text{EQ}; c, i.i'.s) &= \mathcal{D}(c, \texttt{Eq}(i, i').s) \\
\mathcal{D}(\text{ACCESS}(n, l); c, s) &= \mathcal{D}(c, \texttt{VarAccess}(n, l).s) \\
\mathcal{D}(\text{CLOSURE}(n, c'); c, s) &= \mathcal{D}(c, c'[n, \{\}].s) \\
\mathcal{D}(\text{LET}(n); c, v.s) &= \texttt{Let}(n, v, \mathcal{D}(c, s)) \\
\mathcal{D}(\text{ENDLET}; c, s) &= \mathcal{D}(c, s) \\
\mathcal{D}(\text{APPLY}; c, v.c'[n, e'].s) &= \texttt{Apply}(\texttt{Lambda(n}, \mathcal{D}(c', \{\})), v) \\
\mathcal{D}(\text{RETURN}; c, v.c'.e'.s) &= \mathcal{D}(c', v.s) \\
\mathcal{D}(\text{RETURN}; c, s) &= \mathcal{D}(c, s)
\end{aligned}
$$

$$\mathcal{D}(\text{IF}; c, e.c'[e'].c''[e''].s) = \mathcal{D}(c, \texttt{If}(e, \mathcal{D}(c', s), \mathcal{D}(c'', s)).s)$$

This decompiler works for:

- Any program-stack pair (P, {}) where P was compiled by $\mathcal{C}$ above.
- A program-stack pair (P, S) which is an intermediate state of the evaluation procedure $\mathcal{E}$ (minus the environment) where P begins with an instruction such as OP or APPLY.

Our example program decompiles properly from bytecode, both as a whole, and when part-evaluated. Two such examples are shown in figures 12 and 13.

7.3.5 Prototype

Once we have such a decompilation regime, we can build a step-by-step interpreter very easily. We compile the program to bytecode, then evaluate it one bytecode instruction at a time, decompiling after each step, and displaying the resulting program. There would be many repeated lines, for example when an instruction simply puts something onto the stack. We need a notion of what constitutes an 'interesting' instruction execution. Presently we consider a step 'interesting' if it follows immediately the execution of an ACCESS, BRANCH, OP, or EQ instruction. For our program, this gives the following.

```
$ ./bytecode tests/example.ml
let x = 5 in if x = 4 then 1 else (fun x -> x + 1) 2
if 5 = 4 then 1 else (fun x -> x + 1) 2
if false then 1 else (fun x -> x + 1) 2
(fun x -> x + 1) 2
2 + 1
3
```

It is somewhat unintuitive that such a simple scheme should work so well – eliding more than our own interpreter by default and mimicking so well what we might write on paper, given that it operates at such a low level. However, since the process of compilation is intended to reduce a program to an efficient form where the fewest or fastest instructions are generated and each instruction or little sequence of instructions does something to make definite progress in a computation, it is perhaps not so surprising.

This work has produced some intriguing results – its properties we have just described, and its undoubted speed, but we have put it aside. Our new mechanism for selective interpretation, described in the previous chapter, probably makes the speed increase unimportant in most cases, and plain interpretation continues to have the compelling advantage of full information at all times. It will be interesting, though, to compare the results of interpreting (with regard to quality of output) sample programs such as those in Appendix B, between OCaml, OCaml2 and the bytecode decompiler described, once the projects are all in a state in which such a comparison is possible.

7.4 Summary

In this chapter we have exhibited three pieces of work which, whilst we consider them to be interesting and worth describing for an audience, were either dead ends due to design decisions taken elsewhere, or which simply became surplus to requirement for the same reason.

In chapter 3, we have justified our work. In chapters 4, 5, 6 and 7 we have described it. Now it is time to step back, and evaluate it in as dispassionate and impartial way as is possible.

$\mathcal{C}$(`Let(Int 5, If(Eq(Var 1, Int 4), Int 1, Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))))`

Rule $\mathcal{C}$-`Let`

$\mathcal{C}$(`Int 5`); LET; $\mathcal{C}$(`If(Eq(Var 1, Int 4), Int 1, Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))`); ENDLET

Rule $\mathcal{C}$-`Int`

INT 5; LET; $\mathcal{C}$(`If(Eq(Var 1, Int 4), Int 1, Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))`); ENDLET

Rule $\mathcal{C}$-`If`

INT 5; LET; $\mathcal{C}$(`Lambda (Int 1)`); $\mathcal{C}$(`Lambda(Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))`); $\mathcal{C}$(`Eq(Var 1, Int 4)`); IF; ENDLET

Rule $\mathcal{C}$-`Eq` then Rule $\mathcal{C}$-`Eq` then Rule $\mathcal{C}$-`Eq`

INT 5; LET; $\mathcal{C}$(`Lambda (Int 1)`); $\mathcal{C}$(`Lambda(Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))`); ACCESS 1; INT 4; EQ; IF; ENDLET

Rule $\mathcal{C}$-`Lambda` then Rule $\mathcal{C}$-`Int`

INT 5; LET; CLOSURE [INT 1; RETURN]; $\mathcal{C}$(`Lambda(Apply(Lambda(Op(Var 1, Add, Int 1), Int 2)))`); ACCESS 1; INT 4; EQ; IF; ENDLET

Rule $\mathcal{C}$-`Lambda`

INT 5; LET; CLOSURE [INT 1; RETURN]; CLOSURE [$\mathcal{C}$(`Apply(Lambda(Op(Var 1, Add, Int 1), Int 2))`); RETURN]; ACCESS 1; INT 4; EQ; IF; ENDLET

Rule $\mathcal{C}$-`Apply`

INT 5; LET; CLOSURE [INT 1; RETURN]; CLOSURE [$\mathcal{C}$(`Lambda(Op(Var 1, Add, Int 1))`; $\mathcal{C}$(`Int 2`); APPLY; RETURN]; ACCESS 1; INT 4; EQ; IF; ENDLET

Rule $\mathcal{C}$-`Int` then Rule $\mathcal{C}$-`Lambda` then Rule $\mathcal{C}$-`Op` then Rule $\mathcal{C}$-`Var` then Rule $\mathcal{C}$-`Int`

INT 5; LET; CLOSURE [INT 1; RETURN]; CLOSURE [CLOSURE [ACCESS 1; INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; ACCESS 1; INT 4; EQ; IF; ENDLET

Figure 12. Compilation of the program `let x = 5 in if x = 4 then 1 else (fun x -> x + 1) 2` with $\mathcal{C}$.

	Machine state after		
Instruction	Code	Env	Stack
-	INT 5; LET; CLOSURE [INT 1; RETURN]; CLOSURE [CLOSURE [ACCESS 1; INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; ACCESS 1; INT 4; EQ; IF; ENDLET	{}	{}
INT	LET; CLOSURE [INT 1; RETURN]; CLOSURE [CLOSURE [ACCESS 1; INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; ACCESS 1; INT 4; EQ; IF; ENDLET	{}	{5}
LET	CLOSURE [INT 1; RETURN]; CLOSURE [CLOSURE [ACCESS 1; INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; ACCESS 1; INT 4; EQ; IF; ENDLET	{5}	{}
CLOSURE	CLOSURE [CLOSURE [ACCESS 1; INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; ACCESS 1; INT 4; EQ; IF; ENDLET	{5}	{[INT 1; RETURN]{5}}
CLOSURE	ACCESS 1; INT 4; EQ; IF; ENDLET	{5}	{[CLOSURE [ACCESS 1; INT 1; OP +; RETURN]{5}; INT 2; APPLY; RETURN]; [INT 1; RETURN]{5}}
ACCESS	INT 4; EQ; IF; ENDLET	{5}	{5; [CLOSURE [ACCESS 1; INT 1; OP +; RETURN]{5}; INT 2; APPLY; RETURN]; [INT 1; RETURN]{5}}
INT	EQ; IF; ENDLET	{5}	{4; 5; [CLOSURE [ACCESS 1; INT 1; OP +; RETURN]{5}; INT 2; APPLY; RETURN]; [INT 1; RETURN]{5}}
EQ	IF; ENDLET	{5}	{false; [CLOSURE [ACCESS 1; INT 1; OP +; RETURN]{5}; INT 2; APPLY; RETURN]; [INT 1; RETURN]{5}}
IF	CLOSURE [ACCESS 1; INT 1; OP +; RETURN]; INT 2; APPLY; RETURN	{5}	{[ENDLET]; {5}}
CLOSURE	INT 2; APPLY; RETURN	{5}	{[ACCESS 1; INT 1; OP +; RETURN]{5}; [ENDLET]; {5}}
INT	APPLY; RETURN	{5}	{2; [ACCESS 1; INT 1; OP +; RETURN]{5}; [ENDLET]; {5}}
APPLY	ACCESS 1; INT 1; OP +; RETURN	{2; 5}	{[RETURN]; {5}; [ENDLET]; {5}}
ACCESS	INT 1; OP +; RETURN	{2; 5}	{2; [RETURN]; {5}; [ENDLET]; {5}}
INT	OP +; RETURN	{2; 5}	{1; 2; [RETURN]; {5}; [ENDLET]; {5}}
OP	RETURN	{2; 5}	{3; [RETURN]; {5}; [ENDLET]; {5}}
RETURN	EMPTY	{2; 5}	{3; [RETURN]; {5}; [ENDLET]; {5}}
RETURN	EMPTY	{5}	{3; [ENDLET]; {5}}
ENDLET	EMPTY	{}	{3}
EMPTY			

Figure 13. Evaluation of `let x = 5 in if x = 4 then 1 else (fun x -> x + 1) 2` under $\mathcal{E}$. Stacks and environments are written {items}, and a closure on the stack is written [instructions]{environment}. Environments may be put on the stack.

$\mathcal{D}$(INT 5; LET x; CLOSURE [INT 1; RETURN]; CLOSURE [CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; ACCESS (x, 1); INT 4; EQ; IF; ENDLET, {})
Rule $\mathcal{D}$-CONST
$\mathcal{D}$(LET x; CLOSURE [INT 1; RETURN]; CLOSURE [CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; ACCESS (x, 1); INT 4; EQ; IF; ENDLET, {`Int 5`})
Rule $\mathcal{D}$-LET
`Let(x, Int 5,` $\mathcal{D}$(CLOSURE [INT 1; RETURN]; CLOSURE [CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; ACCESS (x, 1); INT 4; EQ; IF; ENDLET, {}))
Rule $\mathcal{D}$-CLOSURE
`Let(x, Int 5,` $\mathcal{D}$(CLOSURE [CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; ACCESS (x, 1); INT 4; EQ; IF; ENDLET, {CLOSURE [INT 1; RETURN]}))
Rule $\mathcal{D}$-CLOSURE
`Let(x, Int 5,` $\mathcal{D}$(ACCESS (x, 1); INT 4; EQ; IF; ENDLET, { CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN; INT 2; APPLY; RETURN]; CLOSURE [INT 1; RETURN]}))
Rule $\mathcal{D}$-ACCESS
`Let(x, Int 5,` $\mathcal{D}$(INT 4; EQ; IF; ENDLET, {`VarAccess(x, 1)`; CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN; INT 2; APPLY; RETURN]; CLOSURE [INT 1; RETURN]}))
Rule $\mathcal{D}$-INT
`Let(x, Int 5,` $\mathcal{D}$(EQ; IF; ENDLET, {`Int 4; VarAccess(x, 1)`; CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN; CLOSURE [INT 1; RETURN]}))
Rule $\mathcal{D}$-EQ
`Let(x, Int 5,` $\mathcal{D}$(IF; ENDLET, {`Eq(VarAccess (x, 1), Int 4)`; CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN; CLOSURE [INT 1; RETURN]}))
Rule $\mathcal{D}$-IF
`Let(x, Int 5,` $\mathcal{D}$(ENDLET, {`If(Eq(VarAccess (x, 1), Int 4),` $\mathcal{D}$(CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN, {}), $\mathcal{D}$(CLOSURE [INT 1; RETURN], {}))
Rule $\mathcal{D}$-ENDLET
`Let(x, Int 5, If(Eq(VarAccess (x, 1), Int 4),` $\mathcal{D}$(CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN, {}), $\mathcal{D}$(CLOSURE [INT 1; RETURN], {}))
Rule $\mathcal{D}$-CLOSURE then $\mathcal{D}$-INT then $\mathcal{D}$-RETURN
`Let(x, Int 5, If(Eq(VarAccess (x, 1), Int 4),` $\mathcal{D}$(CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN, {}), `Int 1))`
Rule $\mathcal{D}$-CLOSURE
`Let(x, Int 5, If(Eq(VarAccess (x, 1), Int 4),` $\mathcal{D}$(INT 2; APPLY; RETURN, {CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]}), `Int 1))`
Rule $\mathcal{D}$-INT
`Let(x, Int 5, If(Eq(VarAccess (x, 1), Int 4),` $\mathcal{D}$(APPLY; RETURN, {`Int 2`; CLOSURE [ACCESS (x, 1); INT 1; OP +; RETURN]}), `Int 1))`
Rule $\mathcal{D}$-APPLY
`Let(x, Int 5, If(Eq(VarAccess (x, 1), Int 4), Apply(`$\mathcal{D}$(ACCESS (x, 1); INT 1; OP +; RETURN, {}), `Int 2), Int 1))`
Rule $\mathcal{D}$-ACCESS then $\mathcal{D}$-INT then $\mathcal{D}$-OP then $\mathcal{D}$-RETURN
`Let(x, Int 5, If(Eq(VarAccess (x, 1), Int 4), Apply(Lambda(Op(Var 1, Add, Int 1)), Int 2), Int 1))`

Figure 14. Decompilation under $\mathcal{D}$ of the program **`let`** `x = 5` **`in if`** `x = 4` **`then`** `1` **`else`** `(`**`fun`** `x -> x + 1) 2` compiled under $\mathcal{C}$, with an empty stack to begin, since the program is unexecuted.

$\mathcal{D}$(IF; ENDLET, false; [CLOSURE [ACCESS(x, 1); INT 1; OP +; RETURN]; INT 2; APPLY; RETURN]; [INT 1; RETURN])
Rule $\mathcal{D}$-IF
$\mathcal{D}$(ENDLET, `If (false,` $\mathcal{D}$([INT 1; RETURN], {})) $\mathcal{D}$([CLOSURE [ACCESS(x, 1); INT 1; OP +: RETURN]; INT 2; APPLY; RETURN], {}), {})
Rule $\mathcal{D}$-ENDLET
`If (Bool false,` $\mathcal{D}$([INT 1; RETURN], {}), $\mathcal{D}$([CLOSURE [ACCESS(x, 1); INT 1; OP +: RETURN]; INT 2; APPLY; RETURN]{}), `)`
Rule $\mathcal{D}$-INT then $\mathcal{D}$-RETURN
`If (Bool false, Int 1,` $\mathcal{D}$([CLOSURE [ACCESS(x, 1); INT 1; OP +: RETURN]; INT 2; APPLY; RETURN]{}),`)`
Rule $\mathcal{D}$-CLOSURE
`If (Bool false, Int 1,` $\mathcal{D}$([INT 2; APPLY; RETURN], {[ACCESS(x, 1); INT 1; OP +: RETURN]}))
Rule $\mathcal{D}$-INT
`If (Bool false, Int 1,` $\mathcal{D}$([APPLY; RETURN], {`Int 2`; [ACCESS(x, 1); INT 1; OP +: RETURN]}))
Rule $\mathcal{D}$-APPLY
`If (Bool false, Int 1, Apply (Lambda(x,` $\mathcal{D}$(ACCESS(x, 1); INT 1; OP +: RETURN, {}), `Int 2)))`
Rule $\mathcal{D}$-ACCESS
`If (Bool false, Int 1, Apply (Lambda(x,` $\mathcal{D}$(INT 1; OP +; RETURN, {`VarAccess(x, 1)`}), `Int 2)))`
Rule $\mathcal{D}$-INT
`If (Bool false, Int 1, Apply (Lambda(x,` $\mathcal{D}$(OP +; RETURN, {`Int 1; VarAccess(x, 1)`}), `Int 2)))`
Rule $\mathcal{D}$-OP
`If (Bool false, Int 1, Apply (Lambda(x,` $\mathcal{D}$(RETURN, {`Op(VarAccess(x, 1), Add, Int 1)`}), `Int 2)))`
Rule $\mathcal{D}$-RETURN
`If (Bool false, Int 1, Apply (Lambda(x, Op(VarAccess(x, 1), Add, Int 1), Int 2)))`

This is the program `if false then 1 else (fun x -> x + 1) 2`, as required.

Figure 15. Decompilation under $\mathcal{D}$ of the program `let x = 5 in if x = 4 then 1 else (fun x -> x + 1) 2` compiled under $\mathcal{C}$, and partly evaluated with $\mathcal{D}$.

Chapter 8

Evaluation

> *In Africa a thing is true at first light and a lie by noon and you have no more respect for it than for the lovely, perfect weed-fringed lake you see across the sun-baked salt plain. You have walked across the plain in the morning and you know that no such lake is there. But now it is there absolutely true, beautiful and believable.*
>
> — Hemingway, *True at First Light*

We have looked again at the problem of debugging, identified what we believe to be a step forward which may lead to more people using debugging tools, and built a proof-of-concept debugger for the functional language OCaml. As we look back upon the prototype we have built and plan how to turn it into a finished and usable tool, can we evaluate what has been done to inform our future work? By what criteria might we measure success?

The primary measure, of course, is whether the tool, once finished, is widely used. There is little we can do about this type of evaluation now, but it must remain our most important concern. Quantitatively, 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?

For now, we can evaluate only what has been done, and the design of the whole tool as it is envisaged. We shall do this by qualitatively measuring our progress against our stated aims. First, we shall give a broad, narrative discussion of the results of our work, to provide a general assessment. Then, we shall use more structured approaches – comparing our progress with the research questions, and against various criteria for successful debuggers given in the literature. Then, we will discuss what methods of evaluation might be possible once the work is more advanced. Finally, we give details of the implementation status of our work.

Summary of Work

In chapter 1, we describe twin motivations for the work: experience in teaching and in real-world programming tasks. We decide upon research questions, and a thesis. Chapter 2 contains two literature reviews: of historical literature in debugging and of debugging

today, both in functional and imperative settings. In chapter 3 we decide upon an approach to the work, taking lessons from our literature review. We review literature in the related field of software visualization. Chapter 4 describes our first, experimental implementation of a step-by-step interpreter and explores some of its difficulties and deficiencies. In chapter 5, we re-assess the interpreter from the previous chapter and produce a new design with numerous improvements, such that it constitutes a practical system. Chapter 6 describes an annotation-based design for an interface to our interpreter, forming the heart of our eventual approach to debugging. In chapter 7, we talk about pieces of work which did not make the final cut, and why they did not. In this chapter, chapter 8, we evaluate the work and update the research questions. Finally, in chapter 9, we give concluding remarks.

A statement of our contributions to research is given in section 1.5.

8.1 Narrative discussion

Our plan was to look at the literature and practice of debugging, trying 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 significant 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 this accessibility requirement being fulfilled, the rest is in vain.

We chose a radical approach to these key requirements: to build an interpreter for our chosen language, OCaml. 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. We produced two systems, `OCamli` and `OCamli2/ppx_interpret`. How did we fare? Let us take them in turn.

Our first program, `OCamli` (see chapter 4) was written, first, to answer the question "What would an interpreter for OCaml look like?", and second to begin to explore the design of a visualizer – and thereby a debugger – for OCaml programs. However, it fails on several counts to 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 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 3.3? `OCamli`, as presently constructed, fails the most important of our 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. It is clear that the other test we set, about being easy to keep in sync with the toolchain, has not been fully achieved at this stage. Using the library form of the OCaml compiler helps us a lot, but there would still be significant work to make the software function with each new compiler version.

Let us compare the OCamli model with the typical debugging methods we have talked about, both in imperative and functional languages: breakpointing, inserting print statements, using a REPL, and tracing. It is clear that breakpointing, either in its traditional form, or by search mechanisms, can be replicated. OCamli effectively introduces universal printing of values (which OCaml does not have by default), a clear boon for debugging. However, the OCamli command line interface is clearly inferior to the REPL (save for the ability to show the steps of evaluation). An interactive REPL-like version of OCamli should be explored as an alternative debugging interface. We consider the online behaviour of OCamli-style interpretation as likely more useful than offline tracing (e.g. Haskell's Hood [Gill, 2000]), due to the ability of the user to refine and experiment interactively.

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.

Let us now consider our second program OCamli2/ppx_interpret (see chapters 5 and 6) in the same way. This, we believe, will meet our principles of usability when it is complete. It should make a lightweight debugging mechanism which may be invoked at will no matter the environment, build system, or other circumstances. It meets the tests set out in section 3.3, due to a happy coincidence in the way that the PPX mechanism works. It can be used with any build system, works with mixed C/OCaml code, is relatively simple to keep in sync with compiler toolchain releases, requires only a tiny patch to OCaml itself, and is flexible enough that it should be suitable for debugging large projects, even the OCaml toolchain itself. Just like OCamli, ppx_interpret provides the opportunity for universal printing, exploration of the steps of evaluation, and opportunities for interaction. We have identified a number of encouraging avenues for interface design of ppx_interpret functionality (see chapter 6), and we would expect some or many of those options to provide a pleasurable debugging experience. So, ppx_interpret is promising, and a definite improvement to OCamli with regard to our aims and the tests which we have set ourselves.

8.2 With regard to teaching

The impetus for this research was the author's experience of teaching functional programming to beginners, some of whom had never written a computer programs before. The step-by-step diagrams which our interpreter produces are frequently drawn out by hand in such lessons because it demonstrates the semantic operation of the program text so well. However, once we decided our focus must be debugging for the working programmer, we hoped merely that the teaching uses would be subsumed by the work on debugging – that a good debugger for the working programmer could be a good debugger for the student. Has that happened? The prototype OCamli has been written to cover enough of

the language to execute almost all the programs in the author's own introductory OCaml textbook "OCaml from the Very Beginning" [Whitington, 2013], and so we can try it on those programs. Whilst it is useful for many of them, the evaluator's ability at elision is plainly not good enough to replace pen and paper examples properly yet. This is not surprising, though, since we curtailed our work on visualization somewhat to attack the central problem of debugging more thoroughly.

However, we believe the problems are solvable, and therefore, that our contention that our design of debugger (both in the form of `OCamli` and `ppx_interpret`) will eventually be useful to the beginner as well as the experienced programmer will turn out to be substantially true.

8.3 Against the research questions and claims

In chapter 1, we wrote down two separate lists. The first list consisted of a number of research questions, decided right at the beginning of the project, and modified since only by a little deletion for time pressure; the text of the remaining entries has not been changed. The second was a list of 'claims', which was written after significant prototyping of the interpreter, but still only part way through the research presented here. It remains unmodified from that time. We summarised the claims as our thesis: *an interpretive debugger for a functional language is technically practicable and might be expected to make debugger use more widespread, routine and productive.* What we shall do now is to look back on those two lists, and discuss to what extent the questions were answered or claims met as the research stands now, and as it may stand in the future. In addition, we shall decide upon new or modified research questions and claims which we can use to take the work forward, and set them in context.

8.3.1 The research questions

Taking the research questions in turn:

> ***It appears that debuggers are not as widely used as one might expect, despite being common for decades. Why?*** *This appears to be doubly true for functional programming languages. How does debugging practice vary among languages (compiled vs interpreted, stateful vs stateless)? What can we learn from debugging theory and practice since the dawn of the computer age?* [original research question]

We gave an account of the surprising lack of widespread use of debuggers in our literature review in chapter 2. We found debugger use did, indeed, seem to be even less widespread in the functional programming community. Many functional programmers never use a debugger, even where the toolchain provides it. We also discovered research over several decades discussing the problem, including much recent work in the Haskell community, which seemed to be very well thought out with regard to the kinds of usability concerns which turn out to be central, we believe, to the lack of use of debuggers.

The difficulty here, of course, is that we might ask "If these problems are so longstanding, why would you believe you can solve them?". The answer, first, is that we do not seek some miraculous, instant solution to the whole field of debugging, but merely a concrete step forward, built on proper foundations. Second, whilst we can never be sure of the efficacy of our approach until the tool is finished and widely used, we can – thanks to the breadth of our literature review – be sure that our thoughts on the nature and character and deficiencies of debugging are not unusual, but shared by many through the years. So, we can be confident we are attacking the right problem, and have some confidence we are attacking it in the right way. Of course, there is a small chance that the dream of the usable, applicable, universal debugger which we described is simply a mirage. But we believe that, given the advances in so much of our field, that much more exploration of debugging is required before giving too much credence to that distressing conclusion. Now that we are moving on to the next stage of this project, to finish the implementation and release a debugger which can be used and tested, we replace this research question with a more practical one, which may be more tangibly evaluated in the future. This replaces our wide questions about the history of debugging with somewhat more narrow ones about the utility of our creation:

> ***Is our debugger suitable for widespread use?*** *Is it used by beginners when they come into contact with the language? Is it used by ordinary working programmers? What proportion of such users? Do they consider it better than existing solutions for debugging OCaml? Than simple solutions like print statements? Do they use it only occasionally, treating debugging as a separate activity, or seamlessly as part of everyday programming activities?* [new research question]

These are the key questions when we come to evaluate, in the round, our final implementation, once it is released and in use. Here is the next original research question:

> ***Can we find a good way to visualize functional program execution?*** *Is the automatic production of such diagrams always going to be inferior to drawing them on paper? How can we deal with scale? How can we show exceptions? What about imperative and mutable features? What are the practicalities of directly interpreting an Abstract Syntax Tree? Can the direct interpretation of the AST of a program ever or always have the same time or space complexity as running the compiled program?* [original research question]

We had, right at the beginning of this piece of research, intended it to be all about the visualization of functional program execution, with a view to debugging. It became clear fairly quickly, though, that the challenge of debugging itself should be central. Thus, we have not done as much work on visualization itself as we might otherwise have. Nonetheless, the OCamli prototype addresses a number of questions of scale, and of the visualization of various tricky aspects like mutable state. For a final implementation of our ideas, especially after practical use, it would be appropriate to revisit this work.

We have explored the efficient implementation of a step-by-step interpreter but stopped well short of proving mathematically that such an interpreter can or cannot preserve the

time and space complexity of the program it is interpreting. When such an interpreter is used for debugging, we believe that reducing the part of the program which need be interpreted (i.e. using ppx_interpret) reduces the immediate concern with speed, by side-stepping it. But a robust final implementation would have to address the issue of efficiency with more formality. What is left unanswered of this research question and what must be altered? We have not adequately addressed in this thesis the time and space complexity. We have discussed it in general terms, but achieved nothing formal. So this must remain. The rest of the research question has been altered to shift the emphasis to the annotation-based debugging interface described in chapter 6.

> ***What are the visualization characteristics of the annotation-based debugger in practice?*** *Is the automatic production of such diagrams always going to be inferior to drawing them on paper? How can we deal with scale in the annotation scenario, including a proper sense of granularity? Can the direct interpretation of the AST of a program ever or always have the same time or space complexity as running the compiled program? In any case, does the selective nature of the annotation-based system obviate or reduce the need for the time and space complexity to be as good as compiled code?* [new research question]

These questions are somewhat related to the usability ones in the rewritten first research question above, but the emphasis on the characteristics of the visualization itself is, we think, worth keeping separate.

> ***Is such an interpreter useful for debugging?*** *Taking into account of the current practice of debugging, is an interpretive debugger better than what is already available? If so, why, and in what ways?* [original research question]

We believe we made a good case that this is (or will be) true in chapter 6. The primary advantages are the accessibility of the debugger – that it is available whenever required, does not need complex setup and is applicable to all programs no matter how compiled and that, in the form of ppx_interpret, it mimics the common "insert print statements" method of debugging. In addition, we have shown how ppx_interpret can encapsulate many techniques from traditional debuggers such as breakpoints and selective printing and interactive debugging. That such flexibility is available in what is, in some ways, quite a novel implementation of a debugger is, we believe, a sign of success. This research question is subsumed into our first rewritten question above, so is deleted. Here is the next original research question:

> ***Is there an alternative abstract machine which might allow for this kind of visualized debugging?*** *That is to say, are we condemned to interpret the AST in the simplest way if we want to be able to properly visualize the evaluation in a human-readable manner? We know that this may have greater complexity of running time than well-known abstract machines. Can we design a bytecode that retains the ability to produce source code for the running computation upon demand, but which is much*

> *faster than brute interpretation and maybe even close to that of a normal bytecode system?* [original research question]

We briefly addressed this in chapter 7 by experimenting with a way to modify OCaml's bytecode interpreter to print steps of execution and found the results interesting but probably unworkable. We discussed abstract machine options in chapter 5. The question needs to be looked at again, especially if we cannot make the debugger efficient enough as a simple interpreter in practice. So we may leave this research question unaltered. Here is the last original research question:

> ***Could we, instead, build an interpreter which can work alongside the native code execution of a program, interpreting only when required?*** *We could compile a program in a slightly different manner. It would run as usual, but when it comes to a part we wish to debug, it would begin interpretation. After this part, it would return to native code, as if nothing had happened.* [original research question]

Indeed we can. And it turned out easier than expected, due to the recently-introduced PPX mechanism in OCaml. We believe it will be the final interface to our debugger. We update the research question to remove the part already completed, and to include more formal criteria for success of the next stage of our work:

> ***Could we, instead, build an interpreter which can work alongside the native code execution of a program, interpreting only when required?*** *Can we show formally (or convincingly) that it is robust with respect to the OCaml runtime, multi-threaded environments, and the use of external linked code? Can we build a system which allows the interpreted section of code to be selected at runtime rather than compile time?* [new research question]

Here are the new research questions in one place:

> ***Is our debugger suitable for widespread use?*** *Is it used by beginners when they come into contact with the language? Is it used by ordinary working programmers? What proportion of such users? Do they consider it better than existing solutions for debugging OCaml? Than simple solutions like print statements? Do they use it only occasionally, treating debugging as a separate activity, or seamlessly as part of everyday programming activities?*

> ***What are the visualization characteristics of the annotation-based debugger in practice?*** *Is the automatic production of such diagrams always going to be inferior to drawing them on paper? How can we deal with scale in the annotation scenario, including a proper sense of granularity? Can the direct interpretation of the AST of a program ever or always have the same time or space complexity as running the compiled program? In any case, does the selective nature of the annotation-based system obviate or reduce the need for the time and space complexity to be as good as compiled code?*

> ***Is there an alternative abstract machine which might allow for this kind of visualized debugging?*** *That is to say, are we condemned to interpret the AST in the simplest way if we want to be able to properly visualize the evaluation in a human-readable manner? We know that this may have greater complexity of running time than well-known abstract machines. Can we design a bytecode that retains the ability to produce source code for the running computation upon demand, but which is much faster than brute interpretation and maybe even close to that of a normal bytecode system?*
>
> ***Could we, instead, build an interpreter which can work alongside the native code execution of a program, interpreting only when required?*** *Can we show formally (or convincingly) that it is robust with respect to the OCaml runtime, multi-threaded environments, and the use of external linked code? Can we build a system which allows the interpreted section of code to be selected at runtime rather than compile time?*

Having discussed the research questions, we move on to our more explicit claims, made after initial exploratory research had been performed.

8.3.2 The claims

We cannot say we have proven these claims, because many of them are social not scientific, only that we have provided some evidence for them.

> *That not many programmers use debuggers, even though they exist.*

The literature review of chapter 2 makes it clear that this is a perennial problem.

> *That (almost) everyone could benefit from a debugger.*

It is harder to make the case for this in definite terms, but it seems to be widely accepted as an assumption by authors we excerpt in the literature review.

> *That the reasons for this disparity are frequently incidental rather than intrinsic and include:*
>
> - *The inability of debuggers working on compiled programs to properly reflect their workings at a source code level.*
> - *The requirement to learn a new tool at exactly the moment one is trying to fix a bug.*

This is largely backed up by the literature review and the results of our programmer survey and others. If we were to rewrite this statement now, with the benefit of hindsight from our widespread reading, we might formulate it differently, emphasising different priorities. And, we cannot say with certainty that those reasons listed are exhaustive. Each time a new debugger is launched on the unsuspecting programming public and they toss it to one side, we learn a new lesson.

> *That many of these barriers melt away in the presence of an interpreter ranking equally to a native compiler, with the same language and toolchain support.*

Perhaps "melt away" is too strong a phrase. But we have managed to produce a design in ppx_interpret which could indeed rank equally to a native compiler, and we expect this to solve (or almost solve) the problems given in the two bullet points above.

> *That the huge disadvantage of slowness which comes with this approach:*
>
> - *Can be ameliorated more that one might expect.*
> - *In any case, is not a show-stopper for most uses, since the external steps of debugging such as case reduction are still in place.*
> - *May be obviated by finding a way to produce mixed native/interpreted programs so that, in any case, the need for interpretation is much reduced.*

As we have already mentioned, it may still be the case that we need a more radical approach to the problem of speed, such as a specialised abstract machine which can display steps of evaluation, rather than a simple interpreter. But, we think that a combination of our mixed native/interpretative model amounts to much the same thing: if the part of the program we interpret is too slow, it is probably producing too much output to read sensibly, so we expect to narrow the bug down further anyway. Again, a final determination will require widespread use.

> *That such an interpretive approach is particularly suited to functional programs due to the mental model of calculation.*

This is a little harder to be certain of: we stopped our visualization research short when we realised that the problem of debugging itself was worth tackling. But personal experience of teaching students using such step-by-step evaluation models is that they are the one of the most useful pedagogical techniques, at least for beginners. And as we have said, and as is supported by the literature review of chapter 2, there is much in common between the beginning programmer and the experienced programmer when tackling a bug.

And so, what of our thesis? *An interpretive debugger for a functional language is technically practicable and might be expected to make debugger use more widespread, routine and productive.* Technically practicable? Yes. Expected to make debugger use more widespread? At least for OCaml, we think so. Routine and productive? Perhaps, but a final determination will have to wait until the system is in use. Or not in use, as the case may be.

What criteria could be used in the future, when the project is further forward? Here are new, updated claims, based upon an understanding and evaluation of the work so far. We would like to use to move this research forward to the next stage, towards a real implementation. We claim:

- that the new OCaml interpreter, OCamli2, when finished:
  - can be shown informally to be correct.
  - can support the whole language.

  - is performant, to the extent which can be expected.

- that the integration of our interpreter into the toolchain:
  - is technically possible.
  - is robust against changes in each version of OCaml, requiring only minimal alterations.

- that the annotation-based interface we have described in chapter 6 provides a natural, malleable interface to debugging functional programs which:
  - is likely to be widely used, compared with existing OCaml debugging tools.
  - suitable for both beginners and experts.
  - subsumes existing techniques such as the insertion of print statements.
  - scales to large programs, including the OCaml compiler itself.

Evaluation of these claims can proceed fully only when we have a working, used implementation.

8.4 Against the literature

In chapter 2 we gave an in-depth review of the debugging literature of the last seventy years. It is instructive in our evaluation to go back to this literature. We shall re-examine the section we entitled *"What makes a good debugger"* to see if we can compare our solution against the rules set by the researchers we drew inspiration from. Of course, we must be careful in this exercise, and make sure it is only one method of evaluation, as Sattherthwaite reminds us:

> *Since debugging, as usually understood, is more a practical than a theoretical problem, proposed solutions must be evaluated within a framework of practical constraints.* [*Satterthwaite, 1972*]

Let us begin with one of the earliest pieces of source material in our literature review, Ira Deihm's contribution to the 1952 ACM national meeting "Computer aids to code checking" [Diehm, 1952]. One of our contentions has been that debugging is an abiding and persistent problem, so it makes sense to go back this far.

> *What one tries to achieve in designing such auxiliary routines is to program the machine to select the pertinent information rather than to read out large quantities of data which must be searched through by the programmer.* [*Diehm, 1952*]

We have already discussed the tasks of reducing the trace output of OCamli by adjusting its default rules, by giving it extra command line flags, and by options which search through the trace explicitly. However, it is far from clear that this will be anywhere near enough to reduce the output of real-world programs in a way which will make debugging as painless as it could be on its own. One of the side-effects of our attempts to deal with the

speed problems of interpretation is that ppx_interpret is also a de facto mechanism for trace reduction – what is not executed by the interpreter is not traced. This, together with ppx_interpret's effective emulation of print-statement-based debugging will, we believe, be a potent combination.

> *The degree of difficulty the programmer experiences in isolating a bug once he has noticed an error depends on the nature of the bug and the ease with which he can obtain additional information about intermediate states in the computation.* [*Gaines, 1969*]

> *In a debugging program it is of prime importance that the program be simple, flexible, and highly efficient to use.* [*Brady, 1968*]

The whole purpose of our system is to give what Gaines describes as "information about intermediate states in the computation". As functional programmers, of course, we do not like calling them states – but even functional programmers think *as if* steps in the evaluation of an expression are states when debugging. If we could solve every bug in a functional program simply by staring at the source code, programmers would need only to locate the buggy function. But in reality, this approach frequently fails and we still need the debugger – to step through the function. The ppx_interpret system allows the programmer to move, change, add, or delete annotations to find further information, and fine-tune the information presented. We are cautiously optimistic that this approach will produce a pliable debugging tool which will fit the debugging workflow well.

Eisenstadt gives some more detailed qualifications a good debugging system must have, in terms of availability to the programmer, a sort of lightness of touch or unobtrusiveness:

- *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.* [*Eisenstadt, 1997*]

And Evans and Darley, from thirty years earlier:

> *. . . a very selective and close control over the execution of portions of one's program and for the examination of intermediate results, together with the possibility of making on-the-spot changes based on them, as desired.* [*Evans and Darley, 1966*]

Our ppx_interpret system will, when it is finished, score on all three of Eisenstadt's criteria. On the first, because it is well-integrated into the toolchain, and can be used in any environment where the compiler is available. On the second and third, our system fulfils these by design, since the source code annotation scheme is the very essence of granularity – the parse tree of a program is a granular structure, and annotations may be attached to any node.

> *...to facilitate maintenance, the same program was to be useable in both batch and interactive modes. Second, to facilitate distribution, the system had to be useable without any modification to the operating system.* [*Grishman, 1970*]

Our debugger is, indeed, independent of the operating system, build system, and all other specifics. If a programmer can install the OCaml toolchain on a system (OCaml's only prerequisites are a C compiler and an implementation of `make`), then they can use the debugger. It demands nothing else. It will be suitable for use with the bytecode or native code compilers, the REPL, or anywhere else OCaml code can be used.

Looking back at these quotations from the literature review, conducted earlier in our research, gives us some reassurance that our system is in line with the aims we set out when beginning the implementation. Research projects involve, of course, experimentation, dead ends, and so on, but that our implementation echoes the literature review shows that we have not veered too far off course.

8.5 Future evaluation

As we have established, the only real test of our approach is whether OCaml programmers use our debugger in preference to others, or in preference to debugging methods involving no debugger at all. However, evaluation is not just useful as a binary test, but as a qualitative process too, shedding a sidelight on the work's qualities, and one can never know what insights will be gained without conducting such an evaluation. In addition, as we have mentioned before, this ultimate test can only apply when the tool is completely finished – earlier evaluations help to guide us towards our goal.

Experimental evaluation

It is worth taking stock of the ways in which other researchers have sought to evaluate debuggers – both in the context of their own work, and in surveys of other debugging tools or mechanisms. We shall look at five papers, already cited in our literature review, and which contain such evaluations, and discuss to what extent these methods may be applied to our own work.

The paper "An Analysis of Patterns of Debugging Among Novice Computer Science Students" [Ahmadzadeh et al., 2005] is an examination of students' mistakes in debugging their own and others' code. A Java compiler is instrumented to report each syntactic or semantic compiler error, together with the source code and time stamp. In the first phase, students are answering set questions by writing (and therefore debugging) their own programs. In the second, more tightly controlled, phase, a single Java program containing both compiler errors and errors of logic only was presented to the students, and they were given a fixed time to correct it. The results are analysed quantitatively by categorisation of the errors and qualitatively to categorise the misconceptions leading to each bug. The primary finding of the second phase of study is, as the authors say, rather surprising: "*... the majority (66%) of the competent debuggers, that is those who were able to correct all three*

logical bugs, are also competent programmers. In contrast, only 39% of the competent programmers were also competent debuggers."

Second, we consider "Debugging: An Analysis of Bug-Location Strategies" [Katz and Anderson, 1987]. This is another study of students (academics having ready access to students seems to make these studies much more common than industrial ones). In this case, the students are not beginners. Another modified compiler, this time for Lisp, is used to feed information on student debugging behaviour back to the researchers. The bugs and debugging strategies are categorised to find the sources of errors and to discover how students debug. The authors then compare students debugging their own programs to the same students debugging programs written by others.

Another paper pertaining to debugging as a part of teaching is "Debugging: The Good, the Bad, and the Quirky – a Qualitative Analysis of Novices' Strategies" [Murphy et al., 2008], which takes a looser approach to setting the experimental parameters. Students were allowed to use any method of debugging, and any online resources they felt they required. The authors point out that, although teachers' interactions with students during lab sessions give plenty of anecdotal evidence about the kind of problems the students face, the actual debugging process is rarely observed. Students were given a program containing logic errors, and the qualitative evaluation took the form of semi-structured interviews with the students afterward, and a one-page survey about the debugging processes used. The data analysis consists of categorisation of the debugging strategies used by the students. The authors conclude that, while students used a variety of debugging strategies with some success, there was a lack of methodical, systematic thinking in the application of the strategy.

The authors of MiraCalc, a step-by-step interpreter for the lazy language Miranda [Turner, 1985] write about an interesting debugging experiment in "A Symbolic Calculator for Non-Strict Functional Programs" [Goldson, 1993]. The hypothesis is that such step-by-step interpreters "can have a positive effect on learning formal subjects". Data on the perceptions of students during their functional programming courses took the form of "questionnaires, video-taped interviews, diaries and e-mail". This was combined with data on their exam results, and the two were analysed. This data is partly qualitative and partly quantitative. The data collected in the first stages was used to refine the questionnaire to "quantify the opinions expressed at interview". This is an example of a relatively large multi-year study, whose methodology might be adapted to our situation.

Finally, an older but very extensive study of debugging in an industrial setting is "Expertise in Debugging Computer Programs" [Vessey, 1985]. Subjects were divided based on ability (expert/novice), and then their debugging strategies evaluated by listening to them speak aloud into a tape recorder as they worked. The program in question was in FORTRAN, with a simple logic error introduced. These tape recordings were then transcribed and analysed. The strategies were compared to the expert/novice categorisation to look for a correlation. The paper is interesting because of its careful approach. For example, subjects debugged example programs at length first, to get used to describing their actions into the tape recorder. However, the authors point out an important limitation of their approach is that the method they used to categorise programmers into novice and

expert is not tested independently of the data. Such concerns are common in these studies, together with the perennial issue of small sample sizes.

It is not yet clear, at this point in development, exactly what type of experiment might be suitable for our system, but we can take inspiration from those we have just described. We hope to test our debugger with some formality, in order to compare our system to others, or to see which kinds of bugs, or which kinds of programmers, it is suitable for.

A teaching test

The author's introductory OCaml textbook [Whitington, 2013] is used by a number of universities as a recommended text for first-year undergraduates. It contains some diagrams of step-by-step evaluations, but not nearly as many as we might like, for space reasons. The book has numerous examples and exercises with answers. Our interpreter OCamli can already produce step-by-step evaluations for almost all the examples and exercises, but they are too verbose, especially at the default settings.

It would be interesting, when our debugger is complete, to produce accompanying material in the form of a portable web-based interactive workbook after the fashion of the popular iPython [Pérez and Granger, 2007] but with step-by-step evaluations. This should be technically feasible, compiling OCaml programs to JavaScript, for example with js_of_ocaml [Vouillon and Balat, 2014]. We could then see how much this added functionality helps learners of OCaml using the textbook, based on self-reporting of both the amount of time spent using the extra resources and how helpful they were found to be, or by reporting from lecturers and supervisors of the help (or lack of help) such an approach gave in teaching and learning.

Against our survey of functional debuggers

It is not really sensible to compare our incomplete system existing to functional debuggers now, because the software is not finished, but we will be able to do so in the future. Each researcher or programmer who created those systems thought, of course, that their design would be widely used, and yet many are not. Our claims about our design being widely used may still, of course, turn out just as hollow. And so whilst we could compare our debugger's features and functionality against those others, this would not really be an evaluation as such. We must wait until our debugger is complete. Once it is, we can evaluate against our literature review of existing debuggers for functional languages in the following ways:

Classification For what users is each debugger intended? For teaching or the working programmer? How is it invoked? What is the interface? Is it batch or interactive?

Accessibility What demands does it make on the build environment or compilation settings? What platforms does it work on? How must it be updated when the compiler toolchain is updated? Is it the default debugger shipped with a compiler toolchain?

Use What is known about how many people use it and how much? Have there been formal studies? Why was it written, according to its authors? What were their criticisms of existing solutions?

In addition, of course, we could perform the same sort of evaluation with regard to debuggers for other idioms – for imperative and object-oriented languages, for example.

8.6 State of the implementations

We are not able to claim that either of our principal efforts are complete, implement the whole OCaml language, or that they are anything other than flawed prototypes. Our first such prototype OCamli, described in chapter 4, was written to provide enough coverage of the language to allow experimentation with a wide variety of programs (including loading the OCaml Standard Library). It served as a testbed for implementing visualization elision mechanisms, and more traditional debugging tools such as stepping. It is now mothballed in favour of the second implementation for the reasons we described in chapters 5 and 6. This second implementation is only at the proof-of-concept stage. The OCamli2 implementation from chapter 5 can only handle small, simple programs. The annotation-based interface described in chapter 6 is a design only, with but a tiny working example. The work described in chapter 7 is, as we said at the time, abortive.

The idea is to finish the second system in two stages: first by making it narrow but deep – that is to say covering a small part of the language, but fully integrating the mixed compilation/interpretation model described in chapter 6. Then, once the technical implementation is secure, we can expand it to the full language. Finally, we can return to the question of better visualization of the output, in particular with regard to formatting and elision. Taken together, we believe that real-world usage, such as debugging the OCaml compiler itself, is not imminent, but we have shown it to be technically feasible.

We believe that, though we have exhibited only early implementations and prototypes thus far, enough insight has been gained that we may be sure of a radically different (and, we hope, successful) OCaml debugger which has a good chance at widespread adoption.

The version of our original OCamli prototype, described in [Whitington and Ridge, 2019] is available on GitHub [Whitington and Ridge, 2017a].

8.7 Summary

We have attempted to evaluate our work whilst it is still in progress. To the extent that it is possible to say, we are confident that OCamli alone is not sufficient to genuinely advance the state of the art in debugging, interesting though it is. We are optimistic that ppx_interpret, when fully implemented, will represent such an advancement.

Chapter 9

Conclusion

> *If you want a happy ending, that depends, of course, on where you stop your story.*
>
> — Orson Welles

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 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 too old and too intransigent to allow us too much confidence.

We have begun the process of evaluating the design insofar as it can be evaluated in the absence of a full, widely-used implementation. 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 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 made initial efforts to preserve the time and space complexity of programs under interpretation, even when the intermediate steps are not shown – but no theoretical basis or proper proof has been provided. Is it possible to interpret programs in such a way that we can give a guarantee about the time and space complexity?

There are questions of interface too. In our prototype, the user has to choose which part of the program is to be interpreted by annotating the source file, either manually or automatically through an IDE. Could a system be devised where all the parts of the program are available both in compiled and interpreted form at all times, and interpretation can be switched on and off during the debugging session? Can we automatically insert all breakpoint annotations and choose between them at run-time not compile-time? Zhang et

al. have devised a system [Zhang et al., 2013] for automatic breakpoint generation which may be applicable, with some modifications, to the functional realm.

The kind of diagrams our interpreter draws are also useful for teaching, that is to say for testing 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 step-by-step interpreter, in either or both of its guises, helps beginning students. Besides teaching and debugging, having an 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.

Appendix A

An OCaml primer

OCaml is a statically-typed strict functional language, with optional imperative features.

Simple data types

We have the integers `min_int` ... `-3 -2 -1 0 1 2 3` ... `max_int` of type **int**. The booleans are `true` and `false` of type **bool**. Characters have type **char** and are written like `'X'` and `'!'`.

The mathematical operators `+ - * / mod` take integers and give another. Note that they do not work on floating-point numbers: we have other operators for those.

```
        6 * 2
⟹      12
```

The comparison operators `= < <= > >= <>` which compare two values of like type (except functions) and evaluate to either `true` or `false`.

```
        1 + 2 + 3 = 1 * 2 * 3
⟹      true
```

The conditional expression is **`if`** *expression1* **`then`** *expression2* **`else`** *expression3*, where *expresssion1* has type **bool** and *expression2* and *expression3* have the same type as one another.

```
        if 4 * 3 > 2 * 2 then 1 else 0
⟹      1
```

The boolean operators `&&` (logical AND) and `||` (logical OR) allow us to build compound boolean expressions. They are short-circuiting (evaluating their right-hand side only when needed).

```
        1 = 2 || 2 = 2
⟹      true
```

Tuples combine a fixed number of elements `(a, b)`, `(a, b, c)` etc. with types $\alpha \times \beta$, $\alpha \times \beta \times \gamma$ etc. For example, `(1, '1')` is a tuple of type **int** $\times$ **char**. On the screen, OCaml writes `'a` for α, `'b` for β etc.

Strings are sequences of characters written between double quotes and are of type **string**. For example, `"one"` has type **string**.

Names and functions

We can assign a name to the result of evaluating an expression using the **`let`** *name* `=` *expression* construct.

```
let x = 5 > 2                    x is true
```

We can build compound expressions using

let *name1* = *expression1* **in let** *name2* = *expression2* **in** ...

```
let x = 4 in let y = 5 in x + y
```

Functions can be introduced by **let** *name argument1 argument2* ... = *expression*. These have type $\alpha \to \beta$, $\alpha \to \beta \to \gamma$ etc. for some types α, β, γ etc. For example, `let f a b = a > b` is a function of type $\alpha \to \alpha \to$ **bool**.

Recursive functions are introduced in the same way, but using **let rec** instead of **let**. For example, here is a function `g` which calculates the smallest power of two greater than or equal to a given positive integer, using the recursive function `f`:

```
let rec f x y =
  if y < x then f x (2 * y) else y

let g z = f z 1
```

Mutually recursive functions are introduced by writing `let rec f x = ... and g y = ... and ...`

Anonymous (un-named) functions can be defined like this: **fun** *name* -> *expression*.

```
          (fun x -> x * 2) 4
⟹    8
```

We can make operators into functions using parentheses, for example `( < )` and `( + )`.

```
( + ) 1 2 ⟹ 3
```

Pattern matching

The expression **match** *expression1* **with** *pattern1* | ... -> *expression2* | *pattern2* | ... -> *expression3* | ... matches an expression against a number of patterns in turn, choosing the result expression whose pattern first matches. The expressions *expression2*, *expression3* etc. must have the same type as one another, and this is the type of the whole **match** ... **with** expression. The special pattern _ matches anything.

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

We may match two or more things at once, using commas to separate as in **match** `a, b with 0, 0 ->` *expression1* `| x, y ->` *expression2* `|` ...

```
match x, y, z with
  0, 0, 0 -> true
| _, _, _ -> false
```

Lists

Lists are ordered collections of zero or more elements of like type. They are written between square brackets, with elements separated by semicolons e.g. `[1; 2; 3; 4; 5]`. If a list is non-empty, it has a head, which is its first element, and a tail, which is the list composed of the rest of the elements.

The `::` 'cons' operator adds an element to the front of a list. The `@` 'append' operator concatenates two lists together.

```
1 :: [2; 3] ⟹ [1; 2; 3]
[1; 2] @ [3] ⟹ [1; 2; 3]
```

Lists and the `::` 'cons' symbol may be used for pattern matching to distinguish lists of length zero, one, etc. and with particular contents. For example, we can calculate the length of a list:

```
let rec length l =
  match l with
    [] -> 0
  | _::t -> 1 + length t
```

Exceptions

Exceptions are defined with **exception** *name*. They can carry extra information by adding **of** *type*. Exceptions are raised with **raise**, and handled with **try** ... **with** ...

```
exception Problem of int

let f x y =
  if y = 0
    then raise (Problem x)
    else x / y

let g x y =
  try f x y with Problem p -> p
```

Partial application

Functions may be partially applied by giving fewer than the full number of arguments.

```
let add x y = x + y

List.map (add 3) [1; 2; 3]
⟹ [4; 5; 6]

List.map (( + ) 3) [1; 2; 3]
⟹ [4; 5; 6]
```

New data types

New types are introduced with **type** *name* = *constructor1* **of** *type1* | *constructor2* **of** *type2* | ... We may pattern-match on them just as on the built-in types.

```
type colour =
  Red | Blue | Green | Grey of int

[Red; Blue; Grey 16]      type colour list

type 'a tree =            a polymorphic type
   Lf
 | Br of 'a tree * 'a * 'a tree
```

For example, `Br (Lf, 'X', Br (Lf, 'Y', Lf))` has type **char** `tree`. A useful built-in data type is the **option** type, defined as **type** `'a option = None | Some of 'a`. A type can be polymorphic in more than one type parameter, for example `('a, 'b) Hashtbl.t`, as in the Standard Library.

Basic input / output

The value `()` has type **unit**. Input channels of type **in_channel** and output channels of type **out_channel** are available. Built-in functions such as `open_in`, `close_in`, `open_out`, `close_out`, `input_char`, `output_char` etc. exist for reading from and writing to them respectively.

Mutable state

References of type α **ref** are mutable cells containing values. They are built using `ref`, their contents is accessed using `!` and they are updated by using the `:=` operator.

```
# let p = ref 0;;
val p : int ref = {contents = 0}
# p := 5;;
- : unit = ()
# !p;;
- : int = 5
```

Arrays of type α **array** are written like `[|1; 2; 3|]`. An array is created with the built-in function `Array.make`. Elements are accessed with `a.`(*subscript*), and updated with `a.`(*subscript*) `<-` *expression*.

```
let swap a x y =
  let t = a.(x) in
    a.(x) <- a.(y); a.(y) <- t
```

An action may be performed many times based on a boolean condition with the **while**

boolean expression **do** *expression* **done** construct.

```
while !x < y do x := !x * 2 done
```

Performing an action a fixed number of times with a varying parameter is achieved using the **for** *name* = *start* **to** *end* **do** *expression* **done** construct.

```
for x = 1 to 10 do print_int x done
```

Floating-point numbers

Floating-point numbers are written `min_float` ... `max_float` and have type **float**. Floating-point operators `+. *. -. /. **` and built-in functions `sqrt` `log` etc. are available.

```
2. ** 0.2 ⟹ 1.1486983549970351
```

The OCaml Standard Library

Functions from the OCaml Standard Library are used with the form *Module*.*function*. For example, `List.map`, `String.length`, `Array.copy` etc.

Simple modules

Modules are written in `.ml` files. Corresponding interfaces live in `.mli` files. For example, the `.ml` file with contents **let** `f x = x + 1` might have the interface **val** `f : int -> int`

Compiling programs

In addition to the `ocaml` REPL, OCaml has two compilers, `ocamlc` for bytecode, and `ocamlopt` for native code. For example:

`ocamlc -o x x.ml` builds `x` (or `x.exe`) from `x.ml` with the bytecode compiler.

`ocamlopt -o x x.ml` builds `x` (or `x.exe`) from `x.ml` with the native code compiler.

Appendix B

Test programs

`donothing.ml`

This program does nothing, consisting only of a single value which is immediately returned and ignored. Its purpose is to provide a baseline for the time and space usage of each compiler or interpreter.

```
1
```

`factorial.ml`

Calculates a factorial. The *<count>* in this and other examples is the input number n in our timing tables. It is used to ensure an appropriate running time for accurate measurement.

```
let rec factorial n =
  if n = 1 then 1 else n * factorial (n - 1)
in
  factorial <count>
```

`factorialacc.ml`

An iterative, or accumlative version of the factorial function. This requires only constant stack space with an ordinary compiler.

```
let rec factorial a n =
  if n = 1 then a else factorial (a * n) (n - 1)
in
  factorial 1 <count>
```

helloworld.ml

Prints a string to the screen a number of times.

```
for x = 0 to <count> do print_string "Hello, World!\n" done
```

reference_swap.ml

Creates and swaps the values of two references (mutable cells).

```
let swap () =
  let x = ref 0 in
    let y = ref 1 in
      let t = !x in
        x := !y;
        y := t

let _ = for x = 0 to <count> do swap () done
```

exception.ml

Raises and catches an exception.

```
for x = 1 to <count> do
  begin try raise Exit with Exit -> 4 end
done
```

table.ml

Prints a times-table. We multiply the counts to reach our input size n.

```
for x = 1 to <count> do
  for y = 1 to <count> do
    print_int (x * y);
    print_string "\t"
  done;
  print_string "\n"
done
```

tree.ml

Defines a new data type for binary trees, and functions to insert a single item and multiple items, and exercises them. The order of the data is so as to lead to a relatively balanced tree: the pathological case for the algorithm is thus avoided.

```
type 'a tree = Lf | Br of 'a tree * 'a * 'a tree

let rec insert t i =
  match t with
    Lf -> Br (Lf, i, Lf)
  | Br (l, i', r) ->
      if i < i' then Br (insert l i, i', r) else
      if i > i' then Br (l, i', insert r i) else
        Br (l, i, r)

let rec insert_many t vs =
  match vs with [] -> t
  | x::xs -> insert_many (insert t x) xs

let _ =
  for x = 1 to <count> do
    let t =
      insert_many Lf
        [29; 34; 71; 15; 100; 46; 66; 70; 92; 20; 37; 29;
         26; 84; 77; 100; 3; 63; 73; 52; 36; 99; 30; 46;
         13; 67; 79; 85; 6; 31; 73; 27; 94; 92; 63; 93;
         49; 6; 39; 3; 10; 32; 26; 83; 97; 44; 90; 65; 55;
         36; 90; 48; 38; 96; 46; 38; 70; 81; 63; 10; 67;
         82; 81; 6; 74; 41; 69; 57; 10; 31; 28; 87; 77; 92;
         90; 35; 12; 8; 37; 43; 68; 58; 74; 49; 52; 61;
         100; 63; 72; 65; 55; 56; 31; 35; 86; 93; 82; 50;
         39; 22]
    in
      ()
    done
```

Bibliography

[Ager et al., 2003] Ager, M. S., Biernacki, D., Danvy, O., and Midtgaard, J. (2003). A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN international conference on principles and practice of declarative programming*, pages 8–19. ACM.

[Ahmadzadeh et al., 2005] Ahmadzadeh, M., Elliman, D., and Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin*, 37(3):84–88.

[Alsallakh et al., 2012] Alsallakh, B., Bodesinsky, P., Gruber, A., and Miksch, S. (2012). Visual tracing for the eclipse java debugger. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 545–548. IEEE.

[Auguston and Reinfields, 1994] Auguston, M. and Reinfields, J. (1994). A Visual Miranda Machine. In *Software Education Conference, 1994. Proceedings.*, pages 198–203. IEEE.

[Badiozamany and Wang, 2010] Badiozamany, S. and Wang, H. (2010). Debugging: the difference between novices and experts. Technical report, Uppsala University.

[Balzer, 1969] Balzer, R. (1969). EXDAMS – Extendable Debugging And Monitoring System. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, Memorandum, pages 567–580. ACM.

[Brady, 1968] Brady, P. T. (1968). Writing an online debugging program for the experienced user. *Communications of the ACM*, 11(6):423–427.

[Brooks, 1995] Brooks, Jr., F. P. (1995). *The Mythical Man-month (Anniversary Ed)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Brown, 1965] Brown, W. (1965). An operating environment for dynamic-recursive computer programming systems. *Communications of the ACM*, 8(6):371–377.

[Burstall et al., 1980] Burstall, R. M., MacQueen, D. B., and Sannella, D. T. (1980). HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143.

[Charguéraud, 2015] Charguéraud, A. (2015). Improving type error messages in OCaml. *arXiv preprint arXiv:1512.01897*.

[Charguéraud et al., 2018] Charguéraud, A., Schmitt, A., and Wood, T. (2018). JSExplain: A double debugger for JavaScript. In *The Web Conference 2018*, pages 1–9.

[Chase and Hood, 1987] Chase, B. B. and Hood, R. T. (1987). Selective interpretation as a technique for debugging computationally intensive programs. In *ACM SIGPLAN Notices*, volume 22, pages 113–124.

[Cheng, 2003] Cheng, H. (2003). *The Ch Language Environment User's Guide. Revision 3.7.* SoftIntegration, Inc.

[Cheng, 2010] Cheng, H. (2010). *C For Engineers & Scientists, An Interpretive Approach with Companion CD*. McGraw-Hill, Inc., New York, NY, USA, 1st edition.

[Chitil et al., 2002] Chitil, O., Runciman, C., and Wallace, M. (2002). Transforming Haskell for tracing. In *Symposium on Implementation and Application of Functional Languages*, pages 165–181. Springer.

[Clements et al., 2001] Clements, J., Flatt, M., and Felleisen, M. (2001). Modeling an algebraic stepper. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, pages 320–334, London, UK. Springer-Verlag.

[Cong and Asai, 2016] Cong, Y. and Asai, K. (2016). Implementing a stepper using delimited continuations. *Software Science*, 39:42–54.

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

[Crew et al., 1997] Crew, R. F. et al. (1997). ASTLOG: A language for examining abstract syntax trees. In *Conference on Domain-Specific Languages*, volume 97, page 18.

[CSharp, 2012] CSharp (2012). `https://www.mono-project.com/docs/tools+libraries/tools/repl/`.

[Danvy, 2004] Danvy, O. (2004). A rational deconstruction of Landin's SECD machine. In *Symposium on Implementation and Application of Functional Languages*, pages 52–71. Springer.

[Devanbu, 1999] Devanbu, P. T. (1999). GENOA – a customizable, front-end-retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, 8(2):177–212.

[Diehm, 1952] Diehm, I. C. (1952). Computer aids to code checking. In *Proceedings of the 1952 ACM national meeting (Toronto)*, pages 19–20.

[Digital Equipment Corporation, 1967] Digital Equipment Corporation (1967). *DDT Prorgramming Manual*.

[Dijkstra, 1971] Dijkstra, E. W. (1971). On the reliability of programs. `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html`.

[Dijkstra, 1972] Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10):859–866.

[Ducassé, 1993] Ducassé, M. (1993). A pragmatic survey of automated debugging. In *International Workshop on Automated and Algorithmic Debugging*, pages 1–15. Springer.

[Ducassé and Emde, 1988] Ducassé, M. and Emde, A.-M. (1988). A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th international conference on Software engineering*, pages 162–171. IEEE Computer Society Press.

[Eisenstadt, 1997] Eisenstadt, M. (1997). My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37.

[Evans and Darley, 1966] Evans, T. G. and Darley, D. L. (1966). On-line debugging techniques: a survey. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, pages 37–50. ACM.

[F Sharp, 2016] F Sharp (2016). Visual Studio docs: Debugging F#. `https://docs.microsoft.com/en-us/visualstudio/debugger/debugging-f-hash`.

[Felleisen et al., 2015] Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., and Tobin-Hochstadt, S. (2015). The Racket manifesto. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[Finder et al., 2002] Finder, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. (2002). DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming*, 12(2):159–182.

[Fitzgerald et al., 2010] Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., and Zander, C. (2010). Debugging from the student perspective. *IEEE Transactions on Education*, 53(3):390–396.

[Foubister, 1995] Foubister, S. P. (1995). *Graphical application and visualization of lazy functional computation.* PhD thesis, University of York.

[Freund and Roberts, 1996] Freund, S. N. and Roberts, E. S. (1996). Thetis: an ANSI C programming environment designed for introductory use. In *SIGCSE*, volume 96, pages 300–304.

[Furukawa et al., 2019] Furukawa, T., Cong, Y., and Asai, K. (2019). Stepping OCaml. *CoRR*, abs/1906.11422.

[Gaines, 1969] Gaines, R. S. (1969). *The Debugging of Computer Programs*. PhD thesis, Princeton University, Princeton, NJ, USA.

[Gestwicki and Jayaraman, 2002] Gestwicki, P. and Jayaraman, B. (2002). Interactive visualization of Java programs. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 226–235. IEEE.

[Gestwicki and Jayaraman, 2004] Gestwicki, P. V. and Jayaraman, B. (2004). JIVE: Java interactive visualization environment. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 226–228.

[Gill, 2000] Gill, A. (2000). Debugging Haskell by observing intermediate data structures. *Electronic Notes in Theoretical Computer Science*, 41(1):1.

[Gill, 1951] Gill, S. (1951). The diagnosis of mistakes in programmes on the EDSAC. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 206(1087):538–554.

[Go Format, 2019] Go Format (2019). `https://golang.org/cmd/gofmt/`.

[Goldson, 1993] Goldson, D. (1993). A symbolic calculator for non-strict functional programs. *The Computer Journal*, 37(3):177–187.

[Gough et al., 1994] Gough, K., Ledermann, J., and Elms, K. (1994). Interpretive debugging of optimised code. In *Proceedings of ACSC-17, Christchurch*.

[Gould, 1975] Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7(2):151IN1171–170IN2182.

[Grishman, 1970] Grishman, R. (1970). The debugging system AIDS. In *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, pages 59–64. ACM.

[Griswold et al., 1996] Griswold, W. G., Atkinson, D. C., and McCurdy, C. (1996). Fast, flexible syntactic pattern matching and processing. In *Proceedings of the Fourth Workshop on Program Comprehension*, pages 144–153. IEEE.

[Hailpern and Santhanam, 2002] Hailpern, B. and Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12.

[Hall and O'Donnell, 1985] Hall, C. V. and O'Donnell, J. T. (1985). Debugging in a side effect free programming environment. *ACM SIGPLAN Notices*, 20(7):60–68.

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

[Halpern, 2005] Halpern, M. (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):28–35.

[Hamlet, 1983] Hamlet, D. (1983). Debugging 'level': step-wise debugging. *SIGPLAN Notices*, 18(8):4–8.

[Harper et al., 1986] Harper, R., MacQueen, D., and Milner, R. (1986). *Standard ML*. Department of Computer Science, University of Edinburgh.

[Harris, 2002] Harris, T. L. (2002). Dependable software needs pervasive debugging. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 38–43. ACM.

[Hoffmann and O'Donnell, 1979] Hoffmann, C. M. and O'Donnell, M. J. (1979). An interpreter generator using tree pattern matching. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 169–179.

[Hoffmann and O'Donnell, 1982] Hoffmann, C. M. and O'Donnell, M. J. (1982). Pattern matching in trees. *Journal of the ACM*, 29(1):68–95.

[Huber, 2010] Huber, T. (2010). An introduction to C and Ch: Your one-stop shop for scientific computing. *Computing in Science Engineering*, 12(4):7–11.

[Humphrey, 1988] Humphrey, W. S. (1988). Characterizing the software process: a maturity framework. *IEEE software*, 5(2):73–79.

[IEEE, 1990] IEEE (1990). *610.2 - IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*.

[Johnson, 1982] Johnson, M. S. (1982). A software debugging glossary. *ACM SIGPLAN Notices*, 17(2):53–70.

[Katz and Anderson, 1987] Katz, I. R. and Anderson, J. R. (1987). Debugging: an analysis of bug-location strategies. *Human-Computer Interaction*, 3(4):351–399.

[Kernighan and Plauger, 1978] Kernighan, B. W. and Plauger, P. (1978). *The Elements of Programming Style*. McGraw-Hill, 2nd edition.

[Kiselyov, 2014] Kiselyov, O. (2014). The design and implementation of BER MetaOCaml. In *International Symposium on Functional and Logic Programming*, pages 86–102. Springer.

[Knuth, 1989] Knuth, D. E. (1989). The errors of TEX. *Software: Practice and Experience*, 19(7):607–685.

[Knuth, 1997] Knuth, D. E. (1997). *The Art of Computer Programming*, volume 1. Addison-Wesley Professional.

[Ko et al., 2011] Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., et al. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)*, 43(3):21.

[Ko and Myers, 2003] Ko, A. J. and Myers, B. A. (2003). Development and evaluation of a model of programming errors. In *Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium on*, pages 7–14.

[Lachwani and Srinivasan, 2015] Lachwani, M. and Srinivasan, J. (2015). Remote application debugging. US Patent 9,170,922.

[Laffra and Malhotra, 1994] Laffra, C. and Malhotra, A. (1994). HotWire – A Visual Debugger for C++. In *C++ Conference*, pages 109–122.

[LaToza et al., 2006] LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM.

[Le Bon and Schmitt, 2018] Le Bon, K. and Schmitt, A. (2018). MLExplain. In *OCaml 2018*.

[Le Fessant and Chambart, 2015] Le Fessant, F. and Chambart, P. (2015). Towards a debugger for native-code OCaml applications. *OCaml Users and Developers Workshop*.

[Lerner et al., 2006] Lerner, B., Grossman, D., and Chambers, C. (2006). Seminal: Searching for ml type-error messages. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 63–73, New York, NY, USA. ACM.

[Leroy, 1990] Leroy, X. (1990). *The ZINC experiment: an economical implementation of the ML language*. PhD thesis, INRIA.

[Leroy, 2015] Leroy, X. (2015). Functional programming languages. Part II: Abstract machines. `https://xavierleroy.org/mpri/2-4/`.

[Leroy et al., 2018] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. (2018). The OCaml Language. `https://ocaml.org/`.

[Lieberman, 1997] Lieberman, H. (1997). The debugging scandal and what to do about it. *Communications of the ACM*, 40(4):26–30.

[Maranget, 2008] Maranget, L. (2008). Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, pages 35–46, New York, NY, USA.

[Marlow et al., 2007] Marlow, S., Iborra, J., Pope, B., and Gill, A. (2007). A Lightweight Interactive Debugger for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 13–24, New York, NY, USA.

[McCauley et al., 2008] McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., and Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92.

[Meijer and Miller, 2012] Meijer, E. and Miller, J. (2012). Technical overview of the common language runtime. `http://www.cin.ufpe.br/~haskell/papers/Technical_Overview_of_the_Common_Language_Runtime-Meijer&Miller.pdf`.

[Metzger, 2004] Metzger, R. C. (2004). *Debugging by thinking: A multidisciplinary approach*. Digital Press.

[Meurer, 2010] Meurer, B. (2010). OCamlJIT 2.0 – Faster Objective Caml. *arXiv preprint arXiv:1011.1783*.

[Microsoft, 2020] Microsoft (2020). Visual Studio Code. `https://code.visualstudio.com`.

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

[Minsky et al., 2013] Minsky, Y., Madhavapeddy, A., and Hickey, J. (2013). *Real World OCaml: Functional programming for the masses*. O'Reilly Media, Inc.

[Murphy et al., 2008] Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., and Zander, C. (2008). Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. *SIGCSE Bulletin*, 40(1):163–167.

[Myers, 1979] Myers, G. J. (1979). *The art of software testing*. John Wiley & Sons.

[OCaml Autoformatter, 2019] OCaml Autoformatter (2019). `https://github.com/ocaml-ppx/ocamlformat`.

[ocaml.org debugging, 2019] ocaml.org debugging (2019). `https://ocaml.org/learn/tutorials/debug.html`.

[Owens, 2008] Owens, S. (2008). A sound semantics for ocaml light. In *European Symposium on Programming*, pages 1–15. Springer.

[Pareja-Flores et al., 2007] Pareja-Flores, C., Urquiza-Fuentes, J., and Velázquez-Iturbide, J. Á. (2007). WinHIPE: An IDE for Functional Programming Based on Rewriting and Visualization. *ACM SIGPLAN Notices*, 42(3):14–23.

[Parnin and Orso, 2011] Parnin, C. and Orso, A. (2011). Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA. ACM.

[Patel et al., 1997] Patel, M. J., Du Boulay, B., and Taylor, C. (1997). Comparison of contrasting prolog trace output formats. *International Journal of Human-Computer Studies*, 47(2):289–322.

[Paul and Prakash, 1994] Paul, S. and Prakash, A. (1994). A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475.

[Pavlinovic et al., 2014] Pavlinovic, Z., King, T., and Wies, T. (2014). Finding minimum type error sources. *SIGPLAN Notices*, 49(10):525–542.

[Pea, 1986] Pea, R. D. (1986). Language-independent conceptual 'bugs' in novice programming. *Journal of Educational Computing Research*, 2(1):25–36.

[Penney, 2000] Penney, A. W. (2000). *Augmenting Trace-based Functional Debugging*. PhD thesis, University of Bristol.

[Perera et al., 2012] Perera, R., Acar, U. A., Cheney, J., and Levy, P. B. (2012). Functional programs that explain their work. *ACM SIGPLAN Notices*, 47(9):365–376.

[Pérez and Granger, 2007] Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29.

[Petre and de Quincey, 2006] Petre, M. and de Quincey, E. (2006). A Gentle Overview of Software Visualization. *PPIG Newsletter*, pages 1–10.

[Ranum, 2007] Ranum (2007). Teaching an old dog new tricks: The problem is complexity. `https://www.ranum.com/security/computer_security/editorials/codetools/`.

[Reeves et al., 1994] Reeves, S., Goldson, D., Fung, P., Hopkins, M., and Bornat, R. (1994). The Calculator project – formal reasoning about programs. In *Software Education Conference, 1994. Proceedings.*, pages 166–173. IEEE.

[Regelson and Anderson, 1994] Regelson, E. and Anderson, A. (1994). Debugging practices for complex legacy software systems. In *ICSM*, pages 137–143.

[Reiss, 2005] Reiss, S. P. (2005). The paradox of software visualization. In *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on*, pages 1–5.

[Robillard et al., 2004] Robillard, M. P., Coelho, W., and Murphy, G. C. (2004). How effective developers investigate source code: An exploratory study. *IEEE Transactions on software engineering*, 30(12):889–903.

[Sandewall, 1978] Sandewall, E. (1978). Programming in an interactive environment: the LISP experience. *ACM Computing Surveys (CSUR)*, 10(1):35–71.

[Satterthwaite, 1972] Satterthwaite, E. (1972). Debugging tools for high level languages. *Software: Practice and Experience*, 2(3):197–217.

[Schach, 1996] Schach, S. R. (1996). Testing: principles and practice. *ACM Computing Surveys (CSUR)*, 28(1):277–279.

[Scowen, 1972] Scowen, R. (1972). Debugging computer programs: a survey with special emphasis on ALGOL. Technical report, DTIC.

[Shah et al., 2008] Shah, H., Görg, C., and Harrold, M. J. (2008). Visualization of Exception Handling Constructs to Support Program Understanding. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 19–28, New York, NY, USA.

[Shinwell, 2014] Shinwell, M. (2014). libmonda: make OCaml native debugging awesome. `https://mshinwell.github.io/libmonda/`.

[Siegmund et al., 2014] Siegmund, B., Perscheid, M., Taeumel, M., and Hirschfeld, R. (2014). Studying the advancement in debugging practice of professional software developers. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 269–274.

[Sivaramakrishnan, 2020] Sivaramakrishnan, K. (2020). Sandmark – a benchmarking suite for OCaml. `https://github.com/ocaml-bench/sandmark`.

[Software Preservation Group, 2017] Software Preservation Group (2017). Saber-C / CodeCenter. http://www.softwarepreservation.org/projects/interactive_c/saberc/saberc.

[Sorva et al., 2013] Sorva, J., Karavirta, V., and Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *Transactions in Computing Education*, 13(4):15:1–15:64.

[Stallman et al., 2002] Stallman, R., Pesch, R., Shebs, S., et al. (2002). Debugging with gdb. *Free Software Foundation*, 51:02110–1301.

[Stallman, 1981] Stallman, R. M. (1981). *EMACS: the extensible, customizable self-documenting display editor*. ACM Books.

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

[Syme, 2012] Syme, D. (2012). F Sharp at Microsoft Research. https://www.microsoft.com/en-us/research/project/f-at-microsoft-research/.

[Taylor, 1996] Taylor, J. P. (1996). *Presenting the Lazy Evaluation of Functions*. PhD thesis, Queen Mary, University of London.

[Tolmach and Appel, 1995] Tolmach, A. and Appel, A. W. (1995). A Debugger for Standard ML. *Journal of Functional Programming*, 5:155–200.

[Torvalds, 2000] Torvalds, L. (2000). Re: Availability of kdb (Linux Kernel Mailing List). https://lkml.org/lkml/2000/9/6/65.

[Touretzky, 1989] Touretzky, D. S. (1989). Visualizing Evaluation in Applicative Languages. *Communications of the ACM*, 35(10):49–59.

[Turner, 1985] Turner, D. A. (1985). Miranda: A non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer.

[Ungar et al., 1997] Ungar, D., Lieberman, H., and Fry, C. (1997). Debugging and the Experience of Immediacy. *Communications of the ACM*, 40(4):38–43.

[Urquiza-Fuentes and Velázquez-Iturbide, 2004] Urquiza-Fuentes, J. and Velázquez-Iturbide, J. A. (2004). A Survey of Program Visualizations for the Functional Paradigm. In *Proc. 3rd Program Visualization Workshop*, pages 2–9.

[Urquiza-Fuentes and Velázquez-Iturbide, 2009] Urquiza-Fuentes, J. and Velázquez-Iturbide, J. Á. (2009). A Survey of Successful Evaluations of Program Visualisation and Algorithm Animation Systems. *ACM Transactions on Computing Education (TOCE)*, 9(2):9.

[Vasilev et al., 2012] Vasilev, V., Canal, P., Naumann, A., and Russo, P. (2012). Cling – the new interactive interpreter for ROOT 6. In *Journal of Physics*, volume 396 of *Conference Series*. IOP Publishing.

[Vessey, 1985] Vessey, I. (1985). Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494.

[Vouillon and Balat, 2014] Vouillon, J. and Balat, V. (2014). From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972.

[Wadler, 1998] Wadler, P. (1998). Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27.

[Watson and Salzman, 1997] Watson, R. and Salzman, E. (1997). A trace browser for a lazy functional language. *Australian Computer Science Communications*, 19:356–363.

[Watt, 1994] Watt, S. (1994). Froglet: a source-level stepper for LISP. *Human Cognition Research Laboratory, Open University, Milton Keynes, England*.

[Weiser, 1982] Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452.

[Whitington, 2013] Whitington, J. (2013). *OCaml from the Very Beginning*. Coherent Press.

[Whitington and Ridge, 2017a] Whitington, J. and Ridge, T. (2017a). The OCamli Interpreter.

[Whitington and Ridge, 2017b] Whitington, J. and Ridge, T. (2017b). Visualizing the evaluation of functional programs for debugging. In *6th Symposium on Languages, Applications and Technologies*, volume 56 of *OASIcs*, pages 7:1–7:9, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Whitington and Ridge, 2019] Whitington, J. and Ridge, T. (2019). Direct interpretation of functional programs for debugging. In Lindley, S. and Scherer, G., editors, Proceedings *ML Family / OCaml Users and Developers workshops*, Oxford, UK, 7th September 2017, volume 294 of *Electronic Proceedings in Theoretical Computer Science*, pages 41–73. Open Publishing Association.

[Wilkes, 1949] Wilkes, M. (1949). Programme design for a high-speed automatic calculating machine. *Journal of Scientific Instruments*, 26(6):217.

[Wilkes, 1985] Wilkes, M. (1985). *Memoirs of a Computer Pioneer*. Massachusetts Institute of Technology, Cambridge, MA, USA.

[Wilkes, 1951] Wilkes, M. V. (1951). *The preparation of programs for an electronic digital computer: with special reference to the EDSAC and the use of a library of subroutines*. Addison-Wesley Press.

[Wilkes and Renwick, 1949] Wilkes, M. V. and Renwick, W. (1949). The EDSAC – an electronic calculating machine. *Journal of Scientific Instruments*, 26(12):385.

[Yallop et al., 2017] Yallop, J., Sheets, D., and Madhavapeddy, A. (2017). A modular foreign function interface. *Science of Computer Programming*.

[Zeller, 2009a] Zeller, A. (2009a). Debugging debugging. *ESEC-FSE August 2009*.

[Zeller, 2009b] Zeller, A. (2009b). *Why programs fail: a guide to systematic debugging*. Elsevier.

[Zeller et al., 2000] Zeller, A. et al. (2000). Debugging with DDD. *User's Guide and Reference Manual, Version*, 3.

[Zeller and Lütkehaus, 1996] Zeller, A. and Lütkehaus, D. (1996). DDD – a free graphical front-end for UNIX debuggers. *ACM Sigplan Notices*, 31(1):22–27.

[Zhang et al., 2013] Zhang, C., Yang, J., Yan, D., Yang, S., and Chen, Y. (2013). Automated breakpoint generation for debugging. *JSW*, 8(3):603–616.

Index