

Chapter 1

Introduction

My project concerns the creation of a new compiler for OCaml, a popular statically typed functional language in the ML family, into the C language in such a way as to be able to take advantage of debugging tools provided by the C toolchain. This property, henceforth referred to as “observability”, refers to the ability to view the execution of the program through a debugger as if it was debugging a C program and to recover the program’s execution state.

I have successfully implemented a compiler for a core subset of the OCaml language in to C, which when compiled with a standard C compiler such as `gcc` or `clang` produces an executable that behaves in an identical way up to minor differences with the output of the native OCaml compiler. Additionally, when compiled with the `-g` flag, the resulting executable can be debugged under `gdb` in such a way that preserves the execution order of the original OCaml source, and allows setting breakpoints and observing values in the OCaml code in the same way as if it were debugging the original OCaml code.

In addition, the compiler performs well in comparison with both the OCaml native compiler and the bytecode compiler, creating executables that execute with similar speeds to the native compiler and debug in a similar manner to the OCaml debugger.

1.1 Motivation

There are several motivations for this project, including:

Observability

The OCaml debugger was introduced fairly recently, and only works on compiled bytecode; in addition, since OCaml strips away type information on actual values at runtime, the debugger is unable to inspect the values of certain variables within polymorphic functions.

We can consider the following example OCaml code:

Running this through the OCaml debugger, we see that it is unable to determine the value of the variable `x` within the `id` function.

Compiling to C could take advantage of a different data representation, and GDB extensions to customise debugger behaviour, allowing GDB to ‘see-through’ polymorphic values such as this..

Plausibility

There are many language features in OCaml which do not have similar analogues in C. Examples of these include algebraic data structures, partial application, lexical closures, and polymorphic datatypes and functions. I was interested in whether despite these differences, I could create appropriate transformations to simulate these in C but preserve observability at the same time, giving an OCaml view into what the code is doing.

Performance

By compilation to C, I can take advantage of the fact that the most popular C compilers have undergone decades of optimisation to produce very performant code. We can exploit this to obtain very fast executables without needing to perform much optimisation on the OCaml code. There will however be performance trade-offs in terms of emulating OCaml function calls in C compared to normal function calls, as OCaml functions can take advantage of features such as lexical closures and partial application, as well as being first-class allowing them to be passed as values to higher-order functions.

1.2 Previous Work

There exist prior work for compiling ML-like languages into C [?] but the approach used is to transform the program into continuation-passing style. This is not suitable for the compiler as it would interfere with the program execution and make it untenable to map OCaml source code into regions of the compiled code for debugging purposes, but shows that compilation into C from similar functional languages is feasible.

In addition, this project was also attempted in previous years by a previous student [?] under the same supervisor with a similar starting point, but my approach to the design of the compiler was not influenced and it is of my opinion that a sufficiently different approach was used for this project. A different representation of the output C code was used leading to a different compiler structure from the outset of the project, as well as completely different approaches being taken for more complex features in OCaml, such as data representation and closure conversion.

Chapter 2

Preparation

Before starting the project I had significant experience with functional languages through the Standard ML course undertaken and also personal experience with programming in Haskell, but I had never used OCaml at length, nor worked with the OCaml compiler codebase. While OCaml was very simple to learn from my previous experiences with functional languages, the OCaml compiler codebase was large, complex, and very sparsely commented.

In this respect, a large portion of time during the starting weeks of the project was focused on reading through barely-commented source code in order to understand the different data types used by the OCaml compiler, as well as research into various options for different libraries to use.

2.1 Starting point

The starting point of this project is the existing OCaml compiler¹, as it would be too onerous to duplicate the effort of parsing and performing type inference on OCaml source code. Instead, my compiler takes the `Lambda` intermediate representation from the OCaml compiler and compiles this into C. Since the OCaml compiler backends `Asmgen` and `Bytgen`, which produce the native binaries/bytecode executables, also only take the `Lambda` IR, this approach is akin to introducing a new compiler backend or different target for compilation.

The rest of this section will address why this was chosen as the starting point, and the process during the preparation phase in which the appropriate starting point was determined.

2.1.1 Intermediate representations

The OCaml compiler processes code in the pipeline shown in figure 2.1, with the pipeline branching towards the end into two separate stages representing the targets that the OCaml compiler can compile to. Between each stage, a different intermediate representation of the code is produced.

¹<https://github.com/ocaml/ocaml>

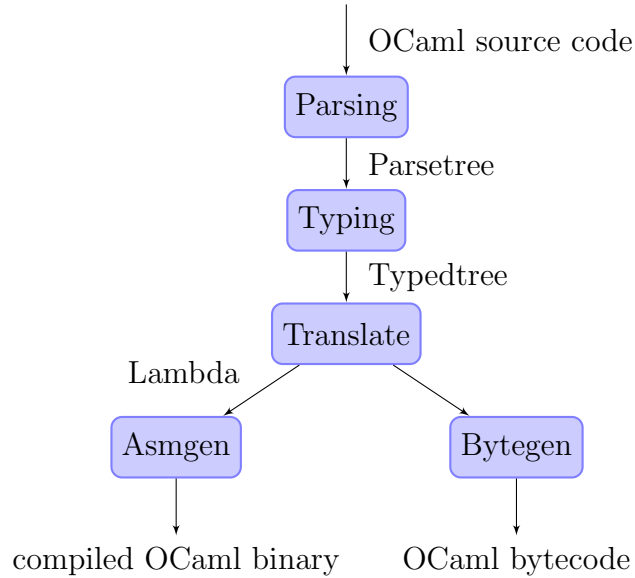


Figure 2.1: A representation of the various stages in the OCaml compiler. Adapted from [?, Chapter 22].

One of the tasks that had to be carried out during the preparation phase was to determine which intermediate representation would be most suitable for compilation into C. The requirements required for such an intermediate representation would be:

- It must be easy to recover type information from the representation. This is because the resulting C code must be statically typed also, so we need to be able to infer the types of variables and functions in order for them to be implemented in C.
- The representation must be fairly normalised, with as few as possible language constructs to simplify the translation into C. Many language features in OCaml are merely syntactically sugared versions of more primitive ones (for example, pattern matching is really a combination of branching based on the value of a variable plus some variable bindings for unpacking) which must have been desugared anyway during the compilation process, so it would be more useful if we could utilise an intermediate representation for which this has already happened.
- The representation must also allow some amount of reconstruction back to the source level, so we can map the output code to the correct sections of the source code. This is important for observability and debugging as we would like debuggers to display the correct segments of code being debugged. This does however present an antagonism, where we would like the IR to be both normalised, but not enough that we cannot recover any source-level information from it.

Parsetree

Parsetree is a parsed AST representation of the source code, which essentially directly represents the original OCaml source code. No processing has occurred on any language

constructs, and type inference has not even occurred on the IR. This obviously makes this intermediate representation unsuitable as the choice of IR, since it falls down on both ability to recover type information and normalisation.

Typedtree

Typedtree is a type-annotated AST which is almost exactly identical to the **Parsetree** IR, except that expressions have been annotated with a type. This makes this IR highly desirable as the starting point for compilation into C, and was initially where the investigation for a usable intermediate representation started.

However, since **Typedtree** still almost directly represents the original source code, it suffers from the same normalisation problem as **Parsetree** does. For example, the number of different sub-expression cases that **Parsetree** represents separately is 31 — modules, classes and objects are still represented as syntactical elements; records, variants, and field accesses have not been normalised into one representation; and pattern matches have not been simplified but instead stored as association lists of raw patterns to expressions.

This makes the **Typedtree** IR very cumbersome to work with, and ultimately it was considered unsuitable for the starting point of the project.

2.1.2 The Lambda IR

The **Lambda** IR is so named because it resembles an untyped lambda calculus, and is what in fact the two existing backends of the compiler generate code from. It has a number of advantages over **Parsetree** and **Typedtree**, including:

- Variants and records have been normalised into one representation, which is referred to as a block, which are similar to tagged unions. In addition, variants without parameters are compiled into bare integers.
- Desugaring of pattern match statements. Because variants have been compiled into integers or tagged unions, the translation pass is able to optimise pattern matching into switch statements. In very simple cases, pattern matches are in fact compiled into an if-else statement instead.
- Removal of modules and functors. These values have instead been compiled into equivalent representations using blocks and functions.

In comparison to **Typedtree**'s 31 different cases for sub-expressions, the **Lambda** representation has 20 different cases, which are relatively much simpler than those in **Typedtree**. This makes the **Lambda** IR far more desirable as a starting point.

However, there are some major disadvantages of the **Lambda** IR:

- The translation pass into **Lambda** does not preserve much of the type information from **Typedtree**. This is a huge problem which is solved somewhat with the existence of **Lambda** events, which will be discussed in §3.1.

- Normalisation in the `Lambda` IR discards a lot of source-level information such as which source-level expressions `Lambda` expressions map to, as well as having very little information about the location in the source code of the current code. This is somewhat remedied with `Lambda` events, which is discussed in §3.2.4.
- The `Lambda` IR has not yet had closure conversion applied to it. Functions are still represented in a similar fashion to how they were in the original source code, meaning the cases of lexical closures, partial application, etc. are not handled.

It was determined that while the `Lambda` IR does not fully satisfy the typing and source reconstruction requirements, enough type and source information was recoverable that it was deemed acceptable as a starting point. Thus, my project starts compilation by obtaining the `Lambda` representation by passing the source code through part of the compilation pipeline, and then translates the resulting IR into C from that point onwards.

2.2 Extensibility of GDB

Part of the project requirements dictate that the resulting code be “observable”, and it was quickly identified that if GDB is to be used as the primary debugger it may be necessary to extend GDB in order to handle the compiled OCaml code correctly and display recursive OCaml values. Thus, a brief investigation into GDB was carried out in the preparation phase to investigate its extensibility.

One of the experiments which was carried out was to see if GDB could display code from other files, and associate parts of compiled C code with instead parts of code from another file. This is a necessary part of the compilation process, as the resulting debuggable executable must display code and symbols from the original OCaml file from which it was compiled from via my compiler, not the C file that my compiler produces. With some investigation it was found that this was possible with the `#line` directive, which is explored in §3.2.4.

Another feature which is required for observability is the ability to display OCaml values at least recursively, if not formatted in the same way as the OCaml compiler does. This is because many OCaml values are recursive (for example, the list type is a recursive type which refers back to itself as one of its parameters) so it must be possible to extend GDB with some way to recursively print values whenever it finds a pointer. It was found that GDB does support extensions to itself via custom macros, and also supports custom pretty-printers (via, for example, Python) for different types. Either of these features would allow GDB to display OCaml values correctly.

2.3 Target subsets

During the write-up of the project proposal, three expanding subsets of OCaml were identified in order to structure the creation of the compiler around. These subsets represent a grouping of similar constructs and features together such that each subset focuses on a similar theme in the new features it introduces.

Identification of these subsets firstly has the benefit of providing a clear structure to the project, and greatly influenced the order in which features were implemented. Optimistically, it was expected that at the end of proposed deadlines a compiler would be completed that could compile the associated Subset – unfortunately, due to interdependence between a lot of the features a working compiler was not produced until towards the end of implementing features for Subset 2.

These subsets also serve to identify what the minimum viable product of the project is, as it was determined on project conception compilation of the entire OCaml language would be far too ambitious. As such, Subset 3 represents the subset of the language that the final product should at minimum be able to compile correctly.

Despite the fact that the subsets ultimately did not produce distinct and recognisable milestones for compilers that could operate on the appropriate subsets of the language, it is still worth describing the subsets for the effect they had on the planning for the rest of the project.

2.3.1 Subset 1

Subset 1 is a very simple language with only a limited number of types and language constructs. This subset contains only basic boolean, integer, floating point, and `unit` types, only basic string support (for input/output), top level function declarations with `let` and `let rec`, and basic language constructs such as `if`, `for`, and `while`.

2.3.2 Subset 2

Subset 2 expands on Subset 1 by introducing custom types and polymorphism. This includes tuples, lists, variant types via algebraic data types, record types, match expressions and function parametric polymorphism. This subset is aimed towards designing and implementing an appropriate representation of OCaml values in C, as well as a way of representing polymorphic values both as values in user-defined types and as parameters to functions.

2.3.3 Subset 3

Subset 3 expands further on Subset 2 by adding treatment of functions as first class values, plus closure conversion features, which would entail compilation of lexical closures, partial application, anonymous functions, etc. The focus of this subset is intended towards the design and implementation of closures in C, and correctly compiling the more difficult features of OCaml functions into C.

2.4 The `liballocs` library

`liballocs`[?] is a library that is able to track all allocations in memory and their associated types with no extra required effort. It exposes an interface that, when given an

arbitrary pointer, is able to return information about the type of the value in the allocated memory. This was identified to be extremely useful for implementing observability features, as it could be used within polymorphic functions to determine the type of certain values whilst debugging at runtime, which would be an advantage over the OCaml bytecode debugger, which due to the untagged nature of OCaml values cannot determine the type of polymorphic values at runtime.

Unfortunately, it was deemed that explicit tagging of types was easier to implement than inclusion of the `liballocs` library, but future extensions to the compiler may incorporate it to provide more detailed typing information.

2.5 The Boehm GC

Since the target subset aims to include a significant portion of OCaml including parts which allow for data structures and closures to be created. It is necessary that the runtime would require some sort of garbage collection. For reference, the OCaml native compiler includes as part of its runtime a specialised garbage collector written specifically for OCaml programs.

Writing a new garbage collector would be very complex and heavily outside the scope of the project, so we would like to make use of a pre-existing solution for garbage collection in C. Fortunately, many garbage collectors exist, and the Boehm GC² was selected and used.

The Boehm GC is a garbage collector for C and C++, which exposes a function `GC_MALLOC` which can work as a replacement for `malloc`. Whenever `GC_MALLOC` is called, it runs the garbage collector by scanning the stack for values that could potentially look like pointers and so performing garbage collection. This means that in simple cases, it can be used as a drop-in solution for GC without extra configuration.

2.6 Setting up the environment

The source code of the compiler was managed using `git`, the repository of which was backed up by publishing to GitHub³. Development initially took place on the master branch, but once the compiler approached feature completeness new features in the compiler would be developed in a forked branch, worked on until they pass the regression tests, and then merged into the master branch.

In addition, a copy of the source code was continuously backed up using Dropbox, which was synced between two local copies of the code, on two separate machines. In this manner, the code had four points of redundancy in the case of any computer failures.

²<http://www.hboehm.info/gc/>

³<https://github.com/t-veor/observable-ocaml>

2.7 Licensing of external code

The principal body of code being used is the OCaml core system⁴, which is released under LGPL v2.1. LGPL is a more permissive version of the GPL license, intended for use for libraries rather than full pieces of software. The LGPL license puts restrictions on what it calls “derivative works of the Library”, which are either modifications of the library or other software that includes the library or statically link to it, requiring that these derivative works are open source and also released with either the LGPL or the GPL library.

However, software that is designed to work with the library by linking against it but does not actually contain any portion of the library fall outside the scope of the license, which is the case with my compiler. As I am only releasing the source code of the compiler, which do not contain parts of the OCaml core system but link against it, the LGPL terms do not apply to the source code. They would however apply if I decided to release a compiled executable which has statically linked in the OCaml core system, but in fact a special exemption in the OCaml’s core system also covers this, allowing works which statically or dynamically link with a publicly distributed version of the OCaml core system to be exempt from the clauses of LGPL.

The other main library used is the Boehm GC⁵, which uses a custom permissive free software license, allowing for distribution and modification as long as a copy of the license is distributed with the library. This means in fact that my code is not within the scope of the license, since I do not distribute any part of the Boehm GC, but merely link against it in the compilation process. I have included the license within my repository nonetheless.

⁴<https://github.com/ocaml/ocaml/blob/4.05/LICENSE>

⁵<http://www.hboehm.info/gc/license.txt>

Chapter 3

Implementation

The following sections discuss the various tasks and features that had to be worked through in order to compile the target subset of OCaml. It is split into four major sections:

- §3.1 and §3.2 discuss the setup steps before translation between OCaml and C could start. These include obtaining types and representation of C types and C values.
- §3.3 discusses basic translation steps from OCaml into C, with more examples included in the appendix at §??. It closely matches up with the Subset 1 described in the preparation.
- §3.4 discusses representation strategies for more complex data structures and polymorphic values and the language constructs around these types. It closely matches up with the Subset 2 described in the preparation.
- §3.5 discusses the compilation of functions from OCaml into C, and solutions to problems such as local functions and closure conversion. It closely matches up with the Subset 3 described in the preparation.

3.1 Obtaining types from the Lambda IR

The Lambda IR does not have any type information associated with it, as within the OCaml compiler, Lambda was intended as the last representation before compilation into actual machine code, so all type information was erased from the underlying data.

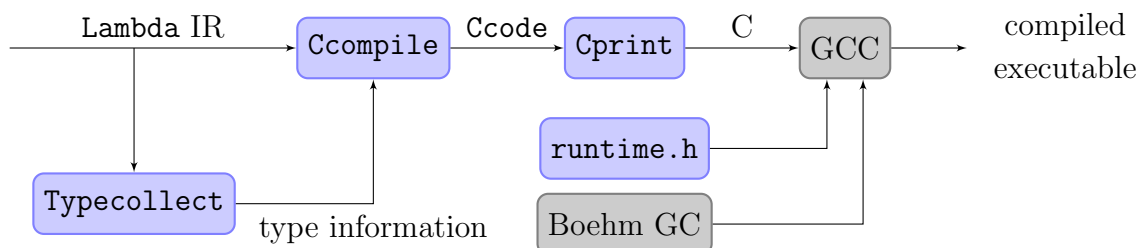


Figure 3.1: A representation of the stages in producing an executable output from my compiler, starting from the Lambda IR.

While not having types isn't such a big problem for the compilation into C, one of the goals of the project was to have the resulting code be “observable”, which means where possible variables should be of a meaningful type in order to be displayed correctly in GDB.

3.1.1 Lambda Events

The OCaml compiler inserts **Lambda** events into the **Lambda** IR when the debug flag is set. A **Lambda** event is simply a wrapper around another **Lambda** expression that carries some extra information, which includes the potential type of an expression and the current location in the source code.

Within the OCaml compiler, the purpose of a **Lambda** event is to mark where an interesting expression may be, so that the debugger may stop execution and inspect the state of the program just before the evaluation of the expression, or just after. This gives **Lambda** events two main useful properties, which are:

- **Lambda** events provide information about the current source code location of the expressions being currently compiled, which will be useful in for specifying source location (see §3.2.4);
- **Lambda** events provide type information about types of certain expressions.

Unfortunately, the compiler does not insert events that give types for every expression, nor is there a one-to-one correspondence between **Lambda** expressions and source-code expressions, since the **Lambda** IR may simplify certain expressions or insert new ones. However, every source-code variable was associated with at least one **Lambda** event describing its type. Therefore, determining types of variables used the **Lambda** events, and types of compound expressions were determined with very basic type- inference.

3.1.2 The Typecollect module

The **Typecollect** module therefore has a function **scrape** which does the following:

1. Initialise a hash table of variable identifiers to types (this is safe as the **Lambda** IR already assigns a different identifier to every variable)
2. Walk recursively down the **Lambda** tree:
3. Upon encountering a **Lambda** event surrounding a variable:
 4. Set the variable and its type in the hash table
5. Return the now-filled hash table.

This preliminary pass over the **Lambda** IR passes the inferred types to the **Ccompile** module.

3.2 Representation of C

Before compilation can start, an appropriate representation of C must be chosen as the compilation target for the compiler.

3.2.1 Expressions to statements

OCaml is what is known as an expression-oriented programming language, where every syntactical construct is actually an expression of some kind. This is in contrast to C, which is statement-oriented. Contrast for example the following (slightly contrived) piece of code in OCaml and C:

To solve this problem, my compiler assigns every sub-expression in the program to a separate variable. This means that if an expression consists of just a variable, use the variable name directly; otherwise, create and assign the result of each sub-expression to a new temporary variable.

By setting up a variable for each sub-expression, this creates an equivalence between variables and sub-expressions. The compilation of any expression therefore generates some sequence of statements, by the end of which the value of the expression is assigned to a variable. The compilation function therefore adds the statements to the current context, and returns the variable.

3.2.2 The Ccode module

The `Ccode` module is a module of the compiler that simply stores the type declarations for the internal representation of C. More details is given in the appendix at §??.

3.2.3 The Cprint module

The `Cprint` module is another module that's fairly straightforward – it contains functions for printing types from `Ccode` to an output stream, most likely a file. Its functions simply recursively traverse down the `Ccode` representation and prints out the corresponding C as it goes along.

3.2.4 #line directives

A key observability feature is for the debugger to associate the currently executing machine code with the relevant line from the source.

C compilers support controlling the line and source file of your code with use of the `#line` directive. This means the compiler can put all statements associated with one line of source on the same line¹, interspersed with a `#line` directive whenever the source code line actually changes.

¹The line number changes whenever there is a new line, even after being specified with `#line`.

To actually obtain information about which line we're on, we again use `Lambda` events (§3.1.1), as they carry information about the current file and line. Since they also represent places where the OCaml bytecode debugger `ocamldebug` may pause execution, this gives us a pleasing way to treat them: `Lambda` events simply compile directly into `#line` directives.

3.3 Compiler basics

Structures from Subset 1 were implemented first, which included basic structures, some of which are included below. In the interest of brevity other more trivial structures are described in the appendix at §??.

3.3.1 `let` bindings

When compiling `let` bindings, special attention needs to be paid to the variable scoping (see §??). In OCaml, a `let` binding has the structure:

$$\text{let } x = \textit{expr} \text{ in } \textit{body}$$

Note that x is not a free variable of \textit{expr} but is one in \textit{body} , and also that the value of the overall expression is the evaluated value of \textit{body} . Thus, a block is required to emulate the scoping correctly.

We therefore define the compilation of `let` bindings like thus, using the notation defined in §??:

\textit{expr} must be evaluated outside of the inner scope, where x is visible, and \textit{body} must be evaluated inside the inner scope. We declare and assign the value of x only inside the inner scope. In addition, the actual value of \textit{expr} must then be available outside the inner scope, so we propagate it outwards by declaring a variable `result` in the outer scope and performing its actual assignment in the inner scope.

OCaml also allows `let` bindings to be a shorthand for function declaration; function compilation will be discussed in §3.5.

3.3.2 Recursive bindings with `let rec`

Normally in a `let` binding, the variable being bound is not in scope for the expression being bound. This however makes it difficult to define recursive functions, so OCaml supplies the `let rec` binding which is useful for creating recursive functions, or sets of mutually recursive functions. For example, consider the following functions:

However OCaml allows `let rec` bindings to be used for creating a restricted class of non-functional recursive values. The typical example given is:

$$\text{let rec } x = 1::y \text{ and } y = 1::x \text{ in } \textit{expr}$$

This binds `x` to the infinite list `1::2::1::2::...`, which is accomplished by making the list cyclical. Unsurprisingly, this is very rarely used feature within OCaml, but in the interest of fully supporting all features within our OCaml subset we would also like to compile expressions like this correctly.

Informally, the class of values that are allowed to be used as the right-hand side of a `let rec` binding are those where the recursively bound names occur only within a function, or a data constructor. This means that to preserve the semantics in C, the compiler therefore compiles `let rec` in two phases:

- Variables are declared. If the associated expression has no free variables that are bound as part of the same `let rec`, evaluation can happen immediately; otherwise, its size is determined and then space for them is allocated with `malloc`, issuing them addresses in the process.
- Once all addresses are known, evaluation proceeds using the obtained addresses in place of variables which reference a value bound in the `let rec`.

3.4 OCaml value representation

This section largely details work undertaken to implement Subset 2, which deals with data representation and the language structures surrounding them.

3.4.1 Representation requirements

Before discussing the strategy for representing OCaml values, it is useful to discuss the problems that the value representation needs to solve.

Polymorphic types

Polymorphic code does not know the precise type of values it is manipulating, so cannot deal with variable size data as they may not copy polymorphic values correctly.

OCaml blocks

More complex types such as tuples and records are represented by a construct known as an OCaml block, which is a variable length array of values with a header containing its length and an integer tag value.

The Lambda IR in fact normalises all data structures into blocks, including references (they're a mutable singleton block), arrays, modules etc. which is rather convenient as we can implement all of these data types for free by implementing blocks.

Algebraic Data Types

A major feature in OCaml is the ADT, which combines elements of both enums and record types. As an example, consider the following type declaration:

size = 2
tag = 0
data[0]
data[1]

Figure 3.2: Example OCaml block containing two values with tag 0.

The data constructors (`Red`, `Green`, `Blue`, `Gray`, and `RGB`) within an ADT declaration are referred to as variants, and the type is a union of all of these types together. In the `Lambda IR`, data constructors that do not have any parameters are represented as integers starting from 0 (so `Red` is 0, `Green` is 1, and `Blue` is 2), and data constructors that do are represented using OCaml blocks, with tag values starting from 0 (so `Gray` is an OCaml block of size 1 with tag 0, and `RGB` is an OCaml block of size 3 with tag 1). This is the significance of the tag values – they differentiate between different variants in ADTs.

3.4.2 Representation strategy

As a consequence of the fact that we want the resulting C code to be observable, whenever there is an exact primitive type the C code should attempt to use the equivalent type in C as much as possible.

Representation for non-primitive types however is more complex. Integer types and OCaml blocks must be unified in representation, because variants that don’t have any parameters and variants that do can be of the same type.

Tagged pointers

OCaml blocks must be allocated on the heap since they are of variable size, and so have to be referenced using a pointer. This means that we need a type that can either be a pointer or an integer, which is what tagged pointers provide.

Tagged pointers are just a union type between pointers and integers. On most architectures, pointers are word-aligned, meaning that the lower bits of a pointer are always 0. The tagged pointer approach employed in this compiler is to use the least significant bit of an 8-byte word as the tag bit to differentiate between pointers and integers, and the remaining 63 bits to store the integer or pointer.

There is a choice as to which way round the tag bit should be – should 0 represent integers or pointers? I chose to use 0 for pointers, and 1 for integers, which means that the “boxed” version of a pointer is identical to the pointer in memory. This has the effect that dereferencing pointers is cheap and we can use a conservative GC with no modification – see §3.6.

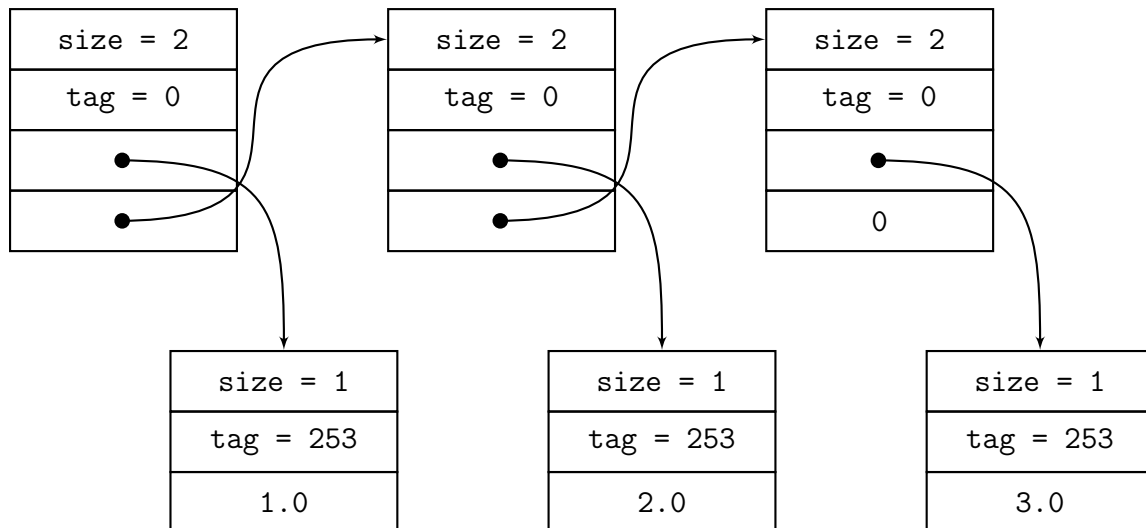


Figure 3.3: Example box-and-pointer diagram for the list `[1.0; 2.0; 3.0]`. Floating point numbers must be boxed to be stored in a block, so a special block is created for each one with the special tag 253.

Polymorphic values

We can use the tagged pointer representation to represent polymorphic values as well, by boxing all non-integer values into a block. To disambiguate these blocks from other normal blocks so that at runtime, the debugger is able to see what type they really are, we assign a special tag to them which for example marks the block as containing only floats or strings.

3.4.3 Representation in C

By taking advantage of flexible array members, it is easy to implement an OCaml block in C.

Recall that a block needs to store its size, an integer tag, and any number of other OCaml values. Since OCaml tags only range between 0-255², and 45 bits are sufficient for storing the size of a block in words, we can reduce the amount of space used for the header using C bit-fields. This means the header only needs to be one word long.

We can now implement actual OCaml values as a union type:

`BOX_INT` and `UNBOX_INT` demonstrate how to box integers into a `value_type`, and likewise for block pointers. We also have a special macro `IS_INT`, which checks if a `value_type` is holding an integer or a pointer – it simply checks the last bit, returning 1 if the last bit is also a 1.

²In fact, the native OCaml runtime restricts the number of possible variants to 246, using the tags 0-245 for variants. It uses the last 10 possible tags for special tags.

Type-punning with `any_type`

While `blocks` normally hold an array of other `value_types`, we would like to have special blocks that represent the result of boxing certain other primitive types, such as `floats` and `strings`. To do this, we declare a union type `any_type`³:

We refer the act of casting one of these types to an `any_type` as “packing”, or “unpacking” for the opposite direction. Note that since this is a union type, the “pack” operations don’t represent any actual operations in machine code.

`block` therefore declares a `any_type` array to support both storing `value_types` as well as special blocks that store `doubles` and `char*s`.

3.4.4 Conversion between C types

Despite the `Lambda` IR having already gone through type inferencing, there are still a few cases in which “casting” is necessary. This is because, for example, the C types for polymorphic types and monomorphic types in OCaml are different, so a C cast is necessary when passing a value of a known type to a polymorphic function.

In general, there are several cases where we need to perform a conversion operation in C. These cases are:

- A conversion from an integer to an OCaml value is required, because the `Lambda` IR uses integers for representing certain non-integer OCaml values.
- A conversion from any other type to an OCaml value, because a variable is being passed into a polymorphic function, or being placed into a block.
- An actual conversion between types, such as from integers to floating point numbers.

The cases all involve a mismatch between the C types of two expressions, so are handled using a function that does the following:

1. If the source and target types are the same, do nothing.
2. If either type is `any_type`, pack or unpack the value as appropriate.
3. If either type is `value_type`, box or unbox the value as appropriate.
4. Otherwise, perform a C cast between source type and target type.

3.4.5 Pattern matching

OCaml employs pattern matching in many constructs of the language, particularly in its `match` expressions. Fortunately, the `Lambda` IR usually compiles these down to an equivalent set of `if` expressions and `let` bindings. In the case of variants however, pattern matching typically compiles down into a `Lswitch` expression.

³For clarity, we ignore the problems associated with recursive type definitions in C, assuming the correct forward declarations.

Lswitch expressions

Lswitch expressions correspond to match statements on a variant expression. For an example, consider the type and match expression:

This match statement is compiled into the following **Lambda** expression:

There's one important difference between this and a C switch statement: each of the cases predicate on both whether the value is an integer or a block in addition to its value or tag.

One **Lswitch** statement therefore compiles into two switch statements in C, one for the integers and one for the tags, which are the two branches of an if statement that checks if the value being switched on is an integer or a value using the tagged pointer check.

3.5 Function compilation and closure conversion

Functions and closures are the final major feature required to compile the target subset of OCaml into C. The following sections will discuss the motivations for a closure representation and the implementation of such a representation, which is the majority of work undertaken in the implementation of Subset 3.

3.5.1 Local functions

All functions in C are required to be defined at top-level, which means that nested function declarations are not possible. This is in contrast to OCaml, where function definitions are simply expressions like any other, and thus can be defined locally and passed as values. In order to emulate this behaviour in C, whenever we come across a function definition we lift the function definition to the toplevel scope, compile the function there, and then return its function pointer when we come back.⁴

We can write the compilation of a local function like thus, using the notation from §??:

3.5.2 Function Types in C

In OCaml, one way of viewing functions is always as functions of one argument, which can return other functions. As an example, the function type `int -> float -> int` represents a function that takes an integer and returns a `float -> int`, which is a function that takes a floating point number and returns an integer.

While this is a correct high-level view of functions, this is not suitable for a low-level implementation at all, since it is much more efficient for functions to actually accept multiple arguments instead of taking one argument at a time.

⁴This clearly does not deal with lexical closures, so we actually return a pointer to a closure object instead of a function pointer, but the idea is the same.

In C by contrast, a function declaration consists of a return type, and a list of input arguments. We therefore need a consistent conversion between OCaml function types and C function types, so we pick the most obvious one – all the types in the OCaml function type are the arguments to the C function, and the final type is the return type of the C function.

For example, a function of type `int -> float -> str` is compiled to `char* (*)(int, double)` in C.

3.5.3 Closure representation

Closures are, implementation-wise, a structure that stores both a function pointer and an environment, containing values required for the execution of the function.

Our closure representation essentially requires storing a function pointer and some list of values, so taking advantage of flexible array members again we arrive at the following definition:

As you can see, this definition contains a function pointer, and a variably-sized array of arguments, which we reused `any_type` from §3.4.3 to implement. Closures also have a `next` field which points to another `closure_type`, allowing them to act like a linked list. The motivation of this is discussed in partial application compilation, in §3.5.4.

Using closure objects

In order to make the closure object useful, we modify all functions to instead take an extra parameter, `closure_type* closure_obj`, which stores the pointer to the current closure object. From there, the function is able to access closure values by indexing `closure_obj->args`.

Calling a closure can be done by invoking the function pointer with the required arguments, passing in the closure pointer as the last argument.

Promoting ordinary functions to closures

This necessitates some sort of promotion process where functions that do not require environment capture are promoted to closures. This is done by creating another function which takes all the same arguments as the original function plus the closure object. This function just passes the arguments along to the original function.

Unification of closures and functions

For ease of representation, it was decided to unify functions and closures in C as well. This means that all functions are promoted to closures, and all function applications in OCaml are compiled into closure calls in C, simplifying the implementation at the cost of possible performance losses.

⁴Note that we are reusing `any_type`, which means we need to extend its definition and the casting rules to accept more types, such as integers, function pointers, and closure types.

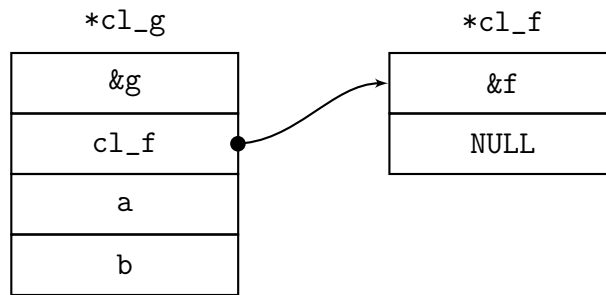


Figure 3.4: Example result of a partial application of a function $f(a, b, c)$ with the arguments (a, b) . The arguments a and b are pushed onto a new closure object, which contains a function pointer to g and a pointer to the previous closure object. When cl_g is called with an argument c , g will obtain a and b from the closure object and pass them together with c to f .

3.5.4 Partial application compilation

Partial application is one of the major use-cases for closures, and is modelled in my compiler using a closure object. Whenever there is a partial application, a function is created instead that takes the remaining parameters, and a closure object is created to store the partially-applied arguments. The newly created function simply obtains the previously partially applied arguments from the closure object and passes them along with its own arguments to reconstruct the original function call.

As an example, observe the following C-style pseudocode, where a closure cl_f (which requires three integer arguments and returns an integer) is partially applied with two integers, a and b , and another closure cl_g is returned⁵.

Chaining partial applications

An important requirement is that a partially applied closure needs to act the same as an unapplied closure, as in we must be able to partially apply it again. This requirement is the justification for why closures are implemented as a linked list – each closure needs to know the function pointer of the next function they need to invoke, but that isn’t always possible to determine statically⁶.

Thus, each closure resulting from partial application points to the closure from which it was derived from so their function can figure out what function pointer to invoke (see figure 3.4).

3.5.5 Lexical closure compilation

Lexical closures are another common use of closures, and occur when a function refers to values in an enclosing scope. This presents a problem in C, as it does not allow nesting functions.

⁵Cast operators have been omitted for simplicity.

⁶As an example, consider a partially applied function which is passed as an argument to a higher-order function, which further partially applies the function.

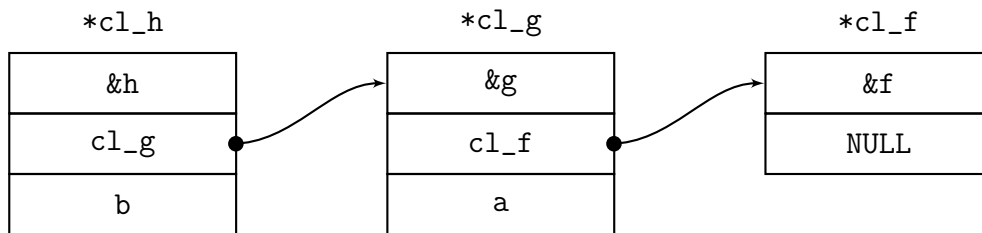


Figure 3.5: Example of a chained double application. In this case, the function $f(a, b, c)$ has been applied with a and b separately, creating two closure objects, cl_g in the first application, and cl_h in the second. Here, h translates a call $h(c)$ into $g(b, c)$, which in turn translates that call into $f(a, b, c)$.

Because our closure representation allows storing arbitrary data already, this is simple to implement using closure objects. When compiling any local function, first check if the function contains any free variables. If it does, take the value of all the free variables at that point and push them onto the closure object, and when compiling the function body, before compiling re-assign all lexically bound variables to values from the closure object.

As an example, consider the compilation of the following OCaml code:⁷

This is compiled into the following C-style pseudocode:

3.5.6 Parametric polymorphism

Parametric polymorphism is an important feature in OCaml, allowing programmers to write general code without knowing the type of its arguments in advance. As discussed in §3.4 we use the `value_type` representation of polymorphic values when defining polymorphic functions.

This works fine for passing simple values and polymorphic datatypes, as long as we remember to cast values to `value_type` when passing values to polymorphic functions, and cast back to their actual types when receiving values from function calls.

However, passing other functions (closures) into polymorphic functions is more problematic.

Erasing function types

When passing a non-polymorphic function, e.g. `float -> float`, to polymorphic function expecting an `'a -> 'b` requires a cast between the two types. This is because the polymorphic function is representing `'a` as `value_type`, and so will attempt to pass data of the type `value_type` into the function, and will expect a return type also of `value_type`. Thus in order to cast `float -> float` we will need to add a wrapper function which performs the casts.

⁷In an actual compilation, this will get eta-expanded to make the function types consistent (see §??) but we suppose this doesn't happen in this example for simplicity.

This in fact needs to happen any time it's possible to “lose track” of the actual type of a function, i.e. it's possible the function will be used in a situation where its type has been generalised away. This includes adding functions to an OCaml block (because the data structure may be passed to a function polymorphic in a subtype of the data structure), or accessing functions from an OCaml block⁸, or storing and retrieving functions from closure data.

Thus, the casting rules from §3.4.4 need to be modified.

1. If the source type is `value_type` or `any_type`, when casting back assume all of its argument and return types are `value_type`, and then cast it back to the expected type using the technique from the previous section.
2. If the target type is `value_type` or `any_type` and the source type is a function, first erase types from that function by rewriting all its argument and return types to `value_type` using the technique from the previous section before proceeding with the cast.

“Upcasting” functions

Another case when dealing with polymorphic functions is when the expected type of a function has fewer parameters than the actual type, e.g. passing a function of type `float -> float -> float` to a higher-order function that is expecting a `'a -> 'b`. This poses a problem for the type system for functions we decided on in §3.5.2.

To see why, consider a polymorphic function which receives a function `'a -> 'b`, and needs to apply it to something of type `'a`. How does it know whether to treat this as a full application, where it needs to invoke the function as a closure, or a partial application, where it needs to create a new closure object?

One solution is to solve this on the caller's side. Essentially, we want to transform a C function signature from:

into⁹:

This is to ensure that from the point of view of a polymorphic function, if the same number of elements is applied to the function type, then it can be considered fully applied.

This transformation can be modelled using the current closure representation, using a process I refer to as “upcasting”. In this process of upcasting a function `f`, two more functions are created, which are `g` and `h`.

`g`'s role is to act as the resulting function that will be passed into whatever polymorphic function. It's essentially a wrapper around a partial application.

`h` acts as the counterpart function to a polymorphic function, which takes the data that `g` has set up for it and applies it to `f`.

⁸It's not safe to simply remove the head from the closure chain to undo this cast, as there's no guarantee the last transformation applied to a closure will be the cast – the closure may have been partially applied or cast again to something else.

⁹The `closure_type*` is actually also cast into a `value_type`, to match the type the polymorphic function is expecting.

The surrounding code adds g as a closure onto f 's closure.

By tracing through the executions, it can be seen that an application by x followed by an application by y gives the correct function call $f(x, y)$.

3.6 Garbage Collection

We can use the Boehm GC to implement a garbage collector without needing to write one ourselves. Because of the way we have defined the implementation of OCaml values, all values potentially representing pointers will be unchanged in their actual binary value. This means that we can simply “drop-in” the Boehm GC by including the following in our runtime:

Chapter 4

Evaluation

The resulting compiler was evaluated in two key aspects, which are the resulting speed of the produced executables and the observability of the debug output. §4.1 talks about regression tests used to test the correctness of the compiler, §4.2 presents results and analyses about the speed of compiled executables, and §4.3 compares the observability of compiled executables with that of the OCaml debugger.

The benchmark results are quite favourable – despite not much effort being put in to optimise the output of the compiler, the resulting executables frequently perform as well as, if not better than, as the OCaml native compiler, although for certain workloads they perform much worse. The observability results show that basic observability features are possible, and in fact exceed the OCaml debugger in some cases.

4.1 Regression tests and feature completeness

To test the correctness of the compilation, 21 different regression tests were written, testing if different features of the language were being compiled correctly. They include simple tests which test one feature each, to more complex tests which test multiple features combined in different variations.

With every iteration of the compiler, regression tests are run against it to ensure that the output of the compiler remains correct. To do this, a simple regression test script was written which compiles the same OCaml program against the OCaml native compiler and my compiler, and asserts that the output of both executables match.

4.2 Benchmarks

One natural way of evaluating the compiler is to compare the execution times of its output against the output of the OCaml compilers. Therefore, for the evaluation we should compile the *same OCaml program* against both my compiler and the OCaml compiler, and then compare the execution speed outputs.

Several different configurations for comparisons can be attempted. OCaml supplies two compiler backends, the OCaml native compiler (`ocamlopt`) and the OCaml bytecode

compiler (`ocamlc`). For our purposes, we will be mostly considering the output of the native compiler, as the bytecode interpreter is at least one order of magnitude slower.

In addition, since my compiler produces C code we have a choice as to which compiler to compile the C code with, and under which configurations. I chose the most popular C compilers, `gcc` and `clang`, and compiled the code under the four optimisation levels provided (`-O0`, `-O1`, `-O2`, and `-O3`).

To simplify the process of running the many benchmarks against the configurations, a script `run_benchmarks.py` was written, and the results were plotted using Python and `matplotlib`.

4.2.1 Sourcing Benchmarks

Benchmarks programs were taken from the Computer Language Benchmarks Game (CLBG)¹, the `operf-micro`² OCaml library, and the rest were written by me. They were chosen to exemplify a wide range of features within the OCaml language, particularly to exercise the more advanced features of the subset of OCaml which I am targeting.

Benchmarks were adjusted to take between 0.5 to 2.0 seconds to run so that the timer does not significantly affect the running time, and were run 50 times each on an idle Linux laptop.

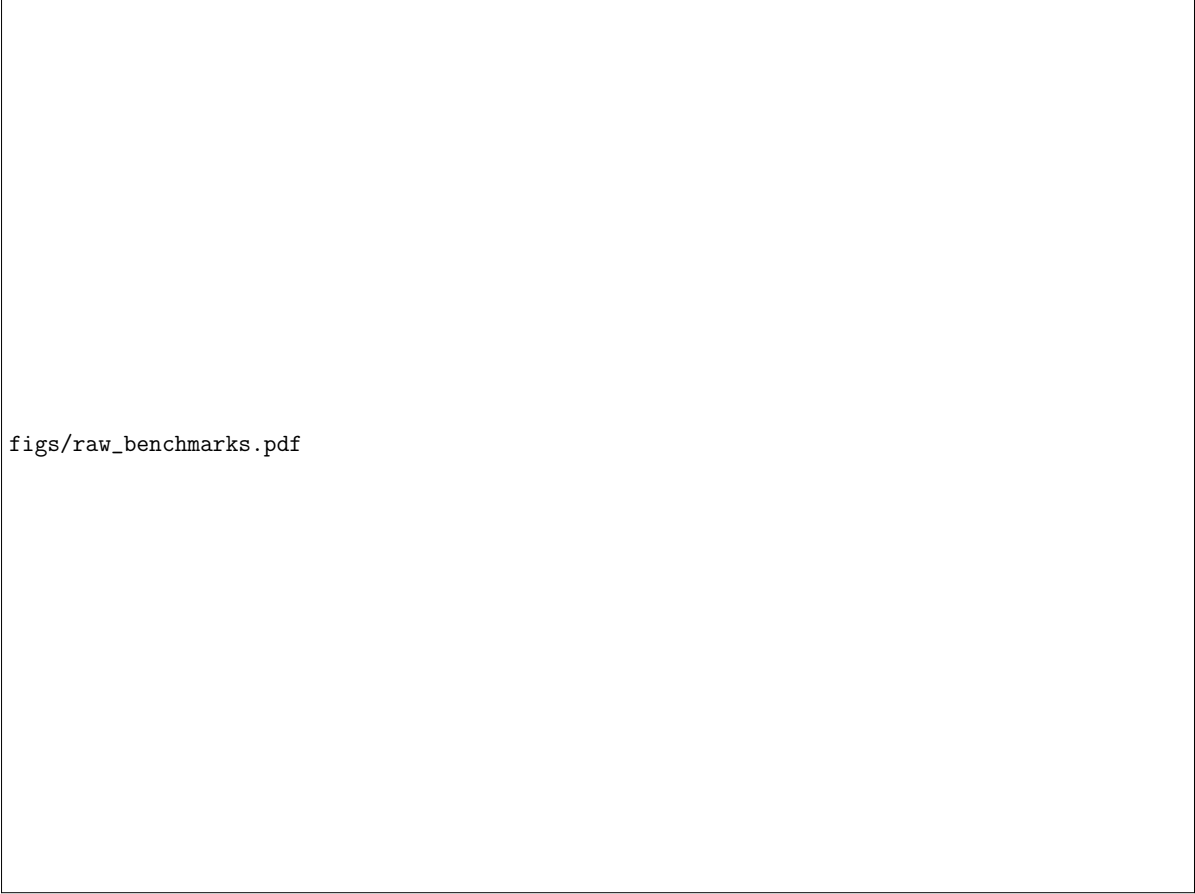
On the whole, C compilers do quite well in comparison to the OCaml native compiler being only a small constant factor time slower in most cases and matching and even outperforming the native compiler in select cases. Also for all benchmarks, the C compilers outperformed the OCaml bytecode compiler, which is surprising since no significant effort was made into optimisation of the C output.

Notable results from the benchmark include:

- The drop-in Boehm garbage collector is unsurprisingly much slower than the OCaml GC, as it is a conservative GC and does not have any domain-specific knowledge. This can be best seen in the `binary-trees` benchmark.
- The C compilers perform extremely well on benchmarks that involve few allocations and straightforward tail-recursive code, often beating even the OCaml native compiler. This is seen in the `collatz`, `even_fib`, and `sieve` benchmarks.
- Strangely, the C compilers perform exceptionally badly in benchmarks involving floating point manipulations, as can be seen in `mandelbrot` and `nbody` benchmarks. This is investigated in §4.2.2.
- Benchmarks which involve lots of closure creation and application are also slower relative to the OCaml native compiler, as can be seen in the `lens` and `stream` benchmarks.
- In the `sieve` benchmark, the executable compiled with `gcc` actually segfaults, the cause of which is a failure to perform a tail-call optimisation. This is discussed further in §??.

¹<https://benchmarksgame-team.pages.debian.net/benchmarksgame/>

²<https://github.com/OCamlPro/operf-micro>



figs/raw_benchmarks.pdf

Figure 4.1: Comparison of speed-ups in terms of execution time, compared to the OCaml bytecode compiler. Higher is better. The compared executables were the outputs of the OCaml native compiler (`ocamlopt`), the OCaml bytecode compiler (`ocamlc`), my compiler (referred to as the Observable OCaml Compiler, or `ooc`) in combination with `gcc` and `clang`. The black line represents the baseline of the OCaml bytecode compiler, and error bars represent 1 standard deviation in the results.

ocamlopt

ooc + gcc -O3

Figure 4.2: Truncated Gprof outputs for the OCaml native executable and the C executable for the **mandelbrot** benchmark, which has been highlighted with the roles of each function. Functions highlighted green are logical functions for the execution of the program, functions highlighted red are to do with allocating blocks on the heap, and functions highlighted blue are functions used by the garbage collector. Note that compiling with profiling information inserts extra instructions and function calls for instrumentation purposes so the times shown are not indicative of how fast the benchmarks execute otherwise.

4.2.2 Investigating performance differences

Instrumentation

The obvious next step after obtaining results for the benchmarks is to perform an investigation into the performance differences; we would like some way of profiling the code to see what parts of the code take longer to run.

Fortunately, a key consequence of the compiler being from OCaml to C is that standard C tools will work on the resulting code. This includes Gprof, a profiler which can be enabled in gcc with the `-pg` flag. Gprof can provide information as to how much time is spent in each function via the flat profile, as well as information about how the call graph, or which functions called what other functions and how many times.

OCaml also supports using Gprof to profile its native code output simply by adding the `-p` flag to `ocamlopt`, giving us a convenient way to compare the execution of the C executable and the OCaml native executable. We can therefore use this profiler to investigate performance differences between the OCaml native compiler and the C compiler.

Floating point and polymorphic structures

For example, one discrepancy which we may wish to investigate is the **mandelbrot** benchmark, which C compilers perform much worse on despite having been written in imperative-style code.

I collected profiling information using Gprof from both the native and C executables, a truncated version of which³ can be seen in figure 4.2. As shown, the OCaml executable spends almost 100% of its execution time in the main logic of the program (which is the **entry** function), in comparison to the C executable, which only spends 44.37% of the execution time in the main logic. Instead, the majority of the time spent during execution is allocating new blocks and garbage collecting old ones.

Upon further investigation, it was found that the **mandelbrot** benchmark stored floats inside `float refs`, and the **Lambda** representation normalises references into sin-

³Since the full output from Gprof is not as useful, the reports have been truncated so that at least the top 95% of the execution time is represented.

Figure 4.3: Disassembly of the the function `let rec add x y = if y = 0 then x else add (x - 1) (y + 1)` compiled using my compiler and then using `gcc -O3`. The compiler is able to optimise the tail call, but is not able to see through the closure representation to optimise into a loop, instead jumping to a function pointer obtained via the closure object (the pointer to which is initially stored in the `rdx` register).

gleton OCaml blocks. This means that according to the casting rules as described in §3.4.4, whenever a `float` needs to be stored inside a `float ref` a new tagged OCaml block is allocated on the heap and then the pointer is stored in the `float ref`. This means that mutable floating point fields within OCaml blocks are in fact incredibly slow, as each store necessitates creating a new block on the heap, and needing to GC the block that was replaced.

The same is true for the `nbody` benchmark, which uses mutable float fields in a record to store the position, velocity, mass etc. of the different celestial bodies.

This unfortunately makes floating point operations very slow, and a solution is attempted for this in §??.

Tail recursion

Another interesting question to ask is if the C compilers can see through the closure representation to perform tail-call optimisation, since it is far more idiomatic to write tail-recursive functions in OCaml than it is to write explicit iteration. It is therefore important for performance that the C compilers can do tail-call optimisation to avoid the extra overheads of function calls as well as avoiding stack overflows.

To do this, we can use the `objdump` Linux utility, which can “disassemble” the executable, printing out the assembly mnemonics corresponding to the machine code. Figure 4.3 shows the disassembly of the following simple tail-recursive function:

As can be seen, the C compiler is able to infer that the closure application is equivalent to a function call in tail-call position, despite not being able to see through the closure representation explicitly, and has therefore compiled the closure application into a single `jmp` instruction. This means that tail-recursive functions do not incur the full penalties from a function call, but a small amount of overhead from needing to load in an address.

Closure creation

For a lot of benchmarks the C compilers perform far worse than the OCaml native compiler which is likely to be due to the fact that the closure operations in the generated C are far slower than the OCaml native compiler can compile them.

To confirm this, I collecting profiling information from the `lens` benchmark, which is the most heavy benchmark in terms of closure creation and application. The truncated flat profiles can be seen in figure 4.4.

As can be seen by the results, the execution time of the C executable is strongly dominated by the garbage collector and functions used in the process of closure creation

ocamlopt

ooc + gcc -O3

Figure 4.4: Truncated Gprof outputs from the `lens` benchmark, coloured using the same scheme as before in figure 4.2, with the addition of yellow functions being related to the closure creation/application process.

benchmark	ocamlc		ocamlopt		ooc +				ooc (reduced allocations) +			
	mean	std	mean	std	gcc -O3		clang -O3		gcc -O3		clang -O3	
<code>binary-trees</code>	7.959	0.088	2.068	0.012	4.858	0.032	4.052	0.027	4.864	0.035	4.117	0.095
<code>collatz</code>	28.437	0.055	2.351	0.019	2.236	0.006	2.344	0.008	2.305	0.013	2.348	0.019
<code>even_fib</code>	8.218	0.066	1.423	0.026	0.992	0.007	0.971	0.009	0.964	0.009	0.971	0.011
<code>fibonacci</code>	7.211	0.190	1.032	0.009	1.275	0.006	1.122	0.015	1.321	0.006	1.124	0.021
<code>lens</code>	4.510	0.120	1.294	0.027	2.788	0.056	2.620	0.058	2.322	0.020	2.084	0.079
<code>mandelbrot</code>	16.609	0.151	0.931	0.009	9.307	0.076	8.788	0.249	1.548	0.007	1.354	0.018
<code>nbody</code>	8.251	0.133	1.366	0.021	3.311	0.078	3.203	0.059	0.764	0.013	0.716	0.016
<code>rule30</code>	2.311	0.030	1.128	0.032	1.943	0.054	1.588	0.027	1.923	0.016	1.577	0.016
<code>sieve</code>	2.341	0.021	1.687	0.027	N/A	N/A	0.882	0.025	N/A	N/A	0.880	0.021
<code>stream</code>	3.966	0.022	0.884	0.041	3.614	0.021	3.180	0.128	2.896	0.071	2.409	0.071

Figure 4.5: Summary table of benchmark results. All figures given in seconds. (The final two columns are a result of §??.)

and application, with functions implementing logic actually being a small fraction of the execution time. This is in strong contrast to the OCaml code, where the run time is dominated instead by the actual logic of the program and the printing functions.

This suggests that in certain workloads the closure representation can degrade the performance significantly, resulting in unnecessary amounts of work being put into closure operations as compared to if the compiler was able to see through them. Furthermore, since the closure operation necessitates allocating new blocks onto the heap, creation of lots of closures can put much more extra stress onto the garbage collector.

These problems may be allayed with a better closure representation, for example turning partial application into generating function stubs which push the correct arguments into the correct registers before jumping to the function body. Unfortunately, there was insufficient time to investigate more performant closure representations.

4.3 Observability

The other way to evaluate the the compiler is to determine the observability of the compiled output, or how much information we are able to recover about the internal state of the program as it is executing.

The OCaml native code compiler can be said to be not observable at all, since the native executables do not support source-level debugging. Instead, comparisons will be made with the OCaml bytecode debugger.

4.3.1 Sample GDB session

We consider the following simple program, for summing a list of numbers using a generic fold function.

Debug sessions when debugging using both GDB and `ocamldebug` can be found in appendix §??, with equivalent commands entered for both sessions. There are some minor differences with regards to which locations the debuggers stop at, and the commands have been altered slightly to reflect this.

By loading in a script which enables custom pretty-printers, we are able to observe OCaml values in a nicer representation than printing out raw pointers. In addition, the use of `#line` directives has meant that that GDB can display correctly the lines of OCaml code the code it is currently debugging correspond to, allowing breakpoints to be set. Since what is being debugged underneath is essentially just a C program, the full power of GDB's tools can be utilised, including breakpoints, backtraces, watchpoints, etc.

4.3.2 Identifiers

In the C code, in order to avoid ambiguity we suffix identifier names with their unique 'stamp' in order to avoid naming conflicts. Practically, this is more of an annoyance than a problem. As GDB allows tab completion in order to reference variable names, to obtain the name of a variable in the C code one only needs to type the name followed by an underscore, then press tab to tab-complete the appropriate stamp.

This does however mean that debugging is a worse experience in comparison to `ocamldebug`, where typing the name of a variable automatically refers to the variable with that name which is currently in scope. Because of the precautions outlined earlier in §??, it should not be theoretically difficult to erase these 'stamps' from variable names, but practically there are a few problems with this:

- The `Lambda` IR introduces new variables in certain cases, but does not care about naming conflicts as its transformation is post-source-code level where identifiers are compared using their stamps only. This means that erasing stamps naively would cause naming conflicts.
- There is no equivalent way to deal with this for top-level identifiers, as we cannot have shadowing in the top-level scope in C.

4.3.3 Polymorphic printers

The `debug/pprint.py` script is a script written to enable printing of OCaml values within GDB, as otherwise many values will only be displayed as pointers otherwise.

This is a point where my implementation offers better functionality than `ocamldebug`. Because of the representation strategy described in section 3.4, we can determine the types of values at runtime even if the code we're debugging does not know the types themselves.

We can see this at lines 24-31 in the GDB output (??), and lines 10-15 in the `ocamldebug` output (??). The pretty printers which I have loaded into GDB are correctly able to infer that the values being printed are integers and an integer list, but `ocamldebug` cannot, and instead can only print the polymorphic values as `<poly>`.

One limitation of this however is that when printing data structures, I print the in-memory representation of the structure instead of how it would be represented in OCaml. Had I more time this could be remedied by also outputting a lookup table for the pretty printers to use to find out what type a data structure is and how it should be formatted. As of currently, OCaml blocks are printed with the tag first, a colon, and then the data within the block; this results in the list `[1; 2; 3]` being printed as:

```
[0: 1 [0: 2 [0: 3 0]]]
```

4.3.4 Line number matching

While the line number matching between the C code and the OCaml code is acceptably good for debugging, there are certain cases which it does fall down, due to a fundamental difference in the way GDB and `ocamldebug` treats points where the debugger may pause execution.

This is especially confusing sometimes for local anonymous functions. For example, in the `sum` example from earlier, stepping inside the `(f acc x)` expression on line 5 steps you back to the line 8 (`let sum = foldl (+) 0`), because we are stepping inside the implicitly defined `(+)` function, which was defined on line 8.

`ocamldebug` notably only pauses at `Lambda` events, which are described in §3.1.1, but GDB only pauses on lines of code. Since `Lambda` events are inserted directly before or after certain expressions, `ocamldebug` is able to pinpoint exactly where in the evaluation of the code it has stopped, but GDB is unable to do this, only printing the line of code where it stops at.

4.4 Summary and discussion

The benchmark results are reasonably good; under idiomatic tail-recursive code, the benchmarks can be seen to run just as fast as the OCaml native compiler, and the compiler solidly beats the bytecode compiler in all cases. This is encouraging because during implementation no significant effort was paid towards optimisation, showing that compiling to C as an intermediate language can make use of the many optimisations the C compilers have undergone and even naive transformations of OCaml code can match in performance to the official compilers.

We see that the current compiler has two major downfalls in terms of performance, which is GC time and closure conversion. My implementation for closure conversion is quite a bit slower than the OCaml native compiler, as well as creating lots of objects on the heap – which puts even more strain on the garbage collector. Using a garbage collector simplifies the implementation of the project by a lot, but it is clear that naively using the Boehm GC decreases the performance of the produced executables significantly, and a specially-written garbage collector would perform better than the Boehm GC.

In terms of observability, we can see that the produced output is fully debuggable by GDB, if not quite rough around the edges – GDB is able to print variable values, and even see through polymorphism in data structures and polymorphic code which the OCaml debugger can not. There is something to be desired in terms of user experience however – identifiers need to be tagged with a stamp, which means the user needs to use tab-completion to print the variable they want, and printing OCaml blocks does not print the actual OCaml constructors they represent, leading to a suboptimal user experience.

Chapter 5

Conclusion

Overall, the project was largely successful as I had accomplished my goal of compiling the target subset of OCaml correctly into C, as well as providing facilities under which the resulting C code could be “observed” by a debugger as if it was OCaml code.

Translations of technically challenging features of the language such as data representation and closure represented were attempted, and largely successful. My implementation also leaves room for further additions to the compiler to add support for the remaining features of OCaml which were not included in the target subset if more time was to be dedicated.

We can see that the motivations of the project have been largely fulfilled, leading to a novel approach for compilation that allows for a debugger of a different language to debug the same code. These motivations include:

Observability

The work into making the code observable has largely held up, with GDB being able to display the current location of the code and observe OCaml values, even polymorphic values that the `ocamldebug` debugger could not discern. It is a weak point however that GDB is unable to print out OCaml data structures using OCaml’s notations for them, but this is very possible with an extension to the compiler that generated debug information for the types of objects within the executable being debugged.

Plausibility

The completion of the compiler is a proof that my approach to compiling OCaml and other similar languages into C works, and it is possible to represent even complex closure structures and OCaml’s algebraic data types within C, as well as a debugger being able to “see through” parametric polymorphism to determine the types of the values being manipulated.

Performance

Despite no heavy efforts being paid to make the resulting code performant, under all tested workloads my compiler outperforms the OCaml bytecode compiler and matches or

is sometimes even faster than the OCaml native compiler in select workloads.

My project leaves plenty of room for future extension work, and there are things that I would've done differently had I started the project again. These include:

- Support for OCaml's module system, allowing multi-file programs to be compiled correctly. Currently the compiler is only capable of compiling one file at a time, but it should not be difficult extend this support;
- Support for more of OCaml's standard library, allowing standard library modules to be compiled and used;
- Support for a larger subset of OCaml's core language, including array types and a better support for floating point values, in order to make them more performant;
- A key assumption of the code is that all primitive types needed are 8 bytes long, or word-sized on a 64-bit system. It would be nice if less reliance was depended on this so the compiler was more portable to other systems than 64-bit Linux;
- More performant closures, such as generating function stubs at runtime to implement closure operations, or using a closure implementation from a popular library such as `libffi`;
- More time spent with run-time support libraries, such as my supervisor's `liballocs`, in order to make it possible to have runtime reflection.