Cheng Sun

An observable OCaml via C and `liballocs`

Computer Science Tripos – Part II

Churchill College

May 19, 2017

Proforma

Name:	**Cheng Sun**
College:	**Churchill College**
Project Title:	**An observable OCaml via C and `liballocs`**
Examination:	**Computer Science Tripos – Part II, July 2017**
Word Count:	**11669**[1]
Project Originator:	Stephen Kell
Supervisor:	Stephen Kell

Original Aims of the Project

To create an OCaml backend that can compile a useful subset of the OCaml language to C, including for instance tuples, records, polymorphic and first class functions. Furthermore a subset of the OCaml standard library should be provided, and a C runtime library implemented, providing support for closures – a feature non-native to C. The produced C code is to be integrated with liballocs, a library tracking allocations at runtime, and should be debuggable using liballocs to allow polymorphism to be "seen through".

Work Completed

Despite the difficulty of implementing an entirely new backend into any compiler, I have successfully created a working compiler backend translating OCaml's `Lambda` intermediate representation into standard C, which is capable of compiling the `Pervasives` and `List` stdlib modules from upstream as-is. Furthermore I implemented a C runtime library which provides support for I/O, exceptions and closures, amongst others. Observability has been demonstrated with `liballocs`, and an extensive test and benchmark suite demonstrates competitive performance compared to the upstream compiler.

Special Difficulties

None.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Cheng Sun of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

OCaml is one of the most commonly used members of the ML family of functional languages. It is popular for its expressivity, type system and performance. My project augments the OCaml compiler with a new backend which is capable of targetting the C programming language. The code produced by my compiler can then be compiled with a C compiler such as gcc, and the resulting binary, when linked with my runtime library, behaves equivalently to the original OCaml code.

1.1 Motivation

There are several motivating factors behind this project.

1. **Feasibility**: I was curious whether it was technically possible to compile from OCaml code to C, and if so, how hard it would be to do so. I was also excited by the challenge of working on the internals of an industrial-strength compiler for the first time;

2. **Universality of `Lambda`**. The `Lambda` IR is a fairly flexible and expressive target, which is rapidly garnering attention as a suitable target IR for compiler frontends of other languages [6]. My project attempts to implement the other direction as well – that `Lambda` is suitable for being easily translated into other representations;

3. **Performance**: C has very mature toolchains that have been refined and optimised over tens of years. It would be nice to leverage the optimisations that the GCC and Clang compilers can provide for free;

4. **Observability**: gdb is a very mature debugger for C, which far surpasses OCaml's in power. Along with enhancements brought about by `liballocs`, I want to make it possible to observe polymorphic values propagating through an OCaml program. A more detailed example is given below;

5. **Portability**: C can be compiled on a diverse range of platforms. It would be nice to know that, by using C as an intermediate step, OCaml code could theoretically run on these platforms with minimal to no changes.

There is not as of yet a good story for debugging OCaml programs, and observing their behaviour at runtime. The following is an example which I brought up in my project proposal:

```
let my_rev lst =
  match lst with
  | [] -> []
  | x::xs -> List.append xs [x]

let result = my_rev [1; 2; 3]
```

ocamldebug, the OCaml bytecode debugger, cannot see through polymorphism. This means that if a breakpoint is set in `my_rev`, ocamldebug would not be able to print out the value of the polymorphic argument `lst`, because its actual type is not known.

My project solves this problem via `liballocs` – a library written and maintained by my supervisor, Stephen Kell.[2] This library keeps track of runtime allocation information with low overhead. The library is further described in section 2.7.

There exists related prior work [1] that shows that the concept of translating from a functional language from the ML family is feasible. However, my design and implementation will be different. For instance, the presented compiler generates code that uses continuation-passing style, which I would like to avoid doing, as it harms observability – backtraces would no longer be directly meaningful.

Chapter 2

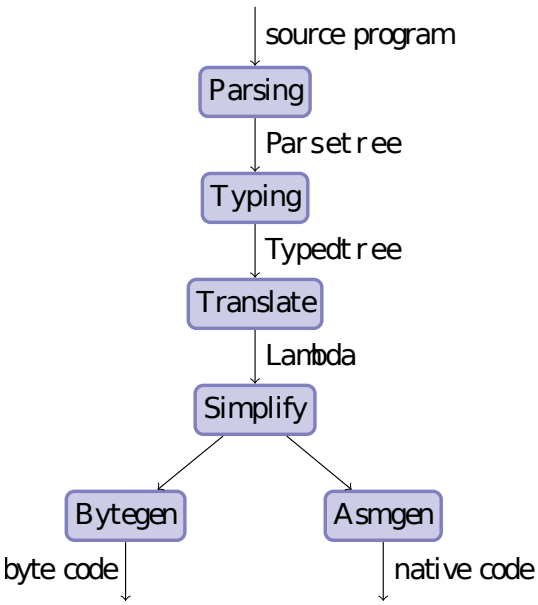
Preparation

Before starting the project, I was fairly comfortable with both the OCaml and C languages. However, I had no experience with the OCaml compiler codebase, nor indeed any experience in working on a project as large as it.

The vast majority of my code will be written to be built as part of the OCaml compiler. The compiler is itself written in OCaml, and is a large and complex code base developed over the course of over 20 years, consisting of 197k lines of OCaml and 42k lines of C.[1]

Unfortunately it turns out the codebase is very sparsely commented, with no documentation available as to its inner workings. I spent a significant amount of my preparation phase tracing through dumps of internal representations of example OCaml programs to gain understanding.

The OCaml compiler processes OCaml code in several phases:



2.1 Common types

The compiler represents the name of all variables as `Ident.t`, which is conceptually a tuple of the variable name string and a unique integer. Each let binding is given its own

[1] As counted by the `cloc` tool.

unique `Ident.t` – once the frontend has resolved variables using OCaml's scoping and shadowing rules, the rest of the compiler has the guarantee of globally unique[2] identifier names.

A `type_expr` represents the OCaml type of a particular expression.

2.2 `Parsetree` and `Typedtree`

The very first representation used by the compiler is `Parsetree`. This is an Abstract Syntax Tree produced directly from the OCaml source code by the parser. At this stage all original OCaml features are still present, before any desugaring has happened yet. For instance, pattern matches are as originally expressed in the source code.

The typing phase takes a `Parsetree` and produces a `Typedtree`. This is the phase that ensures that all expressions are well-typed. The `Typedtree` is almost identical to `Parsetree`. The difference is that every `expression` node in the tree is augmented with its inferred `type_expr`.

2.3 `Lambda` IR

For our purposes, `Lambda` is the most important representation in the OCaml compiler, as it is the starting point for our compiler. This Intermediate Representation (IR) is based on a heavily augmented untyped lambda calculus. It is a fairly low level representation, for example using integers to represent what were originally booleans or variants.

The pass compiling `Typedtree` into the `Lambda` IR most significantly desugars pattern matching into elementary conditionals and destructuring operations. Also class and module constructs are compiled down to operations on records.

Notably this pass does NOT:

- preserve type information from `Typedtree`. This is annoying for us because we would like type information for `liballocs` to consume;
- perform closure conversion;
- perform uncurrying. Significantly for us this means partial applications and "over-applications" are not explicitly expressed as complete applications.

These issues are addressed in the implementation section.

2.3.1 Why `Lambda`?

Quite a lot of research was done into the various intermediate representations in the OCaml compiler before starting my project. In the end, none of the Intermediate Representations were deemed perfect, but `Lambda` was the closest starting point to what I wanted.

[2]Within a particular module.

1. `Parsetree` and `Typedtree` are not appropriate as a starting point for my compiler. This is because no desugaring has been performed yet, which means that there are too many cases and constructs that would have to be handled;

2. `Clambda` (a late-stage IR that is part of the OCaml native compiler) is more suitable as a starting point, because it represents closures explicitly. However it is even harder to retrieve type information from than `Lambda`. Although `Lambda` doesn't preserve `type_exprs` for every expression, some specific information, in the form of "events" (see section 3.3), are stored for the benefit of the ocamldebug bytecode debugger. These events are dropped in `Clambda`.

2.3.2 Introduction to `Lambda`

Each OCaml file implicitly defines a module, whose name is the same as the filename capitalised. For instance, `foo.ml` defines a module named `Foo`.

At the toplevel, there are two different types of let statement. Let-statements which define functions have bodies should not be executed immediately – only when the function is called. All other let-statements should run exactly once when the module is loaded for the first time. For instance:

```
(* function -- not run immediately, exported *)
let foo () = [ 3 ; 2 ; 1 ]

(* ignored result -- run immediately, discarded *)
let _ = foo ()

(* variable -- run immediately, exported *)
let bar = List.sort (-) (foo ())
```

This will get translated into the following (abridged) `Lambda` code.

```
1  Lseq (
2    Lprim (Popaque, [ Lprim (Pgetglobal, [ "List" ]) ]),
3    Llet (
4      foo_1199,
5      Lfunction ([ param_1255 ], ...),
6      Lseq (
7        Lapply {
8          ap_func = foo_1199,
9          ap_args = [ Lconst (Const_base (Const_pointer 0)) ]
10       },
11       Llet (
12         bar_1200,
13         Lapply {
14           ap_func =
15             Lprim (Pfield 40, [
16               Lprim (Pgetglobal, [ "List" ])
```

```
17              ]),
18            ap_args = [ ... ]
19          },
20          Lprim (Pmakeblock (0, Immutable), [foo_1199, bar_1200])
21        )
22      )
23    )
24  )
```

There are several things to note about the `lambda` tree produced.

1. **`let`** gets translated to `Llet`, as can be seen on lines 3 and 11 of the output. The tree ends up very right-side heavy, as each `Let` constructor takes as its last argument a `lambda` representing the rest of the module. This mirrors the form of a (non-toplevel) **`let`** `...` **`in`** `...` expression in OCaml, in contrast to a more imperative style `var ...; var ...;`. Although one does not write the **`in`** keyword for let-statements at the top-level, under the hood it desugars to this form regardless.

2. All variable references are represented using `Ident.t` (see 2.1), denoted here using an underscore between name and numeric identifier.

3. On line 7, the `foo` function is applied using `Lapply`, which takes a record containing (amongst other things) the function and the arguments to apply.

4. On lines 13–19, `List.sort` is applied. Note how this requires accessing the globally defined `List` module object. Also note that nowhere does the name "`sort`" appear.

   The OCaml compiler compiles modules to records at the `Lambda` level. These are stored as blocks of contiguous word-sized values, and are indexed into by an integer offset (using the `Pfield` primitive). The order of values in the record is well defined by the OCaml compiler, so that other modules that link against it know the mapping from exported variable name to field offset.

   In this case the compiler knew that `List.sort` was the 40th exported variable in the `List` module.

5. Line 20 constructs the "return value" of the `lambda` expression. This is a record, representing the currently compiled module object, which contains the variables and functions that the module exports.

   Note that not all variables and functions are necessarily exported – exports can be restricted by the `.mli` file if present.

2.4 Starting point

The project will be focused on creating a new "backend" for the OCaml compiler, so I will build on the existing code of the rest of the compiler. This includes using preexisting code for lexing, parsing and typing of OCaml programs, as well as the transformation from the typed AST to the `Lambda` Intermediate Representation that we will be using.

In order to provide a standard library to compile programs against, I used the existing OCaml `stdlib`. However, the C runtime library was completely written by myself.

I did not make use of either `cil` or `libffi` as suggested in my project proposal. The reason behind this is explained in sections 2.5 and 2.6. All of the functionality they would have provided was instead manually implemented by myself.

I did make use of `liballocs`, described below (section 2.7).

2.5 Investigating C AST representations and output

My compiler is required to output C code, so a representation of C is required that allows for easy manipulation. My supervisor suggested the use of `cil`, a C "intermediate language" which is capable of semantically representing all C programs using a minimised subset of the language.

I came to the conclusion that this wouldn't be a good fit for my project, because CIL is, in a sense, trying to accomplish the opposite of what I want. CIL tries to represent a minimal subset of C, and allow transformations to be performed on this. On the other hand I'd like to represent the useful subset of C which I can use to compile OCaml constructs to.

In addition, I anticipated that I may need fine-grained control or custom features to be represented in my AST. Hence I ended up going with a hand-rolled representation.

2.6 Investigating closure support in C

Supporting closures in C was one of the earliest identified points of difficulty. Early on my supervisor recommended that I look at the approach introduced by Breuel [3]. After reading and understanding the paper I had a better understanding of the problem and was fairly confident I could implement the proposed solution.

Even so, some preliminary investigation was done towards possible external libraries that could be used to provide this support for me. One of these, `libffi`, was also suggested by my supervisor.

I came to the conclusion that `libffi`'s API was too generic and complex for the very specific use case that I had in mind. In addition, I wasn't sure whether the library would give me enough control. Finally, I wanted to take on the challenge of implementing closures myself, so that I could gain a better understanding of the technique.

2.7 `liballocs`

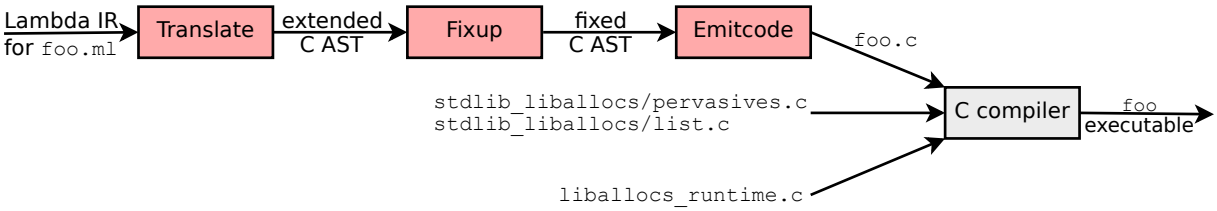
I will use the open-source `liballocs`, written by Stephen Kell. `liballocs` is a C library (and accompanying toolchain wrapper) written and maintained by my supervisor, Stephen Kell. The library transparently keeps track of the sizes and types of all allocated memory, with no modifications required. Given an arbitrary valid pointer, `liballocs` can return information about the start and the extend of the corresponding allocated block, and also the C type of the allocation.

As the library and toolchain were designed to be drop-in replacements for gcc or clang, I spent little preliminary time studying the library.

Chapter 3

Implementation

The stages of my C compiler backend are as follows:



3.1 `module C`: the extended C abstract syntax tree datatype

I have defined a datatype representing a C-like syntax tree. Not all C features are modelled – only the constructs that my compiler needs to produce.

The tree datatype is more expressive than standard ANSI C. These extra features are rewritten into standard C in a later pass, as detailed later in section 3.4.

The following datatypes are defined for use inside my compiler. Their output representations in the C code are described in later sections.

- `C.ctype` represents a C type. Notable constructors include:
  - `C_Int`;
  - `C_Double`;
  - `C_Pointer of C.ctype`;
  - `C_Value`, a word-sized type capable of representing any OCaml value – described later in section 3.2;
  - `C_Struct of Ident.t * (C.ctype * string) list`;
  - `C_FunPointer of C.ctype * C.ctype list` taking return type and argument types.

  For instance, the type of a function pointer `void (*)(double *)` is represented in my OCaml code as `C_FunPointer (C_Void, C_Pointer C_Double)`.

- `C.expression` represents an expression – program fragments that evaluate to a value (possibly of type `void`). Notable constructors for this datatype include:
  - Data literals of various types:
    - `C_IntLiteral of Int64.t`
    - `C_StringLiteral of string`
    - `C_FloatLiteral of string`[1]
    - `C_CharLiteral of char`
    - `C_PointerLiteral of int`[2]
  - Variables. Two different kinds are distinguished:
    - `C_Variable of Ident.t` takes an identifier (see section 2.1), and is used for all local variables originating from OCaml code. The variables are formatted suffixed with their unique integer: `variablename_1234`.
    - `C_GlobalVariable of string` takes a string and outputs it verbatim – there is no integer suffix. The compiler generates these references when it needs to reference internal functions, such as `ocaml_liballocs_close` (see section 3.10.1). Cross-references to other modules also use these.
  - `C_FunCall of C.expression * C.expression list` represents function calls as an expression evaluating to the function to be called, and the list of arguments. See section 3.6.1.
  - `C_BinaryOp of string * expression * expression` represents all arithmetic and logical binary operations.

  For instance, the expression `foo + 2` is represented as

  ```
  C_BinaryOp (C_Variable foo, "+",
              C_IntLiteral (Int64.of_int 2))
  ```

  (where `foo : Ident.t` is the identifier representing the name of the `foo` variable).

- `C.statement` represents a statement – language constructs such as:
  - `C_If of expression * statement list * statement list`
  - `C_While of expression * statement list`
  - `C_Return of expression option`
  - etc.

  Note that statements typically contain one or more expressions, whereas expressions in standard C cannot contain statements.

[1] Floats are always represented in the OCaml compiler as strings, to prevent rounding errors.

[2] This doesn't take `int64` as might be expected, as it is used to represent lambda's `Const_pointer of int`, which seems only to be used by the compiler to represent `NULL`.

- `C.toplevel` represents a function or global variable declaration/definition at the toplevel.

3.1.1 Inline statements

The fact that C language makes a distinction between "statements" and "expressions" makes it less expressive than OCaml, as all core language features in OCaml can be used as expressions. For instance, consider the translation of the following OCaml code:

```
let x = if foo then (if bar then 1 else 2) else 3
```

Clearly a direct translation of this wouldn't be possible, as `if` statements cannot be used as expressions in C. The ternary operator `?:` exists in C as an expression analogue of `if`. However, it isn't used by my compiler for two reasons. Firstly in the interests of readability – nested ternary operators get hard to read very quickly. The second reason is that debuggability of if statements would be hampered by the fact that debuggers tend not to be able to single-step easily from the condition to the body.

Even if the ternary operator were used, there are other statements that cannot be made into expressions as easily. An example is the let-statement, which translates to variable declarations in C. Hence, a solution to this general problem would still be required.

The way that we handle this is by extending our `C.expression` so that statements can used as "pseudo-expressions". We add a new constructor `C_InlineRevStatements of C.statement` to the `C.expression` type, which allows us to use any block of C statements as if it were an expression.

By doing this, our example from above can be translated directly, simply by wrapping the `C_If` statement up as an "inline statement". The inline statement evaluates to the "value" of the last statement in the block.

We fix up the presence of these in a later phase – see section 3.4.

3.2 Runtime value representation

In the produced C code, `ocaml_value_t` is a type which is capable of representing any OCaml value. This type corresponds to the `C_Value` datatype in my compiler. This type is required to represent polymorphic OCaml types: for instance a function with a parameter of OCaml type `'a` would take a C parameter of type `ocaml_value_t`.[3]

A simple way of implementing this type is to make all OCaml values boxed. This means that everything is allocated on the heap, and the variables of type `ocaml_value_t` are all just pointers into the heap. However this has several significant downsides.

One downside of using boxed objects is that the extra indirection required to access each variable is very slow. Another is that boxing small types, such as integers, requires twice as much memory – one word-sized integer on the heap and the word-sized pointer to it.

[3] As it turns out, it's easier to represent *all* OCaml values as values of this type, not just polymorphic ones.

Because of this, I decided to make my `ocaml_value_t` representation capable of storing some types of value unboxed. However, we would still like `ocaml_value_t` to be exactly 64-bits in size, so that it fits into a register. A technique is needed to squeeze both pointers and *immediates* (unboxed values such as integers and floating point numbers) into just 64 bits. Various such techniques have been used over the years by high-performance compilers and interpreters:

1. **untagged values**, as used by the D programming language, for example. This "technique" simply represents each type of value as itself. Note that there is no way to tell what type the value is just from inspecting the value itself.

   Although OCaml is an example of a statically typechecked language which would still be correct without runtime type information[4], this approach is not used by the upstream compiler. This is because any garbage collector that works with such a value representation must be conservative, which has many disadvantages.

2. **tagged pointers**, as used by OCaml's upstream compiler, the V8 JavaScript engine, and many others. This technique steals the least significant bit of each 64-bit value to use as a "tag bit", signifying whether that value is boxed or not. In OCaml's scheme, an unset tag bit indicates that the value is a pointer, and a set bit indicates an immediate 63-bit integer. The reason this works is because pointers to valid OCaml objects are always 8-byte aligned, which means that all valid pointers have the bottom three bits zeroed. The disadvantage of this representation is that there is no space to represent doubles – they must be boxed.

3. **NaN boxing**, as used by WebKit's JavaScriptCore engine, SpiderMonkey and LuaJIT. This is the scheme I chose to use, because unlike the tagged pointers approach, both immediate integers and doubles can be represented. The details are described in the next section.

3.2.1 NaN boxing

NaN boxing is a surprising technique which, in contrast to the other techniques described previously, has the following desirable properties:

1. has the ability to store IEEE 754 floating point numbers (`doubles`) unboxed;

2. provides additional space to store user-space pointers[5];

3. provides even more additional space (which we use to store 50-bit integers);

4. none of these three ranges overlap: a simple comparison test can be used to distinguish which of the three types a given NaN boxed value is.

[4] Almost – OCaml's "polymorphic comparisons" feature require a limited form of value tagging to support structural comparison.

[5] On 64-bit Linux, user-space pointers are only 48-bits long. The top bits are always zero, to distinguish them from kernel pointers, which have top bits set to one.

NaN boxing allows us to represent doubles unboxed. Our implementation of NaN boxing was inspired by JavaScriptCore's [4], however the exact choice of encoding and implementation details are original.

The representation of IEEE 754 double-precision floats is shown in figure 3.1.

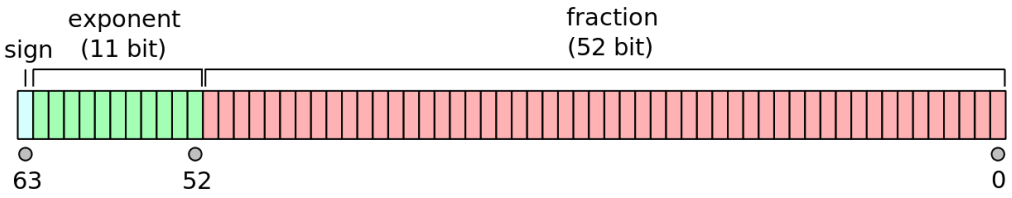


Figure 3.1: The memory format of an IEEE 754 double floating point value.

Not-a-Number (NaN) values in IEEE 754 double values take up a needlessly large range of the 64-bit space. In particular, any of the $2^{53} - 1$ numbers with an exponent of `0x7FF` and non-zero mantissa is a NaN value. However, there are actually only four different NaNs: plus or minus quiet (±qNaN) and signalling (±sNaN). Modern CPUs only ever use one specific (canonical) bit representation for each type of NaN, which leaves the rest of the range to embed other types of values into (see the first two columns of table 3.1).

Before we store the embedded value into the `ocaml_value_t`, we perform one final adjustment: add 2^{48} to the integer representation of the double. This gives us the "in-memory representation" shown in the third column of table 3.1. The purpose of this is so that the final representation of pointers is exactly the same as their raw representation (with top 16 bits clear). This means the common operation of dereferencing a pointer `ocaml_value_t` requires no additional overhead.

Notice that for efficiency, the "in-memory representation" of integers was chosen such that the original integer can be recovered exactly using a single bit-mask operation.

The actual C type definition of `ocaml_value_t` is the following union:

```
union _ocaml_value_t;

typedef union _ocaml_value_t *generic_datap_t;
typedef union _ocaml_value_t (*generic_funcp_t)();

typedef union _ocaml_value_t {
    intptr_t        i;
    generic_datap_t p;
    generic_funcp_t fp;
} ocaml_value_t;
```

Note that we use the `i` field for both integers and doubles. This is because doubles need to be reinterpreted as integers in order to apply the 2^{48} shift as described previously.

Also note that data pointers and function pointers are accessed using different union fields, despite the fact that both are 64-bit pointers into the same address space on Linux.

purpose	double space representation	in-memory representation
−infinity	fff0 000000000000	fff1 000000000000
unused	⋮	⋮
−sNaN (canonical)	fff4 000000000000	fff5 000000000000
unused	⋮	⋮
−qNaN (canonical)	fff8 000000000000	fff9 000000000000
unused	⋮	⋮
Integers (50-bit space)	fffb 000000000000 ⋮ fffe ffffffffffff	fffc 000000000000 ⋮ ffff ffffffffffff
Pointers (48-bit space)	ffff 000000000000 ⋮ ffff ffffffffffff	0000 000000000000 ⋮ 0000 ffffffffffff

Table 3.1: Table showing negative NaN space. There is a similar unused space inside the positive NaN region, but we don't use that. Values which we cannot reuse are coloured grey, and ranges which we've reserved for ourselves are coloured green.

This is a quirk of C: casting between these two types of pointers is undefined behaviour. Interestingly `gcc` will indeed miscompile code that casts betwen these two types.

We define some helper macros in C:

1. `NEW_I`, `NEW_D`, and `NEW_P` allow us to wrap raw integer, double or pointer values into new `ocaml_value_t`. These perform the encoding step as described previously. For instance, `NEW_D` is defined by viewing the double as an integer, and adding 2^{48}, as follows:

```
static intptr_t __double_encode(double x) {
    union {
        double d;
        intptr_t i;
    } s = {.d = x};
    return s.i + (1LL<<48);
}
#define NEW_D(v) ((ocaml_value_t){.i = __double_encode(v)})
```

2. `GET_I`, `GET_D`, and `GET_P` allow us to unwrap an `ocaml_value_t` back into a raw integer, double or pointer value respectively. For instance, `GET_D` is defined as:

```
#define GET_D(v) (__double_decode((v).i))
```

   where `__double_decode` is the inverse of the `__double_encode` operation defined previously.

3. `IS_I`, `IS_D`, and `IS_P` allow us to discriminate which type of value is wrapped inside an `ocaml_value_t`. This is done by simple integer range tests – for instance:

```
#define IS_I(v) \
  ((uintptr_t) (v).i >= 0xfffc000000000000ULL)
```

3.2.2 50-bit integer arithmetic

NaN boxing limits our integers to be 50-bit sized. Luckily, the size of OCaml's standard unboxed integer is deliberately left unspecified – in fact the upstream compiler uses either 31-bit or 63-bit integers depending on the platform.

Interestingly, most native word-sized arithmetic operations work correctly on non-word-sized integers, due to the properties of modular arithmetic and twos complement representation. For instance, consider two 50-bit integers stored in 64-bit registers, with arbitrary values for the top 14 bits. A 64-bit multiply of the two registers will generate the correct answer in the bottom 50 bits of the resulting register.

The only cases where non-word sized integers need careful treatment are operations where upper bits in the input can influence lower bits in the output, such as the right shift and divide operations. OCaml has two types of shift:

- logical shift, where the shifted-in bits are set to zero;
- arithmetic shift, where the shifted-in bits are set to the sign bit.

Implementing logical right shift (lsr) is relatively simple – first mask the top 14 bits to zero, then perform a native-width logical right shift.

Arithmetic right shift (asr) and division are trickier. We first need to "clean up" the top 14 bits by extending our real sign bit (bit 49) into them. This can be achieved using the following C bitfield trick:

```
static intptr_t __sext50(intptr_t x) {
    struct {
        intptr_t x : 50;
    } s = {x};
    return s.x;
}
```

This defines a bitfield integer which is logically 50 bits long. We write `x` into it, and then read out a value of type `intptr_t` (which is 64 bits long). C semantics cause the 50-bit field to be sign extended into the rest of the integer, which is exactly the operation we want.[6]

3.2.3 Casting `ctypes`

In order to work smoothly with both wrapped and unwrapped expressions in my compiler, I define a NaN-box-aware cast operation that transforms between the two when required.

First, define the *kind* of a non-`C_Value` `ctype` as one of the following: integer-like, float-like, data-pointer-like, or function-pointer-like. The operation of casting from an expression of `ctype` "source" into type "target" is defined as below:

[6] The compiler will likely implement this under the hood as `(x lsl 14) asr 14`.

1. If source and target are the same, nothing needs to be done;
2. If source is `C_Value` and target is not, then we need to unwrap the expression. Assuming the cast is valid, then use the appropriate `GET_*` macro to retrieve the original value;
3. If target is `C_Value` and source is not, then we need to wrap the expression using the appropriate `NEW_*` macro;
4. Otherwise use a C cast to the target type.

3.3 `module Translate`: compilation from lambda IR to C AST

The implementation of `Translate` is done via a recursive tree-walk over the `Lambda` tree. Two mutually recursive functions each target the different types of C program fragment:

```
val lambda_to_texpression :
  Lambda.lambda -> C.ctype * C.expression
val lambda_to_trev_statements :
  Lambda.lambda -> C.ctype * C.statement list
```

Note that each translation function also returns the `C.ctype` of the expression or statement. This is for the purposes of casting, as defined earlier. For instance, if an OCaml expression evaluates to an `ocaml_value_t`, and is subsequently used in an if condition, the value must first be unwrapped into a boolean.

Because of the flexibility of the `cast` mechanism, and the expressivity that `C_InlineRevStatements` provides, the actual recursive translation from `Lambda` is actually fairly straight-forward.

Unfortunately the lambda IR is not documented anywhere, so the semantics of its instructions were divined through reverse engineering and code diving.

To illustrate, the following is the implementation of `lambda_to_trev_statements` for OCaml if statements:

```
...
| Lifthenelse (l,lt,lf) ->
  let cexpr_cond = cast C_Bool (lambda_to_texpression l) in
  let trev_t = lambda_to_trev_statements lt in
  let trev_f = lambda_to_trev_statements lf in
  let (ctype, sl_t, sl_f) =
    unify_revsts ~hint:"ifthenelse" trev_t trev_f
  in
  ctype, [C_If (cexpr_cond, sl_t, sl_f)]
  ...
```

(The `unify_revsts` function picks the more general type out of the types of the true and false statement blocks, and casts the other so that it evaluates to the general type. The overall return type of the if statement is the more general one.)

Primitive operations

"Primitive" operations in `Lambda` are represented by

```
Lprim (primitive, args)
```

where `args` are a list of `lambda` terms, and `primitive` is a type of operation. Most primitive operations map fairly simply to C constructs. In total I handle 49 different primitives. Notable `primitives` include:

1. unary and binary operations on integers and doubles;
2. `Praise` which throws an exception (see section 3.9);
3. `Pstringlength` which evaluates to the length of a C-style string. I translate this to a call to `strlen`;
4. `Pmakeblock` which performs an (untyped) allocation of a given size.

   This is more complex to handle than the others, as this primitive specifies the values that the newly allocated memory is initialised to. We need to recursively translate these values, as well as wrapping up the statements required for allocation and initialisation into a single pseudo-expression.

   Things are made even more complex if this `Pmakeblock` references a mutually recursive let-binding. We have to split up the allocation and initialisation phases: see section 3.6.2.

Events

Events are a special `Lambda` constructor which contains type information on the let bindings in scope at the point of the event. These are used by the OCaml debugger.

It's not possible to make use of events during the translation step itself, as an event describing a particular binding can be encountered too late during the recursive traversal. For instance the event describing the types of arguments to a function can be buried inside the body of the function, whereas translation needs to know the type at the point the arguments are first encountered.

We perform an "event scraping" pass before compilation proper. This is a traversal of the entire `Lambda` tree, where all non-event nodes are ignored. Type expressions in all events encountered are cached in a "variable library," a hash table mapping variable identifiers (`Ident.t`) to their types. Once event scraping is complete, we have information about the types of all variables defined in the entire module.

Inline function definitions (lambdas)

OCaml allows one to define new functions anywhere, for example in the following example:

```
let x = (fun x -> x + 1) 41
```

However, C does not – all functions must be defined at the toplevel. The way that we handle this is similar to how we expressionify statements: by introducing an "inline function definition" pseudo-expression. After fixup the function body will get pulled out to the toplevel, and the pseudo-expression replaced with the name of the newly created function.

Note that this does not work if the function is not closed (i.e. depends on values from its environment). In this case we must create a closure at runtime instead, and this is the subject of section 3.10.

3.4 `module Fixup`: translation from extended-C AST to valid C AST

The purpose of fixup is to eliminate inline statements and inline function definitions. Similar to `Translate`, the implementation is done via a recursive tree-walk. Mutually recursive functions each visit different types of nodes:

```
var fixup_expression :
  Fixup.t -> C.statement list -> C.expression ->
  C.statement list * C.expression
var fixup_rev_statements :
  Fixup.t -> C.statement list -> C.statement list ->
  C.statement list
```

Each fixup function takes an accumulator of type `C.statement list`, which is a (reversed) list of statements to be placed before the expression or statement currently being fixed.

As we traverse the AST, fixed statements get pushed onto the front of the accumulator. When we encounter a `C_InlineRevStatements`, we first push the inlined statements onto the accumulator, and replace the expression with the expression that the inlined statements evaluate to. Then, once there are no more statements to fix, we reverse the accumulator and return the result.

To illustrate, consider fixing the following (where square brackets delimit an inline statement block):

```
x = [
  if (a) { 42; } else { foo(); }
];
```

This turns into the following after fixup:

```
int __deinlined;
if (a) { __deinlined = 42; } else { __deinlined = foo(); }
x = __deinlined;
```

Care must be taken to pass a new empty accumulator when entering a new scope, for example when fixing the bodies of if, for and while loops. Also, care must be taken to preserve evaluation order – statements must be consed onto the head of the accumulator in the order that they must be evaluated.

Inline function definitions are easier to handle. All such functions are pulled out and placed at the top of the C file in the order they were encountered. Each "inline function definition" pseudo-expression is replaced with the identifier of the newly-created function.

3.5 `module Emitcode`: outputting C AST to a file

`Emitcode` is responsible for stringifying the AST into valid C code, and outputting this to the target file. The majority of code is straightforward string manipulation, using recursion to traverse the tree. There are two things to note:

1. Not all OCaml identifiers are legal in C – for instance, the apostrophe character is allowed by OCaml. C only allows alphanumeric characters and underscore. We encode illegal identifiers as follows:

   (a) If an identifier contains at least one illegal character, suffix the identifier with an underscore. Because identifiers normally always end with their unique number, encoded identifiers will never collide with a legal identifier;

   (b) For each illegal character, expand it to the character sequence `"_uXXXX_"` where `XXXX` is the hexadecimal character code of the illegal character.

2. Certain variable declarations require special casing to format properly. In particular, a function pointer `x` with type `void (*)()` is not defined as:

   ```
   void (*)() x;
   ```

   as might be expected, but instead as

   ```
   void (*x)();
   ```

3.6 Basic constructs

3.6.1 Functions and complete application

OCaml functions map directly to C functions, with complete function application corresponding to function calls. Currently all functions have arguments and return values of wrapped type `ocaml_value_t`. This is to facilitate calling functions in another module – at the `Lambda` level I do not have type information about functions in other modules.

Mutually recursive functions are simply handled in C. All that is required is that the functions in question are predeclared before they are defined.

Partial application and closures are discussed later in section 3.10.

3.6.2 Mutually recursive definitions

Consider the OCaml expression

```
let rec a = 1::b and b = 2::a in ...
```

This defines a mutually recursive *value* – creating a infinite (cyclic) list of period 2. The value of `a` will be `1::2::1::2::1:: ...`

In order to ensure correct semantics even in the presence of mutually recursive value definitions, my compiler separates the variable definition process into three phases:

1. **Allocation**: first the required memory for each variable are allocated using `malloc`;
2. **Closure creation** (if necessary): at this point the memory addresses of variables are known, and so can be frozen into closures as needed. The process is described in great detail in section 3.10;
3. **Initialisation**: finally, the contents of the variables are set.

The resulting C code for the above OCaml code will look like:

```
// Phase 1: allocation
ocaml_value_t a_1213 = NEW_P(malloc(sizeof(ocaml_value_t)*2));
ocaml_value_t b_1213 = NEW_P(malloc(sizeof(ocaml_value_t)*2));

// Phase 3: initialisation
a_1213[0] = 1;
a_1213[1] = b_1213;
b_1213[0] = 2;
b_1213[1] = a_1213;
```

3.6.3 Range-based for loops

OCaml's for loops are range-based, coming in two varieties:

```
for i = 1 to 10 do ... done
for i = 10 downto 1 do ... done
```

The range limits are inclusive.

This needs to be translated at some point in into C's more general initialisation-condition-update for loop. I chose to reflect OCaml's for loop semantics rather than C's with my `C.expression` constructor, deferring the translation to C style for-loops in the `Emitcode` stage (see section 3.5).

For instance, the first loop from the example above compiles to:

```
for (int64_t i_1203 = 1, _limit_1212 = 10;
     (i_1203<=_limit_1212);
     (++i_1203)) {
  ...
}
```

There's a slight subtlety in that we need to evaluate the range limits exactly once, as this evaluation may incur side effects. Hence the end limit can't be evaluated in the condition part of the C for loop. Instead we define the end limit as an additional variable in the initialisation step, and test against the variable instead.

3.6.4 String switches

The `Lambda` IR has a specific instruction `Lstringswitch` for string matching:

```
match x with
| "foo" -> 1
| "bar" -> 2
| _     -> 3
```

However, C does not have any way to match on strings (`char*`) – the switch statement only works with "integral" types (such as `int` or `char`). The idiomatic way to compare strings is by using the standard library function `strcmp`. Hence, we need to translate this construct into a `strcmp` if-ladder:

```
char *__stringswitch_1235 = x_1234;
if (0 == strcmp(__stringswitch_1235, "foo")) {
  1
} else (0 == strcmp(__stringswitch_1235, "bar")) {
  2
} else {
  3
}
```

Once again we must take care to evaluate `x` exactly once, by assigning the result of evaluation to a variable.

3.7 Standard library: `module Pervasives`

The Pervasives module is the module that is opened by default in every single OCaml program. It exposes (and internally uses) a large number of external functions (such as `caml_sys_exit`), normally provided by the OCaml C runtime library.

I selectively implemented functions that were necessary for the subset of stdlib functionality that was required to run my tests and benchmarks.

3.7.1 Printing to `stdout` and `stderr`

Two sets of C runtime functions had to be implemented in order to support `Pervasives` functions such as `print_int`:

1. string formatting. This is done by implementing functions such as `caml_format_int` and `caml_format_float`, using `snprintf` to perform the formatting;

2. channel operations on `stdout` and `stderr`. Channels are a generic buffered I/O stream interface and can be created inside OCaml code itself. However, I was only interested in supporting the special channels that correspond to the standard I/O descriptors.

   Normally `Pervasives` will get the C library to allocate channel objects for `stdout` and `stderr` at startup, by calling the C runtime function `caml_ml_open_descriptor_out`. However, inside this function we cheat and return opaque (invalid) pointers[7]. Now, when user code tries to print to a channel, the C runtime function simply checks whether the channel is one of the two opaque pointers that we handed out to represent `stdout` or `stderr`. If so, we use `fprintf` to print the string to the correct standard file descriptor.

3.8 Inter-module linking

Recall from section 2.3.2 that inter-module references (such as `List.sort`) are compiled to record field accesses at the `Lambda` level. The top-level module constructor of the currently compiled module returns a block which represents the record to be exported to other modules. Conveniently, this scheme implicitly handles data abstraction for us.

The way that we translate this concept to C is by representing a module `foo` as a C compilation unit `foo.c` exposing the following:

```
ocaml_value_t *Foo;   // the "module object"
void Foo__init();     // the "module constructor"
```

The module constructor function is guaranteed to be idempotent, and after calling it the module object is guaranteed to be valid. Furthermore all toplevel side effects in `foo.ml` are also performed when the module is constructed for the first time.

Note that `Foo` may well depend on other modules, such as `Pervasives`. These references are listed using `Popaque` references at the root of the `lambda` tree. We extract these and translate them into predeclarations of the relevant module objects and constructors. Calls to the constructor functions must be inserted at the start of `Foo__init`.

The C skeleton of a module `Foo` which depends on `Pervasives` is shown in the listing below:

```
#include "liballocs_runtime.h"

void Pervasives__init();
extern ocaml_value_t *Pervasives;

ocaml_value_t *Foo;

void Foo__init() {
    if (Foo) {
        return;
```

[7] Actually we just return the wrapped representation of the integer 1 for stdout, and 2 for stderr.

```
    } else {
        Pervasives__init();

        // allocate and construct module object Foo
    }
}
```

3.9 Exceptions

OCaml has support for exceptions: a `raise Foo` causes the exception `Foo` to propagate up through stack frames until a `try _ with Foo -> _` block is found.

Interestingly, exceptions is one of the cases where the lambda IR is more complex than the OCaml language. Lambda distinguishes between static (local) exceptions, and non-static ones.

3.9.1 Static exceptions

Static exceptions are confined to local lexical scopes – i.e. the raise and the corresponding try are in the same function. The lambda instructions for these are `Lstaticraise` and `Lstaticcatch`. In this case, we can implement these using the C `goto` feature, which allows control flow to jump non-linearly to another point in the same function.

A static exception gets compiled into the following construct:

```
if (1) {
    // body, where raises get transformed into:
    goto label_staticcatch_1;

} else {
label_staticcatch_1:;
    // handler, only reachable by goto
}
```

3.9.2 Non-static exceptions

More generally an exception may unwind through several stack frames, and may be caught in different handlers depending on the dynamic runtime behaviour.

C has a mechanism to perform "non-local" jumps using the special library calls `setjmp` and `longjmp`. A call to `setjmp` will save the CPU registers at the point the function was called (notably, including the instruction and stack pointers) into a `struct jmpbuf` that the user provides. Calling `longjmp` will then restore the state of the program, so that execution flow resumes as if the original `setjmp` returned for a second time. `setjmp` will return non-zero iff it is returning for the second time.

In my scheme, an exception handler installs itself by calling `setjmp`, and then pushing its `jmpbuf` onto the head of a global singly-linked list. When an exception is raised,

`longjmp` is used to "unwind" to the last installed exception handler. The value of the exception is stored in another global variable, so that the exception handler can pattern match against it.

The exception handler is uninstalled (popped off the head of the singly-linked list of handlers) when:

- a try block finishes without raising; or
- the handler starts executing. This prevents exceptions raised inside the handler from incorrectly being recursively handled by itself.

3.9.3 Runtime support

Toplevel handler

If an exception propagates all the way to the top without being handled successfully, the runtime needs to print an error message and abort the program. This is done by installing a catch-all handler in `main` before calling into OCaml code.

Assuming that the module being compiled is called `Foo`, the following is the C `main` function:

```
void Foo__init();

int main() {
    // set up root exception handler
    OCAML_LIBALLOCS_EXN_PUSH();
    if (0 == OCAML_LIBALLOCS_EXN_SETJMP()) {
        Foo__init();
        return 0;
    } else { // catch
        OCAML_LIBALLOCS_EXN_POP();
        fprintf(stderr, "Uncaught OCaml exception: %s\n",
                (const char *)
                  GET_P(GET_P(ocaml_liballocs_get_exn())[0]));
        return 1;
    }
}
```

Builtin exceptions

A number of fundamental exceptions are fairly deep in the OCaml language. OCaml raises some of these exceptions specially for various reasons – for example the `Division_by_zero` OCaml exception is raised when the corresponding FPU exception occurs.

As these exceptions are not defined in the usual way, we must manually declare and instantiate these exceptions in our own C runtime library. The list of predefined exceptions

can be found in `typing/predef.ml`. The C preprocessor quote trick allows us to succinctly define the list of builtin exceptions statically:

```
#define _QUOTE(x) #x
#define QUOTE(x) _QUOTE(x)
#define DEFINE_BUILTIN_EXCEPTION(name) \
    ocaml_value_t __##name[2] = { \
        NEW_P((generic_datap_t) QUOTE(name)), \
        NEW_I(0) \
    }; \
    ocaml_value_t *name = __##name;

DEFINE_BUILTIN_EXCEPTION(Match_failure)
DEFINE_BUILTIN_EXCEPTION(Assert_failure)
...
```

3.10 Closures

C allows function references to be stored in variables, in the form of "function pointers" containing the address of the start of the function's machine code. Calls are performed on function pointers by an indirect jump to the address (using the `call` instruction).

The C language has no notion of closures, which OCaml requires to provide support for lexical scoping of variables in first-class functions. Closures are conceptually a tuple of the function pointer and an environment, which stores the free variables of the closure. Hence, we need to add support for generating closures manually.

One way to represent closures is as "fat pointers", where each closure is stored as a tuple of two pointers. Note however that in OCaml a closure may be used whenever a function is expected, so each indirect function call would now have to check whether the function reference is fat or not – a significant runtime cost.

A unified way to call closures and function pointers is desired. In particular, we'd like the act of closing a C function `f_impl` with an environment `env` to produce a fresh function pointer `f_closure`, such that calling `f_closure(1, 2, 3)` with the standard C calling convention is equivalent to a call to `f_impl(1, 2, 3, env)`.

The technique that we use involves the creation of machine code stubs at runtime. This approach was pioneered by Breuel [3]. However, our implementation was developed completely from scratch, and many details differ from the approach outlined in the paper.

3.10.1 Runtime support

An internal C function `ocaml_liballocs_close(f_impl, n_args, env)` was written, which performs the closing operation as previously described. `f_impl` is the C implementation of the closure – a function which takes `n_args` arguments along with an extra argument containing the environment.

When `ocaml_liballocs_close` is invoked, a small executable stub is written into an executable buffer, with the two pointers `f_impl` and `env` baked in. Then the address of the start of this stub is returned. The stub is described in detail below.

Conceptually, the operation can be described as JIT-compiling the following snippet of pseudo-C code at runtime. (NB: things are a bit more complicated when `n_args` > 5 arguments, as described in the sections below.)

```
ocaml_value_t f_closure(ocaml_value_t arg1,
                        ...,
                        ocaml_value_t argn)
{
    return f_impl(arg1, ..., argn, env);
}
```

The operation of `ocaml_liballocs_close` is to write hard-coded bytes corresponding to the appropriate machine code (listed in the following sections), into an executable buffer. I obtained the machine code by using the `nasm` assembler, and then the `objdump` tool on the produced object file to obtain the bytes representating each instruction. As these are hardcoded into the function, this function currently only works on 64-bit Linux.

Note that this code must be written to a memory page which has both write and execute permissions. This has security implications which mean that additional precautionary steps must be taken if this technique is to be used in production code.

Closing functions that take ≤ 5 arguments

An Application Binary Interface (ABI) specifies a standardised calling convention which programs and libraries adhere to. This includes the specification of the location in registers or memory where function arguments are passed.

C uses AMD's x86-64 ABI on Unix-like systems (notably, not including Windows). The first six arguments are passed in registers in a particular order, shown in the table below. Hence, when closing a function `f` which originally took five or fewer arguments, there is space to pass `env` in another register.

Conveniently, all six argument registers are also caller-preserved whether they are used to pass arguments or not. This means that our stub is allowed to overwrite (or "clobber") any of them without having to clean up afterwards. There are also several other caller-preserved registers, notably `r10`, which we will also make use of.

The following is the table of the register used for each argument in the x86-64 ABI.

nth argument	register
1	`rdi`
2	`rsi`
3	`rdx`
4	`rcx`
5	`r8`
6	`r9`

We shall place `env` into the (`n_args`+1)-th argument. Call the corresponding register `REG_ENV`.

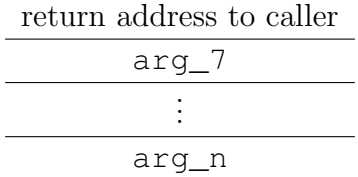
This is the assembly listing for the machine code generated:

```
mov REG_ENV, <env>
mov r10, <f_impl>
jmp r10
```

This works by using two 8-byte immediate `mov` instructions, which each load a fixed constant from the program code, which happen to correspond to our two pointers `env` and `f_impl`. Then `f_impl` is tail-called into.

Closing functions that take > 5 arguments

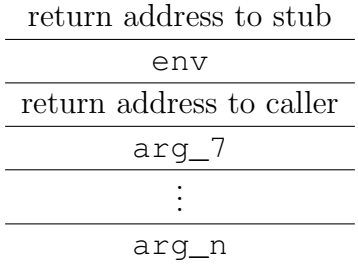
When `n_args` > 5, `env` needs to get passed in on the stack instead. The diagram below shows the layout of the stack when calling an ordinary function with `n` arguments. The head of the stack is at the top of the diagram:



This makes the problem our closure stub has to solve more tricky. Firstly, note that when we modify the stack, our stub can no longer just tail-call into `f_impl` – before returning to the caller we need to undo our stack transformation.

Secondly, implementing the direct solution of placing `env` at the end of the argument list would be extremely painful, as it would involve copying the entire list of arguments one slot upwards, just to make space to put `env` in.

Luckily, I came up with a neat trick which skirts both of these issues and remains performant. It turns out that we can get away with just pushing `env` on top of the stack, before `call`ing into `f_impl`. Observe that the state of the stack on entry to `f_impl` is:



The "return address to caller" will now get passed to `f_impl` as if it were an argument. However, we're free to just ignore it!

This trick only works because our compiler has control over the function signature of `f_impl` – in this case we need to generate the following signature:

```
ocaml_value_t f_impl(ocaml_value_t arg_1,
                     ...
```

```
                    ocaml_value_t arg_6,
                    ocaml_value_t *env,
                    void *unused,
                    ocaml_value_t arg_7,
                    ...
                    ocaml_value_t arg_n);
```

The required machine code stub looks like this:

```
mov r11, <env>
push r11
mov r10, <f_impl>
call r10
pop rcx
ret
```

Note the deliberate choice of `pop rcx` to remove the `env` address from the stack (instead of, for instance `pop r11` or `add rsp, 8`). This is a size optimisation – the chosen instruction encodes in just one byte.

3.10.2 Compiler support

As OCaml immediate value representations are immutable, we can simply copy the values of each free variable into the environment at the time of closure creation.

There is no explicit annotation in the OCaml lambda IR to indicate that a closure must be created – we have to perform free-variable analysis to determine this. We handle three separate scenarios during translation where closures must be created:

- **Function let-binding with non-empty free variable set**:

  ```
  let make_counter_v1 () =
    let ctr = ref 0 in
    let count () = ctr := !ctr + 1; !ctr in
    count
  ```

  This case arises during translation of `Llet` and `Lletrec`. Let-bound functions only need to be closures if their free variable set is non-empty. Otherwise they do not depend on their environment and can simply be represented as an "inline function definition" (section 3.3).

  Otherwise, a new closure implementation function has to be created, and a call to `ocaml_liballocs_close` generated.

- **Anonymous lambda with non-empty free variable set**:

  ```
  let make_counter_v2 () =
    let ctr = ref 0 in
    fun () -> (ctr := !ctr + 1; !ctr)
  ```

This case has to be handled when `Lfunction` is being translated. It is similar to the previous, except that the lambda has no name, so we have to make one up.[8]

- **Partial application**:

```
let make_counter_v3 () =
  let count ctr () = ctr := !ctr + 1; !ctr in
  count (ref 0)
```

This case is triggered whenever `Lapply` is encountered, and the number of arguments given is less than the arity of the function being called. This results in a closure, as an environment is needed to store the partially applied arguments.

A new implementation function is created, and `ocaml_liballocs_close` is called at the point of partial application, which takes an environment containing the partially applied arguments. The implementation function has to extract the partially applied arguments from its environment before calling the original function with all the required arguments (`count` in this case).

Recursive closures

Consider a function `f` referencing itself and `foo`. An environment needs to be created:

`f_env = {f_closure, foo}`

where `f_closure` is the code stub pointer returned by `ocaml_liballocs_close`.

However, this definition is mutually recursive: `ocaml_liballocs_close` needs to know the address of `f_env` in order to create the closure. We use the same technique as we introduced in section 3.6.2 to define these values recursively in a three step process:

1. First `f_env` is allocated, but the contents are left uninitialised;
2. `ocaml_liballocs_close` is called with the address of `f_env`, returning the closure pointer;
3. The contents of `f_env` are then initialised with this closure, along with `foo`.

Mutually recursive closures

Say that function `f1` references `f2` and `foo`, and `f2` references `f1` and `bar`. Currently, two separate environments are created:

- `f1_env = {f2_closure, foo}`
- `f2_env = {f1_closure, bar}`

Where `f1_closure` and `f2_closure` are the code stub pointers with `f1_env` and `f2_env` embedded in them, respectively.

The implementation functions will call into the other function's code stub each time it recurses.

[8] Our naming scheme is `__lambda_1234`.

Possible optimisations

I had some ideas on improving the current implementation of recursive closures, but did not have time to implement these optimisations.

1. An optimisation that could be made is to merge environments for recursive closures.

   ```
   f1_f2_env = {foo, bar}
   ```

   Notice that this allows us to elide the closures themselves from the environments. This is because the implementation function `f1_impl` no longer needs the `f2` closure code stub: it can just call the other with the environment pointer it received itself: instead of `f2(...)` we make a call to `f2_impl(..., env)`.

   This would also allow recursive tail-calls of closures which take more than 5 arguments to work as expected without overflowing the stack. The fact that the closure doesn't call back into its code stub means that no additional stack space needs to be used.

2. Another possible optimisation is that if a closure only requires one environment value, instead of allocating an environment object containing a single reference, the value itself could be passed instead.

Chapter 4

Evaluation

4.1 Compiling the OCaml `List` module

As a demonstration of the capabilities of my OCaml compiler, I am able to directly compile the `List` module from the upstream standard library, with no modifications.

Although many of the functions in the module are very basic (e.g. `hd`), others serve as good demonstrations of recursion, first order functions and closures, such as `filter`, `fold_left`. By far the most complicated function in the module is `sort`, which implements an optimised mergesort.

Compiling the `List` module is not just for demonstration purposes, however. It is a required step in building the compiler, as many benchmarks rely on this module. The way that `List` support is provided to user code is during the linking stage: the user's generated C code is linked against the generated C code of the `Pervasives` and `List` modules.

4.2 Regression testing

As new features have been implemented in the compiler, I've been writing small tests which ensure the correct functionality of that part of the compiler. In total 22 separate test files were written, many of which test multiple variations in a single file. Each commit to the master branch strives to pass all unit tests – works in progress were done on a separate branch before merging into master when tests pass.

4.3 Debugging

I've implemented a function `ocaml_liballocs_recursive_show` which uses liballocs to determine the structure of an arbitrary pointer. It queries liballocs for the size of the allocation that the pointer points into, and uses NaN boxing discriminators to determine how to print the values.

Here is the motivating example `test.ml`, originally given in the introduction:

```
let my_rev lst =
```

```
  match lst with
  | [] -> []
  | x::xs -> List.append xs [x]

let result = my_rev [1; 2; 3]
let result2 = my_rev [1.5; 2.5; 3.5]
let result3 = my_rev [[1, 2], [3, 4]]
```

Our objective is to be able to set a breakpoint in `my_rev`, and be able to see through the polymorphism and print out the value of `lst` at each call.

I compile this using my liballocs toolchain with the following command:

```
/home/debian/liballocs/tools/lang/c/bin/allocscc \
  -DENABLE_LIBALLOCS -D_GNU_SOURCE -g3 -gstrict-dwarf \
  -std=c99 -fno-eliminate-unused-debug-types -o ./test \
  -I../liballocs/include -I. -L../liballocs/lib \
  -L../liballocs/src ./test.c ./stdlib_liballocs/*.c \
  ./liballocs_runtime.c -lm -lallocs
```

Now, using the gdb debugger, I can successfully set breakpoints and use `ocaml_liballocs_recursive_show` to see through polymorphism. (The first two commands are just required to debug using liballocs.)

```
(gdb) set environment
   LD_PRELOAD=/home/debian/liballocs/src/liballocs_preload.so
(gdb) handle SIGILL nostop noprint pass
Signal        Stop      Print   Pass to program Description
SIGILL        No        No      Yes             Illegal
   instruction
(gdb) break my_rev_1199
Breakpoint 1 at 0x41898e: file ./test.c, line 25.
(gdb) run
Starting program: /home/debian/demo/test

Breakpoint 1, my_rev_1199 (lst_1200=...) at ./test.c:25
25      if (((bool)GET_I(lst_1200))) {
(gdb) call ocaml_liballocs_recursive_show(lst_1200)
[ 1, [ 2, [ 3, NULL ] ] ]
(gdb) continue
Continuing.

Breakpoint 1, my_rev_1199 (lst_1200=...) at ./test.c:25
25      if (((bool)GET_I(lst_1200))) {
(gdb) call ocaml_liballocs_recursive_show(lst_1200)
[ 1.500000, [ 2.500000, [ 3.500000, NULL ] ] ]
(gdb) continue
Continuing.
```

```
Breakpoint 1, my_rev_1199 (lst_1200=...) at ./test.c:25
25      if (((bool)GET_I(lst_1200))) {
(gdb) call ocaml_liballocs_recursive_show(lst_1200)
[ [ [ [ 1, 2 ], NULL ], [ [ 3, 4 ], NULL ] ], NULL ]
```

Notice that lists are printed in their in-memory linked list representation, where the empty list is represented as `NULL`. In order to improve the presentation, I'd need to teach liballocs about OCaml types, as liballocs was only designed to model the C type system. Currently I'm only using liballocs to query the size of each allocation. If I had more time I'd like to explore encoding OCaml types better.

During debugging one will notice that the variables and functions have numbers appended to their original names. This is not a big issue while using gdb because it has good tab completion.

By using gdb to debug programs, one also gets to utilise the rest of the power of gdb. For example, backtraces, memory watchpoints, reverse debugging, etc. These all work perfectly with OCaml code compiled to C with my compiler.

4.4 Benchmarks

I've curated a set of benchmarks to test the performance of my compiler under various tests.

My benchmarking methodology relies on a shell script `benchmark.sh`, which compiles each benchmark under twelve configurations:

- Bytecode generated by the upstream `ocamlc` bytecode compiler. These are run with `ocamlrun` (the upstream optimised bytecode interpreter), with default GC settings, and a disabled GC;
- Native code generated by the upstream `ocamlopt` native compiler, with default GC settings, and a disabled GC;
- Native code generated by `gcc` via my OCaml-to-C compiler, at each of four optimisation levels (`-O0`, `-O1`, `-O2`, `-O3`);
- Native code generated by `clang` via my OCaml-to-C compiler, at each of four optimisation levels.

The timings are measured on an otherwise idle computer using the standard `time` tool. Each benchmark aims to take between 0.1–1 seconds to run, as this is the sweet spot where the granularity of the timer is relatively negligible, and a sensible number of repeat runs can be run.

I repeated each run 50 times, generating over 40000 results. These are processed and graphed using R and ggplot2. I had to learn this language and library from scratch, but it turned out to be a powerful tool for processing the large amounts of CSV data my benchmarking script generates.

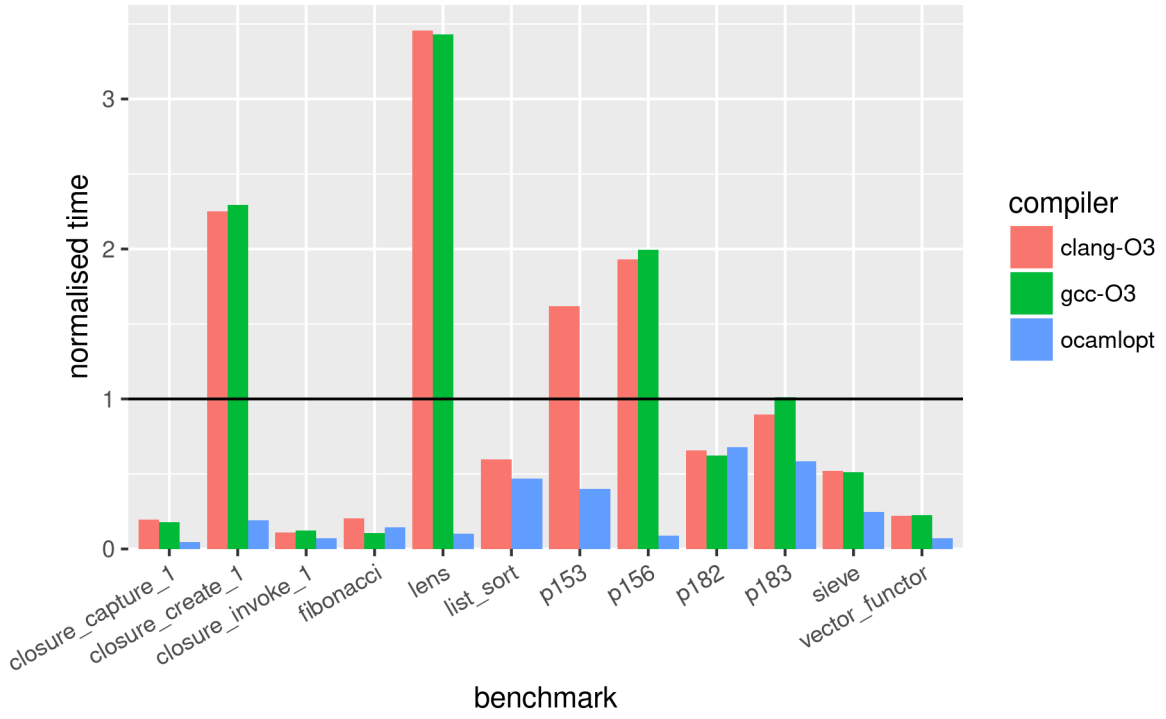


Figure 4.1: Summary graph of performance for optimised configurations, relative to the baseline "ocaml" bytecode compiler. The horizontal line at 1.0 normalised time represents this baseline.

My benchmarks broadly fall into one of three categories, and each benchmark described in its own section below. The results are summarised in table 4.1.

Broadly speaking, the C compilers usually exceed the performance of the OCaml bytecode compiler, apart from in a few select workloads. The OCaml native compiler beats my C compiled code on all except one benchmark.

I ran a configuration where the OCaml GC was disabled. Interestingly, this never significantly improved performance, and in fact hindered performance significantly on some benchmarks. I'm not completely certain of the reason behind this – I'm fairly certain that memory was not swapped to disk, so likely this is just due to poor memory locality caused by a huge heap.

4.4.1 Microbenchmarks

These microbenchmarks written from scratch by me. These test particular aspects of the performance of the generated code.

1. The **`closure_create_<n>`** family of benchmarks test the performance of creating closures which take n arguments and capture 1 environment value.

   There are three interesting things to note from the results, graphed in figure 4.2:

   (a) performance is fairly constant for all values of n, for all compilers;

(b) however, creating closures is more than twice as slow for C compiled programs compared to the bytecode compiler. This is likely due to the overhead incurred in writing the 23/27 byte machine code stub for each closure;

(c) there is a small speed penalty when $n > 5$. This corresponds to the case where extra machine code needs to be emitted to perform stack operations.

It turns out that there is a long-standing bug in gcc which causes it not to merge sequential byte-sized store operations into a single wider store. This was verified by observing the generated assembly using Godbolt's compiler explorer [5]. The clang compiler does not suffer from this same issue.

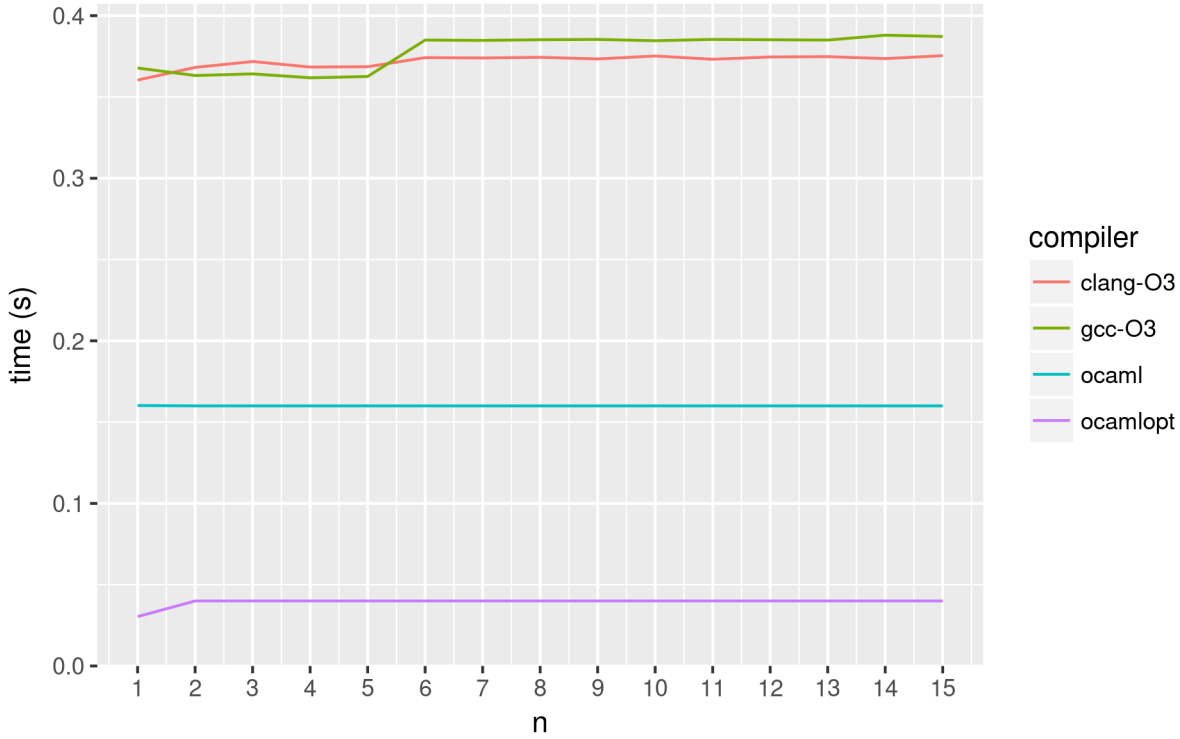


Figure 4.2: `closure_create_<n>` results.

2. The **`closure_invoke_<n>`** family of benchmarks the performance of invoking closures which take n arguments and capture 1 environment value.

   The results are graphed in figures 4.3 and 4.4. From the first graph one can see that the OCaml bytecode is extremely slow (by a factor of more than 9 compared to the C compiled code) at invoking closures. It's more useful to look at the second graph, which only plots the optimised native code compiler configurations.

   The gcc results seem to be very noisy. However, it turns out that the standard error for each particular value of n is extremely low, indicating that this "noise" seems to be deterministic and reproduceable. These are likely caused by cache-related effects.

   It's easier to draw conclusions from the clang results. When $n \leq 5$, everything is passed in registers. This is effectively constant time, as expected. Invoking closures

which take $n > 5$ arguments requires the compiler to place $n - 6$ arguments on the stack.

Interestingly, `ocamlopt` seems to be able to execute the closure in constant time regardless of n. I suspect this is because the compiler backend has managed to optimise out the call through inlining. The gcc and clang compilers cannot see through the closure operation, so unfortunately they cannot perform same inlining operation.

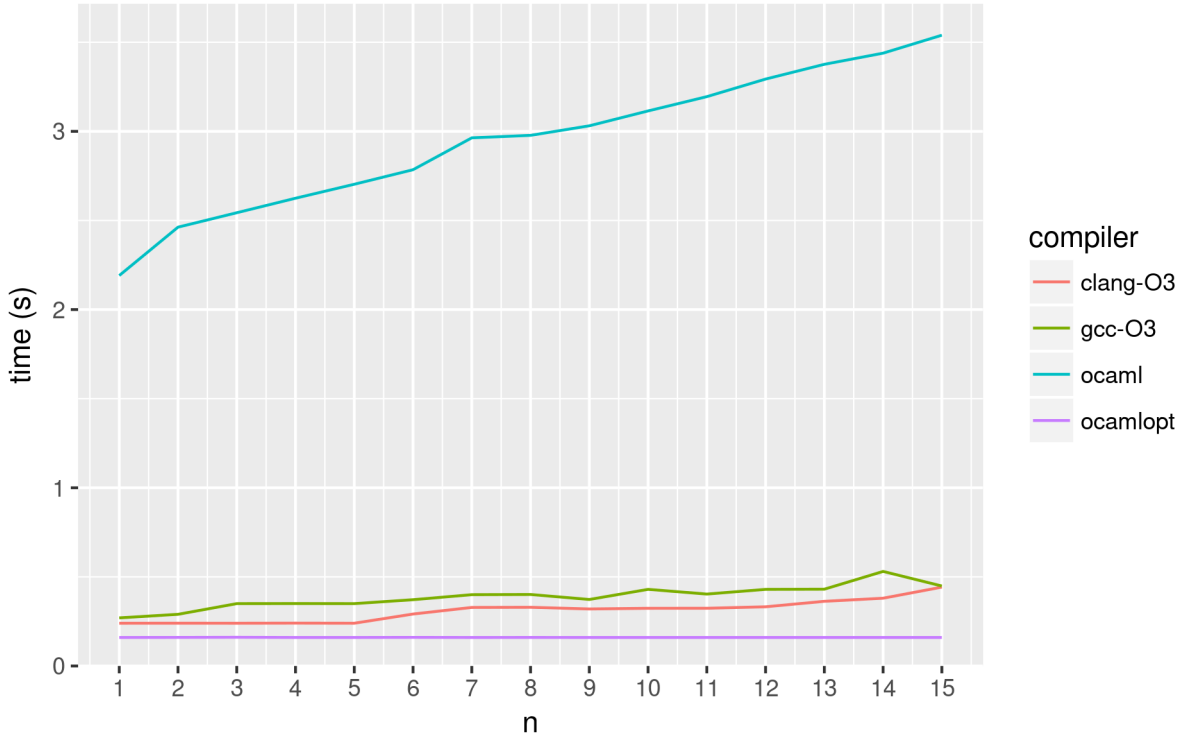


Figure 4.3: `closure_invoke_<n>` results.

3. The **`closure_capture_<n>`** family of benchmarks the performance of invoking closures which take 1 argument and capture n variables from their environment.

   As my C compiler generates code which simply copies each captured variable into the environment object, I would expect time to increase linearly with n. This is indeed observed for each compiler configuration, although interestingly gcc seems to have a slope which is shallower than all others for the range of n tested.

4. The **`list_sort`** benchmark was written from scratch, to test the performance of the unmodified pure-OCaml standard library `List.sort` routine (see section 4.1).

   The routine only creates a constant number of closures, but makes heavy use of closure invocation and allocates heavily.

4.4.2 `operf-micro` benchmark suite

These are various sized benchmarks adapted from the OCamlPro's `operf-micro` benchmark suite. These needed adapting before they could be used, due to being dependent

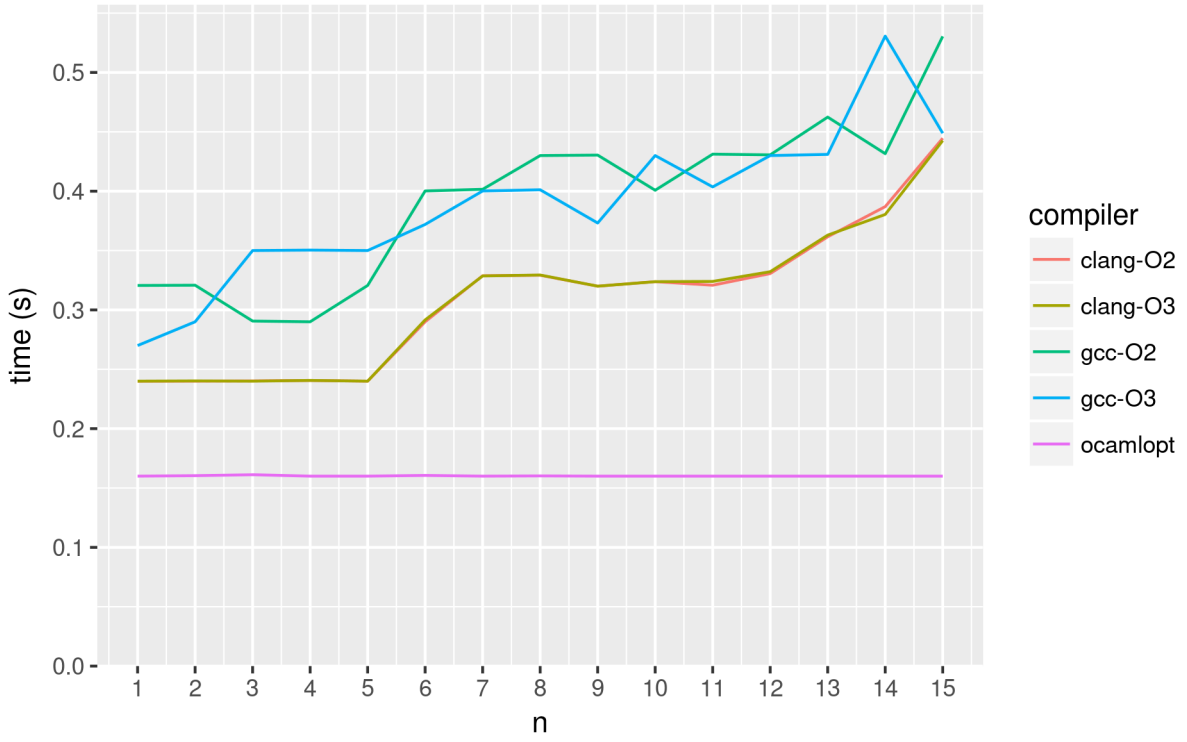


Figure 4.4: `closure_invoke_<n>` results (optimised compilers only).

on an OCaml-side framework which performs timing and measures GC statistics, features which are not supported by my compiler.

1. The **`fibonacci`** benchmark was adapted from the `fibonnaci` [*sic*] benchmark in `operf-micro`.

   This simple microbenchmark implements the naive (non-memoising) recursive Fibonacci function, and uses it to evaluate the 40-th Fibonacci number.

   The benchmark primarily stress-tests the speed of function calls, as the cost of invoking recursive calls dominates the amount of work done inside each call.

   C compilers do an extremely good job with gcc beating out even the native OCaml compiler.

2. The **`lens`** benchmark implements a Haskell-style lens (otherwise known as functional references), and then uses lenses to operate on a `rectangle` record type.

   This makes usage of records and higher order functions, creating a very large number of closures.

   C compilers do unusually poorly on this benchmark; this motivated the investigation in section 4.4.4.

3. The **`sieve`** benchmark implements a linked-list sieve of Eratosthenes.

   This benchmark performs many list operations and is very allocation-heavy.

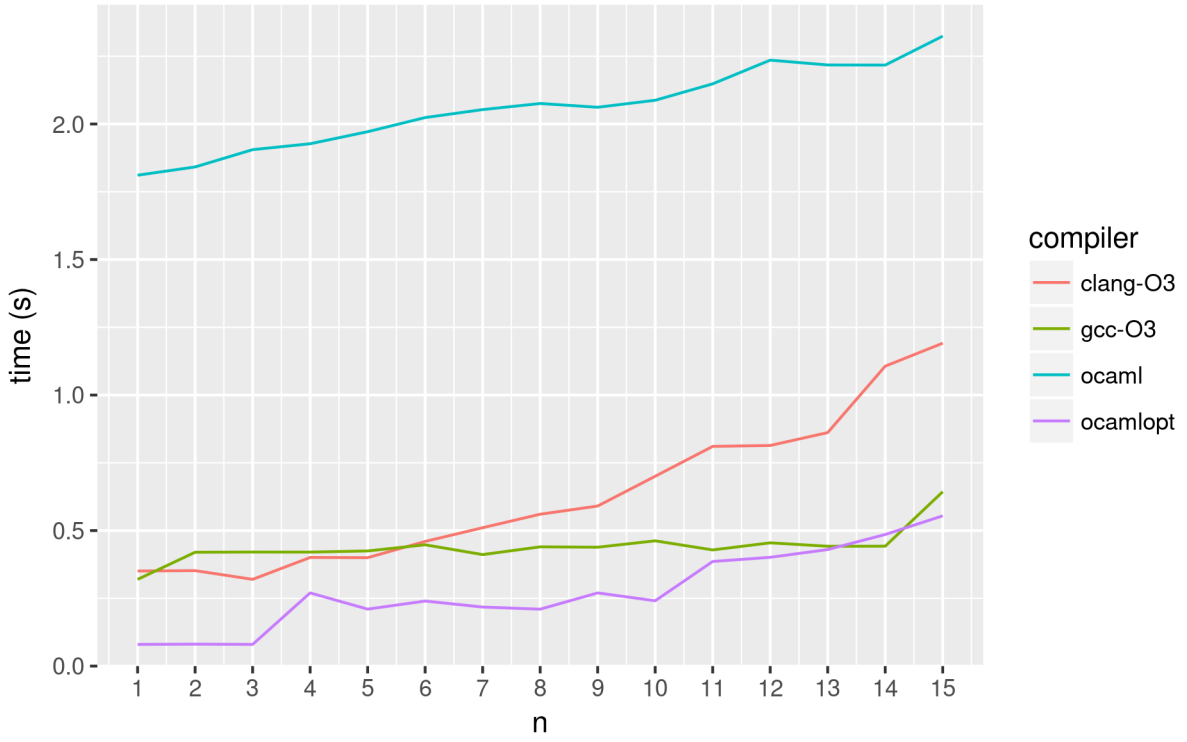


Figure 4.5: `closure_capture_<n>` results.

4. The **`vector_functor`** benchmark implements 2D and 3D vectors in two different styles – directly and by using a generic functor. It then uses these implementations to create and calculate the dot product of 10^6 2D and 3D vectors.

   This benchmark allocates many records, and performs floating point operations.

4.4.3 Macrobenchmarks

These are larger benchmarks written by me, as solutions to Project Euler problems.

1. The **`p153`** benchmark makes usage of exceptions to break out of a recursive loop early.

2. The **`p156`** benchmark makes heavy use of recursion, closures and imperative-style `for` loops.

3. The **`p182`** benchmark makes use of a tail-recursive GCD implementation, reference updates and imperative-style `for` loops.

4. The **`p183`** benchmark makes heavy use of both integer and floating point arithmetic (`log`, `exp`, floating point division).

4.4.4 Investigating performance discrepancies

Segmentation fault

The `list_sort` and `p153` benchmarks fail with a segmentation fault when compiled with `clang -O0` or `gcc` at all optimisation levels. The cause of this is a stack overflow from a lack of tail-call optimisation.

On further investigation, I found the root cause, illustrated by the following C function:

```
int f(int x) {
  int ret;
  if (g(x)) {
    ret = f(x + 1);
  } else {
    ret = 42;
  }
  return ret;
}
```

With a small amount of optimisation, `f(x + 1)` can be moved into tail call position. It is somewhat surprising that GCC fails to recognise and take advantage of this.

Instrumentation

In order to dig deeper into why certain benchmarks seem perform significantly worse than others, I decided to create a branch of my compiler with an instrumented runtime which reported the following runtime statistics. If it turned out that relative program performance were correlated with that program a specific high statistic, this would point at a part of the runtime where my C implementation were weaker than OCaml's.

1. **number of allocations performed**. This is achieved by hooking into the `malloc` function, by adding the following to the `liballocs_runtime.h` header file:

   ```
   extern int n_allocations;
   #define malloc(x) (++n_allocations, malloc(x))
   ```

   (The counter `n_allocations` is instantiated in `liballocs_runtime.c`.)

2. **number of closures created**. This is achieved by hooking into the `ocaml_liballocs_close` function, incrementing a counter each time the function is called.

3. **number of times a closure was invoked**. This is tricker to instrument because unlike the previous two, there is no common function which is called when a closure is invoked. The code that we must insert instrumentation into is the generated code stub itself.

   In `ocaml_liballocs_close`, we prefix the following two instructions to each generated code stub:

```
mov r11, <n_closures_invoked>
add dword [r11], 1
```

where `n_closures_invoked` is the address of the counter variable.

The results are shown in table 4.2.

I discovered a strong correlation between a benchmark's slowness and its rate of closure creation (total number of closures created divided by the absolute amount of time taken). This is clearly seen in the scatter diagram in figure 4.6.

I can conclude from this that the closure creation process (`ocaml_liballocs_close`) is likely the weak link in my runtime, in comparison to upstream OCaml. It would be interesting to see whether this could be optimised, and what effect this could have on the benchmarks.

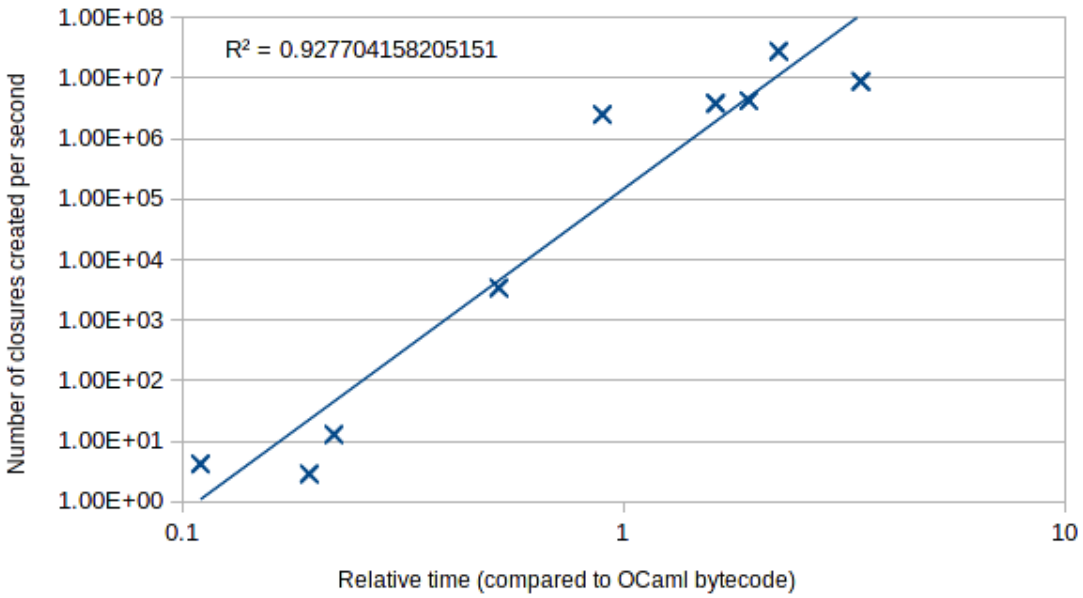


Figure 4.6: A scatter plot of all benchmarks: plotting the normalised time against the rate of closure creation.

benchmark	clang -O0		clang -O1		clang -O2		clang -O3	
	mean	sd	mean	sd	mean	sd	mean	sd
closure_capture_1	1.635	0.010	0.560	0.005	0.350	0.000	0.351	0.006
closure_create_1	0.509	0.050	0.366	0.026	0.361	0.002	0.360	0.002
closure_invoke_1	4.084	0.582	0.700	0.001	0.240	0.000	0.240	0.000
fibonacci	5.226	0.011	1.310	0.000	0.651	0.006	0.650	0.002
lens	1.827	0.009	1.326	0.007	1.356	0.030	1.360	0.031
list_sort	N/A	N/A	0.794	0.053	0.673	0.009	0.671	0.005
p153	N/A	N/A	0.687	0.010	0.652	0.012	0.649	0.012
p156	3.247	0.005	2.072	0.012	1.976	0.009	1.977	0.011
p182	1.928	0.005	0.762	0.004	0.740	0.003	0.731	0.002
p183	0.853	0.005	0.500	0.002	0.400	0.001	0.400	0.003
sieve	2.580	0.026	1.594	0.009	1.441	0.008	1.441	0.011
vector_functor	4.368	0.011	1.741	0.007	1.329	0.008	1.327	0.036

benchmark	gcc -O0		gcc -O1		gcc -O2		gcc -O3	
	mean	sd	mean	sd	mean	sd	mean	sd
closure_capture_1	1.330	0.014	0.422	0.006	0.372	0.004	0.320	0.000
closure_create_1	0.472	0.005	0.361	0.005	0.364	0.005	0.368	0.032
closure_invoke_1	1.809	0.041	0.240	0.000	0.321	0.004	0.270	0.000
fibonacci	4.291	0.019	0.671	0.004	0.711	0.004	0.328	0.004
lens	1.639	0.010	1.365	0.052	1.350	0.015	1.350	0.052
list_sort	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
p153	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
p156	2.895	0.009	2.124	0.008	2.055	0.008	2.043	0.006
p182	1.551	0.005	0.810	0.002	0.772	0.007	0.691	0.006
p183	0.781	0.003	0.490	0.001	0.480	0.002	0.450	0.003
sieve	2.305	0.028	1.514	0.009	1.462	0.009	1.416	0.008
vector_functor	3.468	0.011	1.423	0.014	1.369	0.008	1.350	0.008

benchmark	ocaml		ocaml no GC		ocamlopt		ocamlopt no GC	
	mean	sd	mean	sd	mean	sd	mean	sd
closure_capture_1	1.811	0.005	1.819	0.042	0.080	0.000	0.080	0.000
closure_create_1	0.160	0.001	0.160	0.000	0.030	0.002	0.020	0.000
closure_invoke_1	2.191	0.005	2.194	0.009	0.160	0.000	0.161	0.006
fibonacci	3.165	0.010	3.166	0.013	0.460	0.002	0.461	0.006
lens	0.393	0.010	0.402	0.007	0.040	0.000	0.040	0.000
list_sort	1.122	0.016	1.460	0.036	0.527	0.022	0.846	0.036
p153	0.401	0.003	0.402	0.004	0.160	0.001	0.160	0.000
p156	1.025	0.010	1.039	0.011	0.090	0.001	0.100	0.000
p182	1.111	0.004	1.113	0.009	0.752	0.004	0.752	0.004
p183	0.445	0.008	0.445	0.010	0.260	0.001	0.260	0.001
sieve	2.762	0.061	6.612	0.076	0.684	0.005	3.538	0.082
vector_functor	6.002	0.087	6.216	0.079	0.427	0.007	0.442	0.006

Table 4.1: Summary table of results for all benchmarks. The mean and standard deviations of 50 repeat runs are shown. Performance is colour coded relative to the "ocaml" bytecode benchmark results – green indicates a result that is faster than this baseline, and red slower.

benchmark	allocations	allocations/s	closures created	closure created/s	closures invoked	closures invoked/s
closure_capture_1	3	8.55	1	2.85	1.00×10^8	2.85×10^8
closure_create_1	1.00×10^7	2.78×10^7	1.00×10^7	2.78×10^7	0	0
closure_invoke_1	5	20.8	1	4.17	1.00×10^8	4.16×10^8
fibonacci	9	13.8	0	0	0	0
lens	1.90×10^7	1.40×10^7	1.20×10^7	8.82×10^6	4.00×10^6	2.94×10^6
list_sort	2.09×10^7	3.11×10^7	5	7.45	2.05×10^7	3.06×10^7
p153	4.48×10^6	6.90×10^6	2.48×10^6	3.82×10^6	1.44×10^7	2.22×10^7
p156	1.85×10^7	9.36×10^6	8.41×10^6	4.25×10^6	3.80×10^7	1.92×10^7
p182	10	13.7	0	0	0	0
p183	3.00×10^6	7.50×10^6	1.00×10^6	2.50×10^6	2.00×10^6	5.00×10^6
sieve	3.91×10^7	2.72×10^7	4.90×10^3	3.40×10^3	3.86×10^7	2.68×10^7
vector_functor	4.00×10^7	3.01×10^7	17	12.8	4.00×10^7	3.01×10^7

Table 4.2: Instrumentation results. The per-second results are obtained by dividing the absolute values by the `clang -O3` run time in seconds.

Chapter 5

Conclusion

The project was technically challenging, as I had to work with an unfamiliar compiler's internals, as well as implementing a new backend from scratch.

My original motivations for this project were the following:

1. **Feasibility**: my project was completely successful in demonstrating that compiling OCaml into C code is a feasible task. Many OCaml constructs are already supported, and the ones that are not (detailed below) were mostly omitted due to time constraints, rather than technical ones;

2. **Performance**: despite not worrying specifically about generating performant C code in my compiler, the benchmarks show that the compiled C executables already perform competitively – matching or exceeding OCaml bytecode performance in most cases, when closure creation doesn't dominate runtime;

3. **Observability**: this was one of the weaker points of my project. Although observability through parametric polymorphism was achieved (section 4.3), I did not teach liballocs enough about OCaml types, which meant that values had to be printed as their raw in-memory representation. In hindsight I would have allocated more time working with `Liballocs` to get the best results here, instead of focusing so much on making the compiler as complete as possible;

4. **Portability**: my closure implementation in its current state hinders portability by assuming the generated code is being run on 64-bit Linux. Because of this I somewhat regret my choice of implementing closures from scratch – if I had taken advantage of a widely-ported library such as `libffi`, then portability would have been completely achievable.

There are still many limitations on my compiler that I'd hypothetically like to fix:

- Currently "overapplication" is unsupported. This is when the number of arguments provided to a function application exceeds the arity of the function being applied. This is illustrated by the following example:

```
let f x = fun y -> 42
let overapplied = f 10 20
```

The C code this translates to would have to split this into two function calls, one after the other.

The mechanism for detecting this situation is actually implemented in my compiler, however I did not have time to implement the actual translation.

- Block tags are not supported, which basically means that variant constructors which take arguments (e.g. `Foo of int | Bar of string`) cannot currently be created. The issue is that my blocks do not have headers, so there is no space to store the tag byte. There are three possibilities I can think of to implement this, each with various tradeoffs:

  1. add a header to all tags. This could cause a significant memory requirement increase;
  2. use a field in `liballocs`'s `uniqtype` structure, which is used to describe a particular allocation. This is space efficient because `uniqtypes` are created per allocation-site rather than per runtime allocation. One downside is that `liballocs` would become a hard dependency (which it currently isn't), which may hinder portability somewhat.
  3. use some spare bits in the NaN boxed value representation to store the tag. By squeezing out every single bit we possibly can, we can theoretically recover 6 bits for a tag (3 bits from the 8-byte alignment requirement on pointers, 1 sign bit and 2 mantissa bits from filling up the unused range as fully as possible). The advantage of this approach is that no extra memory is required at all. However the (significant) downside is that pointers must be masked before dereference, which likely incurs a significant performance penalty.

- Large parts of the standard library are unimplemented. Significantly `Array` is missing, which provides a fixed-size contiguous list datatype.

  In fact, the `array` datatype is builtin to the compiler, and there are many primitive `Lambda` operations that operate on these. I did not have time to implement these. (This task is made even more tricky by the fact that there is a different layout of `array` for doubles, and this optimisation is embedded deep into the compiler, bringing with it its own set of primitive operations and quirks.)

- Currently OCaml's polymorphic comparisons are unsupported, but all of the underlying techniques required to implement this are in place already. An implementation of this would use the `IS_*` discriminating macros on NaN-boxed values to perform the correct comparison operation for the type of value stored.

Bibliography

[1] D. Tarditi, A. Acharya, P. Lee, *No Assembly Required: Compiling Standard ML to C*, November 1990.
`http://repository.cmu.edu/cgi/viewcontent.cgi?article=3011&context=compsci`

[2] Stephen Kell, *liballocs*,
`https://github.com/stephenrkell/liballocs`

[3] Thomas Breuel, *Lexical Closures for C++*, In Proc. USENIX C++ Conf., pages 293-304, Denver, CO, October 1988.
`http://www.cl.cam.ac.uk/~srk31/teaching/redist/breuel88lexical.pdf`

[4] WebKit authors, *JSValue.h*.
`https://github.com/WebKit/webkit/blob/master/Source/JavaScriptCore/runtime/JSCJSValue.h`

[5] Matt Godbolt, *Godbolt compiler explorer*.
`https://godbolt.org/`

[6] Stephen Dolan, *Malfunctional Programming*, ML Workshop, 2016.
`https://www.cl.cam.ac.uk/~sd601/papers/malfunction.pdf`

Appendix A

Project Proposal

Computer Science Tripos – Part II – Project Proposal

An observable OCaml, via C and `liballocs`

Cheng Sun, Churchill College

Originator: Stephen Kell

April 28, 2017

Project Supervisor: Stephen Kell

Director of Studies: John Fawcett

Project Overseers: Timothy Griffin & Pietro Lio

Introduction

OCaml is one of the most commonly used members of the ML family of functional languages. It is popular for its expressivity, type system and performance. However, there is not as of yet a good story for debugging OCaml programs, and observing their behaviour at runtime.

The OCaml bytecode debugger, `ocamldebug`, forms part of the core OCaml toolchain. One major problem is that `ocamldebug` is unable to "see through" polymorphism. For instance, suppose we would like to debug the following polymorphic list-reverse function, which has type `'a list -> 'a list`:

```
let my_rev lst =
  match lst with
  | [] -> []
  | x::xs -> List.append xs [x]

let result = my_rev [1; 2; 3]
```

Now let's try to debug the function with `ocamldebug`.

```
$ ocamlc -g -o my_rev my_rev.ml
$ ocamldebug my_rev
        OCaml Debugger version 4.02.3

(ocd) break @ My_rev 1
```

```
Loading program... done.
Breakpoint 1 at 21600: file my_rev.ml, line 2,
   characters 3-62
(ocd) run
Time: 12 - pc: 21600 - module My_rev
Breakpoint: 1
2   <|b|>match lst with
(ocd) print lst
lst: 'a list = [<poly>; <poly>; <poly>]
```

Note that `ocamldebug` is unable to display the contents of the input list, as it does not know the concrete type that the type variable `'a` is instantiated with for this invocation.

There are many other deficiencies with the OCaml debugger, mostly stemming from its immaturity and lack of features. These issues mean that when debugging OCaml code, one often has to resort to "printf debugging" instead.

The aim of this project is to investigate whether the experience of debugging and observing the runtime behaviour of OCaml programs could be improved by utilising the mature C toolchain. My goal is to write a translator that compiles OCaml code into equivalent C code, whilst maintaining a well-defined mapping between the two (in terms of variable names, types, and so on). A user will then be able to debug their OCaml program by using (perhaps an augmented) `gdb` on the generated C code.

In order to solve the problem of "seeing through" polymorphism, I will use a library written by my supervisor, `liballocs`, which keeps track of runtime allocation metadata (including their types) with low overhead. This will allow the debugger to inspect the allocation corresponding to the input to `my_rev`, for instance, and conclude that it is operating on lists with elements of type `int`.

Starting point

The project will be focused on creating a new "backend" for the OCaml compiler, so I will build on the existing code of the frontend. This includes reusing code for lexing, parsing and typing of OCaml programs, as well as the transformation from the typed AST to the Intermediate Representation that we will be using as input.

In order to provide a standard library to compile programs against, I will try to use as much of the existing OCaml `stdlib` as possible. If this turns out not to be feasible then I will resort to writing a subset of the standard library by hand.

I will use the open-source library `liballocs`, written by Stephen Kell. This library provides routines to tag regions in virtual memory ("allocations") with run-time type information.

I may also make use of some further open-source libraries such as:

- `libffi` – a library providing a portable way to perform a call to a function with a foreign function interface (such as the closures that I will dynamically create at runtime);
- `cil` – a library written in OCaml providing a framework for the manipulation of C programs.

There exists similar prior work [1] that shows that the overall project concept is feasible. However, I will be making different design choices to the compiler presented in the paper, due to differing requirements. For instance, the presented compiler generates code that uses continuation-passing style, which I would like to avoid doing, as it harms observability – backtraces would no longer be directly meaningful.

Resources required

The project will primarily be developed on my personal computer for convenience. The computer has a 4-core 3.30GHz Intel Xeon CPU, a 250GB SSD and 8GB of RAM. I am using Arch Linux. No other non-standard equipment is anticipated to be required, and I can easily continue development on MCS machines if required.

The project will be synced to a git server (likely GitHub), and backed up regularly to my MCS network drive.

The project will utilise various standard open-source software packages, including the usual C toolchain (e.g. `gcc`, `gdb`, `make`) along with the OCaml compiler. (As mentioned previously, `liballocs` requires some patches to `binutils`, but these can easily be built.)

Work to be done

The project can be split up into the following tasks:

1. Performing a study of the starting point (OCaml compiler frontend and `liballocs`). This includes an investigation into the forms of the various Intermediate Representations of the OCaml compiler, and selection of the best IR to use as input for our project.

2. Choosing a suitable object representation for boxed objects in memory. There are a variety of alternatives to OCaml's own tagged-pointer representation (which leads to awkward 63-bit integers). This choice requires consideration of the garbage collection strategy, even if a GC is not within the core scope of this project.
3. Implementing the translation from OCaml IR to C (with `liballocs`), for a variety of language features:
   (a) Fundamental types: `bool`, `int`, `float`, `list`, `ref`, ...;
   (b) Tuples and records, which can be implemented using C `structs`;
   (c) (Non-polymorphic) variants, which can be implemented using tagged C `unions`;
   (d) Polymorphic types, which can be represented safely using `void *` opaque pointers, as downcasts are never required in ML (although `liballocs` could provide the necessary information);
   (e) Parametrically polymorphic functions;
   (f) First class functions and closures, which require lambda lifting, or more generally something like the technique of Breuel [2] (dynamically creating an executable stub associated with each closure at runtime).
4. Integrating a subset of the OCaml `stdlib`, or writing my own as necessary.
5. Writing a runtime library to support dynamic closure creation.
6. Creating a corpus of test and benchmark programs.
7. Evaluation, as described in the *Success criteria* section below.
8. Extensions as time permits, as described in the *Possible extensions* section below.

Success criteria

My project will have been successful if I have managed to create a OCaml-to-C translator that works on a commonly used subset of the OCaml language (including closures, parametrically polymorphic functions). Furthermore the polymorphism must be able to be seen through (solving the problem highlighted in the *Introduction* section, for instance).

The project will be evaluated in two ways. Both will require a corpus of OCaml tests and micro-benchmarks.

Firstly, I will measure the "observability" of the compiled C programs compared with their native and bytecode OCaml counterparts. For instance, one metric would be obtained by interrupting these programs at pre-determined points in the code, and counting the number of local variables that could be recovered from the stack frames. This is an accurate metric for observability, as the utility of debugging is determined almost completely by the amount of state that is visible at a breakpoint.

Secondly, I will measure the performance of the resultant C executables, by timing benchmark programs as compiled by both compilers. As the primary objective of the project is debuggability rather than performance, I do not expect the performance to be anywhere near as good as code generated by the OCaml native-code compiler: I will be satisfied as long as the performance is within a reasonable factor for the majority of use cases. It is still necessary to evaluate performance, because it affects the debuggability of CPU-intensive programs.

There are many other qualitative evaluations that I will also consider, such as the ease of interoperability with C libraries using this approach, the ability to insert instrumentation into the code, and the overall debugging experience when using `gdb` on the resultant executables.

Possible extensions

- Improving the runtime, such as adding a garbage collector;
- Adding further language and standard library support, to increase the range of programs supported;
- Optimising the performance of the generated program;
- Improving the debugging experience (such as augmenting `gdb`).

Timetable

1. **Michaelmas weeks 2–3:** Dig into the OCaml compiler frontend; study its Intermediate Representations. Learn to use the `liballocs` library.
2. **Michaelmas weeks 4–6:** Begin work on translator, initially targeting basic features (fundamental types, monomorphic functions). Design an initial representation for boxed objects.
3. **Michaelmas weeks 7–8:** Add support for tuples, variants, records. Test against very simple programs.

4. **Michaelmas vacation weeks 1–2:** Add support for polymorphic functions and types.
5. **Michaelmas vacation weeks 3–4:** Get a small, commonly-used subset of the standard library working. Start creating corpus of test programs, and begin testing.
6. **Michaelmas vacation weeks 5–6:** Reserved for holidays/revision.
7. **Lent term weeks 0–1:** Add initial support for dynamic closure creation. Complete corpus of test OCaml programs; begin evaluation. Start progress report.
8. **Lent term week 2–3:** Milestone: compiler has working support for most test programs (demoable). Submit progress report. Prepare and give presentation.
9. **Lent term weeks 4–6:** Testing; fix bugs. Improve closure creation, and evaluate. Investigate "source maps" to map C code to corresponding lines in the OCaml source.
10. **Lent term weeks 7–8:** Further evaluation, and work on improving evaluated metrics. Start on main chapters of dissertation.
11. **Easter vacation weeks 1–3:** Milestone: compiler works for all intended features. Wrap up evaluation. Continued work on dissertation.
12. **Easter vacation weeks 4–5:** Reserved for holidays/revision. Extension work if time permits.
13. **Easter term weeks 0–2:** Form conclusion and complete dissertation. Time may be limited due to revision.
14. **Easter term week 3:** Proof reading and submission of final dissertation.

References

[1] D. Tarditi, A. Acharya, P. Lee, *No Assembly Required: Compiling Standard ML to C*, November 1990.
`http://repository.cmu.edu/cgi/viewcontent.cgi?article=3011&context=compsci`

[2] Thomas Breuel, *Lexical Closures for C++*, In Proc. USENIX C++ Conf., pages 293-304, Denver, CO, October 1988.
`http://www.cl.cam.ac.uk/~srk31/teaching/redist/breuel88lexical.pdf`