Tianlin Zhang

An Observable OCaml

Computer Science Tripos – Part II

Jesus College

March 30, 2018

Proforma

Name:	**Tianlin Zhang**
College:	**Jesus College**
Project Title:	**An Observable OCaml**
Examination:	**Computer Science Tripos – Part II, June 2018**
Word Count:	**TODO**[1]
Project Originator:	Stephen Kell
Supervisor:	Stephen Kell

Original Aims of the Project

To write a backend for the OCaml compiler that allows compilation from OCaml into C in such a way that it preserves so-called "observability", that is the ability to debug the generated C code, using GDB for example, in a similar fashion to debugging the original OCaml code. A library of test programs should be written to demonstrate the capabilities of the compiler, along with an evaluation into the performance and observability of the compiled code.

Work Completed

I have written and completed a new backend for the OCaml compiler which can compile an admissible subset of OCaml, including a majority of the most commonly used features. A C runtime library has also been written to support the compilation of the resulting C output, as well as a library of test programs which are used in testing and evaluation of the compiler. An evaluation has also been performed investigating the execution times of the compiler with respect to the OCaml native compiler, and the observability of the compiled code with the OCaml bytecode 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, Tianlin Zhang of Jesus 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: Tianlin Zhang

Date: March 30, 2018

Contents

Chapter 1

Introduction

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

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

1.1 Motivation

There are several motivations for this project, including:

Plausibility

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

Observability

The OCaml debugger was introduced fairly recently, and only works on compiled bytecode; in addition, since OCaml strips away type information on actual values at runtime, the debugger is unable to inspect the values of certain variables within polymorphic functions. Compiling to C could take advantage of a different data representation, GDB extensions to customise debugger behaviour, and allow my supervisor's library `liballocs`, which

can be used to track allocation sites and return type information of allocated blocks of memory at runtime.

Performance

By compilation to C, I can take advantage of the fact that the most popular C compilers have undergone decades of optimisation to produce very performant code. We can exploit this to obtain very fast executables without needing to perform much optimisation on the OCaml code. There will however be performance trade-offs in terms of emulating OCaml function calls in C compared to normal function calls, as the OCaml function system is far more complex than that of C.

1.2 Previous Work

There exist prior work for compiling ML-like languages into C [1] but the approach used is to transform the program into continuation-passing style, which is not suitable for the compiler as it would interfere with the program execution and make it untenable to map OCaml source code into regions of the compiled code for debugging purposes, but shows that compilation into C from similar functional languages is feasible.

In addition, this project was also attempted in previous years by a previous student [2] under the same supervisor with a similar starting point, but my approach to the design of the compiler was not influenced and it is of my opinion that a sufficiently different approach was used for this project.

Chapter 2

Preparation

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

In this respect, a large portion of time during the starting weeks of the project was focused on reading through barely-commented source code,

2.1 Starting point

The starting point of this project is the existing OCaml compiler [3], as it would be pointless to duplicate the effort of parsing and performing type inference on OCaml source code. Instead, my compiler takes the `Lambda` intermediate representation from the OCaml compiler and compiles this into C. Since the OCaml compiler backends Asmgen and Bytegen, which produce the native binaries/bytecode executables, also only take the `Lambda` IR, this approach is akin to introducing a new compiler backend or different target for compilation.

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

2.1.1 Intermediate representations

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

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

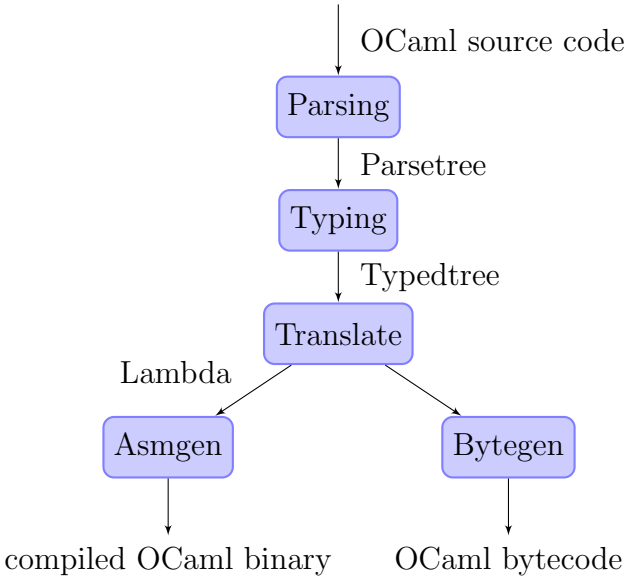

Figure 2.1: A representation of the various stages in the OCaml compiler. Adapted from [4, Chapter 22].

- It must be easy to recover type information from the representation. This is because the resulting C code must be statically typed also, so we need to be able to infer the types of variables and functions in order for them to be implemented in C.
- The representation must be fairly normalised, with as few as possible language constructs to simplify the translation into C. Many language features in OCaml are merely syntactically sugared versions of more primitive ones (for example, pattern matching is really a combination of branching based on the value of a variable plus some variable bindings for unpacking) which must have been desugared anyway during the compilation process, so it would be more useful if we could utilise an intermediate representation for which this has already happened.

`Parsetree`

`Parsetree` is a parsed AST representation of the source code, which essentially directly represents the original OCaml source code. No processing has occurred on any language constructs, and type inference has not even occurred on the IR. This obviously makes this intermediate representation unsuitable as the choice of IR, since it falls down on both ability to recover type information and normalisation.

`Typedtree`

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

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

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

2.1.2 The `Lambda` IR

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

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

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

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

- The translation pass into `Lambda` does not preserve type information from `Typedtree`. This is a huge problem which is solved somewhat with the existence of `Lambda` events, which will be discussed in section 3.1.
- The `Lambda` has not yet had closure conversion applied to it. Functions are still represented in a similar fashion to how they were in the original source code, meaning the cases of lexical closures, partial application, etc. are not handled.

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

2.2 Extensibility of GDB

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

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

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

2.3 Target subsets

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

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

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

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

2.3.1 Subset 1

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

The key choices made in this subset were with restricting the amount of possible functions and data types greatly. Function definitions could only be made at top-level, which means that there was no need to deal with lexical closures; in addition, they could only take one parameter so as to sidestep the issue with partial application. (This requirement was relaxed as it was found to be quite easy to simply restrict function applications to be full applications only.)

Otherwise, all data types and language constructs were chosen so that they more or less had direct analogues in C code (with the exception of the `unit` type, which was deemed necessary as it was required for imperative-style code in OCaml). This in theory made actual implementation details of the compiler simple and so the beginning of the project could more be focused on demonstrating a working environment and setup.

2.3.2 Subset 2

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

The implementation of an OCaml value representation is intended to be a rather large milestone, as it would greatly increase the expressiveness of the compilable language. Custom user-defined types would allow for implementations of custom data structures such as lists, binary trees, etc.

In addition, parametric polymorphism is a large step in terms of expressiveness, but its implementation would require some representation of polymorphic types and values, which can resolve to a specific type at runtime.

2.3.3 Subset 3

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

The implementation of this would also allow the representation of higher-order functions, such as standard list processing functions `map` and `filter`, and its completion would indicate the implementation of all of the most commonly used features expected to be in a functional language.

2.4 The `liballocs` library

A library suggested to me by my supervisor (who happens to be the project originator and also the author of the library) is `liballocs` [5], which is a library that is able to track all allocations in memory and their associated types with no extra required effort. It exposes an interface that, when given an arbitrary pointer, is able to return information about the type of the value in the allocated memory. This was identified to be extremely useful for implementing observability features, as it could be used within polymorphic functions to determine the type of certain values whilst debugging at runtime, which would be an advantage over the OCaml bytecode debugger, which due to the untagged nature of OCaml values cannot determine the type of polymorphic values at runtime.

2.5 Licensing of external code

As my project only operates on open-source code, software licensing issues are not of a great concern to the development of this project. A discussion of the licenses in question is included below, however.

The principal body of code being used is the OCaml core system [3], which is released under LGPL v2.1. LGPL is a more permissive version of the GPL license, intended for use for libraries rather than full pieces of software. **TODO – license discussion**

TODO – Inclusion of the Boehm GC in project

Chapter 3

Implementation

In the implementation, several modules were written in OCaml for implementing different parts of the compiler. The most significant modules were:

- `Typecollect`, for obtaining types from the `Lambda` IR;
- `Ccode`, for representation of C compilation target;
- `Ccompile`, for the actual compilation process from the `Lambda` IR to C.

3.1 Obtaining types from the `Lambda` IR

A problem encountered early into the project was the fact that the chosen intermediate representation, `Lambda`, did not have any type information associated with it. As `Lambda` was intended as the last representation before compilation into actual machine code, all type information was erased from the underlying data.

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

Originally, the `Typecollect` module attempted to obtain types from the `Typedtree` representation of the code before the transformation into `Lambda`, but it was found that `Typedtree` expressions do not correspond exactly with `Lambda` expressions. Eventually, a solution was found using `Lambda` events.

3.1.1 `Lambda` Events

In order for the OCaml compiler to produce debuggable bytecode executables, some type information must be passed through the `Lambda` IR to the `Bytegen` module, as the debugger has to know what type certain variables are. It was found that by turning on the debug flag in the compiler, it would insert so-called `Lambda` events into the compiled bytecode.

```
type lambda =
  (* ... *)
  | Levent of lambda * lambda_event
  (* ... *)

type lambda_event = {
  lev_loc : Location.t;
  lev_kind : lambda_event_kind;
  lev_repr : int Pervasives.ref option;
  lev_env : Env.summary;
}

type lambda_event_kind =
  | Lev_before
  | Lev_after of Types.type_expr
  | Lev_function
  | Lev_pseudo
```

Figure 3.1: Definition of the `Lambda` event in the OCaml compiler source code. The `lambda_event` type stores the attached metadata associated with the event. Of particular note is the `lev_loc` field, which stores the current source file name, line number and column number corresponding to where this event was inserted, and the `lev_kind` field, which potentially stores information about the type of the current expression.

A `Lambda` event is simply a wrapper around another `Lambda` expression that carries some extra information. They are defined within the OCaml compiler as shown in figure 3.1.1.

Within the compiler, the purpose of a `Lambda` event is to mark where an interesting expression may be, in order so that the debugger may stop execution and inspect the state of the program just before the evaluation of the expression, or just after. In fact `ocamldebug` does not step between lines of code at all – it steps between `Lambda` events.

This gives `Lambda` events two principal useful properties, which are:

- `Lambda` events provide information about the current source code location of the expressions being currently compiled, which will be useful in 3.2.4;
- `Lambda` events provide type information about types of certain expressions.

It turns out this is a bit more finicky than one might expect. The compiler does not insert events that give types for every expression, nor is there a one-to-one correspondence between `Lambda` expressions and source-code expressions, since the `Lambda` IR may simplify certain expressions or insert new ones.

A more reliable approach was found, where it was found that every source-code variable was associated with at least one `Lambda` event describing its type; therefore an approach

was adopted where the types of variables were obtained, and the types of compound expressions were determined with some very basic type-inference.

3.1.2 The `Typecollect` module

Putting this all together, the `Typecollect` module has a function `scrape` which performs this sequence of instructions:

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

The compilation pipeline therefore runs this preliminary pass over the `Lambda` IR before compilation, and then passes the returned types to the `Ccompile` module to inform its compilation process.

3.2 Representation of C AST

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

3.2.1 Expressions to statements

Another of the problems associated with the translation of OCaml into C was the fact that the two language had a large difference in syntax. OCaml is what is known as an expression-oriented programming language, where every syntactical construct is actually an expression of some kind. This is in contrast to C, which is statement-oriented – while expressions exist in C, a block of code in C is instead a list of statements, certain constructs can only be written as statements (e.g. `return`, variable declarations), and other constructs such as if/while/for statements etc. cannot be used as expressions.

The scheme that my compiler uses to solve this problem is a simple one: every sub-expression in the program is assigned to a separate variable. This means that if an expression consists of just a variable, use the variable name directly; otherwise, create and assign the result of each sub-expression to a new temporary variable.

This scheme simplifies the compilation strategy greatly. By setting up a variable for each sub-expression, this creates an equivalence between variables and sub-expressions. This means that the compilation function need only recursively traverse down the `Lambda` IR tree, adding statements to the current context as needed and returning the variable representing the expression it was asked to compile. Thus, each sub-expression compiles

to a block of C statements, at the end of which there is an assignment to a variable that holds the value of the expression.

This scheme obviously produces a very large quantity of temporary variables, but C compilers have gone through decades of optimisation and often produce lots of temporary variables anyway in code transformations such as SSA, so are perfectly fine with compiling and optimising code containing large amounts of temporary variables. In fact, through my compiler OCaml code without references compiles (almost!) into SSA form.

3.2.2 The `Ccode` module

The `Ccode` module is an auxiliary module of the compiler that simply stores the type declarations for the C AST. This module is not sufficient to represent the entirety of the C language, but instead is chosen to represent the specific subset of C to which the compiler will target. There are four main datatypes in the module, and a quick summary of them is given here:

1. `Ccode.cident`, for representing identifiers (variable names, macro names, function names etc.) in C. This is largely a wrapper over the `Ident.t` type the OCaml compiler uses to represent identifiers, which is the string name along with a unique integer ID to disambiguate it from other identifiers with the same name.

2. `Ccode.cexpr`, for representing expressions in C. These include a wrapper around `cident`s, literals, and operations (such as unary operations, binary operations, function calls etc.) on other `cexpr`s.

3. `Ccode.cstatement`, for representing statements in C. Most data constructors in this type take a `cexpr` or a block of other `cstatement`s, and include if statements, while statements, switch statements etc. Notably a `cexpr` can be promoted to a `cstatement`, but not the other way around.

4. `Ccode.ctype`, for representing types in C. This is a auxiliary type used by the casting operator in `cexpr` and the variable declaration statement in `cstatement`. A more in-depth discussion of the types used to represent OCaml values is at section 3.4.

3.2.3 The `Cprint` module

The `Cprint` module is another module that's fairly straight-forward – it contains functions for printing types from `Ccode` to an output stream, most likely a file. There's not anything very complicated going on in this module – the functions simply recursively traverse down the C "AST" and prints out the corresponding C as it goes along.

3.2.4 `#line` directives

One feature to do with observability which would be prudent to discuss here is use of the so-called `#line` directive. A key observability feature is for the debug table to associate

sections of the output code with the relevant line from which they were compiled from, which allows debuggers such as GDB to display to the user a listing of the relevant source code, and associate the currently executing machine code with the correct line of source code.

Naturally this ability is highly desirable for this compiler. Luckily, the C preprocessor supports controlling the line and source file of your code with use of the `#line` directive, by inserting it in the source code in the format `#line linenum filename`. This provides a compilation strategy for the compiler, which to compile all statements as being on the same line (the line number changes whenever there is a new line, even after being specified with `#line`), interspersed with a `#line` directive whenever the source code line actually changes.

One final question is, where do we obtain information about which line a `Lambda` expression is from? `Lambda` expressions typically do not map to source-code expressions, nor do they contain information about which line the expression originates from. This is the second place where `Lambda` events are useful (refer back to 3.1.1) – they carry information about the current file and line. Also, since they represent places where the OCaml bytecode debugger `ocamldebug` may pause execution, this gives us a pleasing way to treat them: `Lambda` events compile into `#line` directives.

3.3 Compilation of basic constructs

After the initial foundations of the compiler were made, structures from Subset 1 in the proposal were implemented first. These encompass simple structures which have similar analogues to structures in C.

3.3.1 Variable scoping differences between C and OCaml

Before diving into the implementation of these expressions, one issue that needs to be addressed is the differences in variable scoping between the two languages. This is a significant detail when considering observability – we would like the debugger to print out the contents of the correct variable within a scope when prompted.

In OCaml, each variable is only visible in the body of the expression where it is bound, whether if that's in a let-binding or a function declaration. This is in contrast to variable scope in C, where a variable is visible for the entirety of the rest of the block it is in.

Furthermore, while OCaml does not allow reassignments (of variables, not references), it does allow you to bind another variable with the same name, which you can see in the short snippet:

```
let x = 1 in
let x = 2 in
print_int x
```

Here the second binding is not reassigning the value of `x`, rather it's creating another variable with the same name. The first variable still exists, but it's been shadowed by the

second variable making it inaccessible. Note that after leaving the body of the second let binding the second variable goes out of scope and the first variable is accessible again.

This behaviour can be approximated using block-scoping in C, since variable scopes are limited to the block they were created in C and local variables shadow variables in an outer scope. It's worth noting that however, this does not work for file-level variables, since you cannot create blocks of code at the file-level – all file-level variables must live in the same scope.

3.3.2 `let` bindings

With the differences in variable scoping between C and OCaml appreciated, particular care has to be taken when compiling a `let` binding. In OCaml, a `let` binding has the structure:

`let` *x* `=` *expr* `in` *body*

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

The resulting structure that a `let` binding compiles into is shown therefore by this C-style pseudocode. Recall that according to the compilation strategy outlined in section 3.2.1, each OCaml expression compiles to a block of C statements, at the end of which the value of the expression is assigned to a variable. We will therefore adopt the notation where for an OCaml expression `X`, `comp[X]` denotes the block of C statements `X` compiles into, and `var[X]` denotes the C variable that the value of `X` is assigned to.

```
decl result;

comp[expr];
{
    decl x;
    x = var[expr];

    comp[body];
    result = var[body];
}
```

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

The other detail is that OCaml allows `let` bindings to be a shorthand for function declaration; function compilation will be discussed in section 3.5.

3.3.3 `if-then-else` expressions

OCaml does not have if-statements but if-expressions, which are of the form:

`if` *cond* `then` *A* `else` *B*

Using the same notational conventions as the previous section, this compiles into the pseudocode:

```
decl result;

comp[cond];
if (var[cond]) {
    comp[A];
    result = var[A];
}
else {
    comp[B];
    result = var[B];
}
```

The same trick to propagate results outwards from inner scopes has been employed here also.

3.3.4 `while` loops

While loops are fairly simple in OCaml – they simply repeatedly evaluate their body while the condition evaluates to true. There are no break nor continue statements, so there is not a concept of breaking out of a loop early with the exception of setting the condition to false. They have the syntax:

`while` *cond* `do` *body* `done`

In addition, the overall return value of the loop is `unit`. This therefore translates into the following pseudocode:

```
comp[cond];
while (var[cond]) {
    comp[body];
    comp[cond];
}

decl result = make_unit();
```

There is one interesting thing of note here: *cond* does need to be evaluated twice, once before the loop, and once at the end of the loop. This is because the OCaml while loop re-evaluates *cond* at the beginning of each iteration, but since OCaml expressions turn into a list of statements in C, we cannot fit this re-evaluation into the head of the while

statement – instead, a simple and equivalent way around this is to simply re-evaluate *cond* at the end of the loop.

A discussion of the way the `unit` type is implemented will be in section 3.4.

3.3.5 `for` loops

For loops in OCaml are extremely limited. They permit only iteration over a fixed range of integers, and also only allow increments in steps of 1. Like with while loops, there are no break nor continue statements in OCaml and so there is no possibility of exiting a loop early. They have two forms, which are:

`for` *x* `=` *start* `to` *end* `do` *body* `done`
`for` *x* `=` *start* `downto` *end* `do` *body* `done`

These two versions simply iterate up to or down to a certain number (inclusive). Since for loops are so incredibly limited, it was decided that rather than including for loops in the targeted subset of C, it would be easier to also compile these into while loops in C.

```
comp[start];
comp[end];

{
    decl x = var[start];
    while (x <= var[end]) {
        comp[body];
        x++;
    }
}

decl result = make_unit();
```

(Replace `<=` for `>=` and `x++` for `x--` for `downto`.)

There are a few things about this compilation that warrant elaboration:

- *x* again needs to be declared in its own scope, as the for loop acts as another variable binding site. Thus, the entirety of the for loop is wrapped in another block.
- *start* and *end* are only evaluated once at the start of the loop – after some quick experiments it could be shown that OCaml does this as well, i.e. the limits of the iteration range do not change while the loop is running.
- *x* is mutated through the iteration of the loop. This is fine, as OCaml for loops only operate on integers, which are copied instead of referenced by other constructs in the target subset of C; thus the mutation of *x* cannot affect the behaviour of the program.

3.3.6 Recursive bindings with `let rec`

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

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

```
let rec x = 1::y and y = 1::x in expr
```

This binds `x` to the infinite list `1::2::1::2::...`, which is accomplished by making the list cyclical – that is, it points back it itself.

Informally, the class of values that are allowed to be used as the right-hand side of a `let rec` binding are those where the recursively bound names occur only within a function, or a data constructor. This means that to preserve the semantics in C, the variables should be first initialised with `malloc` to determine their pointers. Once the pointers are determined, compilation proceeds as normal using the newly-determined pointers when referring to the variable.

The compiler thus splits compilation of `let rec` into two phases:

- Firstly, variables are declared. If the associated expression has no free variables, compilation can happen straight away; otherwise, the size of the resulting value in memory is determined and a pointer is obtained using `malloc`.
- Once all the pointers are known, compilation proceeds as normal with assignments going to the declared variable name instead of a locally created variable.

3.4 OCaml value representation

An important aspect of the compiler is the design of the representation of OCaml values within C. OCaml has a rather different data-type model to C, so an appropriate design of value representations is not only crucial to the correctness and speed of the compiler, but also to the observability of the resulting code.

An example of this is that the OCaml compiler discards type information in its compilation process, so that the type information at runtime cannot always be determined. This is detrimental to the observability of its debugger, as `ocamldebug` for example cannot determine the actual types of polymorphic values at runtime and so cannot display their values, opting only to display `<poly>`.

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

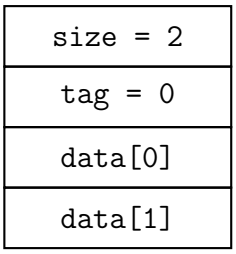

Figure 3.2: Example OCaml block containing two values with tag 0, with a header containing the current size and tag of the block, followed by the values of the block.

3.4.1 Representation requirements

Before discussing the strategy for representing OCaml values, it is useful to explore the exact problems that the value representation are aiming to solve, which will in turn motivate its design.

Basic types

OCaml has a number of primitive types, which have analogues in C. These can be translated directly in most cases. These include `int`, `float`, `bool`, `string`, and `char`.

Integer types

Simple values and variants are represented as integers within OCaml. For example, the `unit` type is represented with the OCaml integer 0, and variants with no parameters are represented as integers, e.g. in `type test = Foo | Bar`, `Foo` is represented by the integer 0 and `Bar` by the integer 1.

OCaml blocks

More complex types such as variants with parameters, tuples, and records are represented by a construct known as an OCaml block, which is a variable length array of values with a header containing its length and an integer tag, which is used to differentiate between different variants and types of blocks. An example is `type test = A of int | B of float | C of str` – `A` will construct a block containing an `int` with tag 0, `B` will construct a block containing a `float` with tag 1, and `C` will construct a block containing a `str` with tag 2.

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

Polymorphic types

Code in polymorphic functions may not know the type of variables they're manipulating, so cannot deal with variably-sized data as they may not copy or pass polymorphic values correctly.

3.4.2 Representation strategy

As a basic consequence of the fact that we want the resulting C code to be observable, whenever there is a basic type the C code should endeavour to use the equivalent type in C as much as possible. Thus for simple cases the translation and representation is straightforward – simply replace the OCaml type with the corresponding C type.

Representation for non-primitive types however is more complex. Integer types and OCaml blocks must be unified somewhat in representation, because variants that don't have any parameters and variants that do can be of the same type; for example, in `type test = Foo | Bar of string`, `Foo` is represented by the integer 0, but `Bar` is represented by an OCaml block with tag 0 and the string as its first element, but these need to be represented as the same type despite one being variably-sized.

Tagged pointers

The solution to this therefore is to employ a tagged pointer structure. Firstly, OCaml blocks must be allocated on the heap since they are variably-sized, so we must reference them using a pointer. This means that we need a union type which combines a pointer and an integer, which is what a tagged pointer representation does.

Tagged pointers take advantage of the fact that on most architectures, pointers are word-aligned. This means that the lower bits of a pointer are always 0, which means that it's possible to store extra bits of information in the lower bits of a pointer, and just zeroing those bits out when you need to dereference the pointer. In the architecture I am developing on, which is 64-bit Linux, pointers are always aligned to 8-bytes – this leaves 3 free bits at the end of any pointer.

The tagged pointer approach employed in this compiler is to use the least significant bit of an 8-byte word as the tag bit to differentiate between pointers and integers. This does mean that integers will lose one bit of precision, but in fact the OCaml native runtime uses either 31-bit or 63-bit integers as well depending on the architecture.

There is a choice as to which way round the tag bit should be – should 0 represent integers or pointers? It was chosen to use 0 for pointers, and 1 for integers, which means that the "boxed" version of a pointer is identical to the pointer in memory. This choice was made on two bases:

- Dereferencing pointers is far more common than integer maths (which also happen to representing variants, not actual integers), so no extra work needs to be performed to dereference a boxed pointer.
- Since boxed pointers are exactly the same as they were prior to being boxed, this means that it is incredibly easy to use a conservative garbage collector as a drop-in replacement. This is discussed further in section 3.6.

It is noted here that an alternative approach to boxing pointers exists and is common in implementations of other languages, in particular JavaScript, which is NaN-boxing. The concept is to utilise the unused "NaN-space" of values in IEEE 754 floating point numbers to store integers and pointers, meaning that this approach is able to store floating

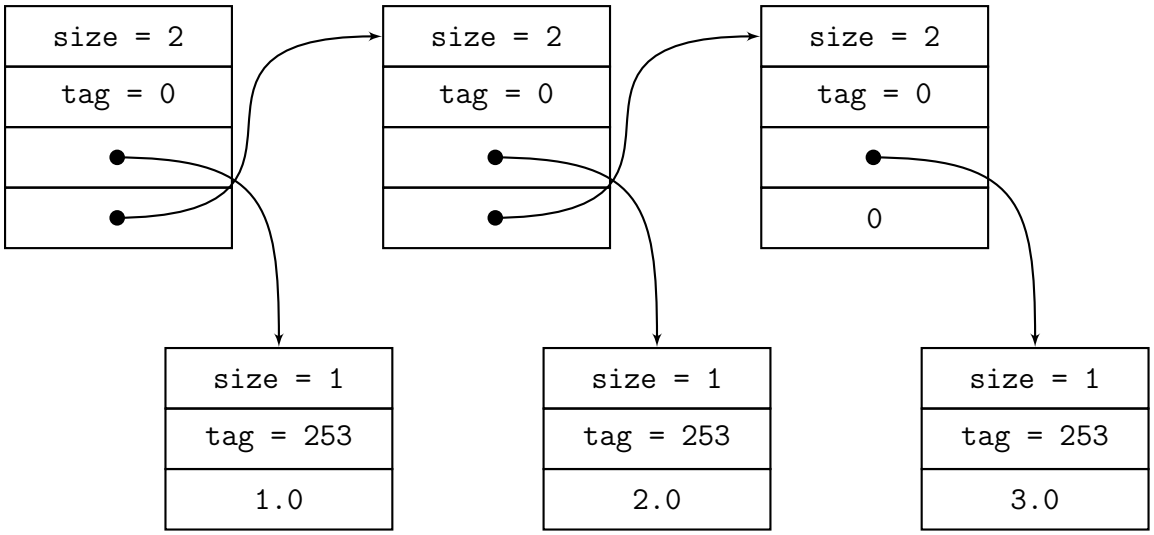

Figure 3.3: Example box-and-pointer diagram for the list `[1.0; 2.0; 3.0]`. As lists are linked lists in OCaml, each node in the list contains the OCaml value of its element, and the next value in the list, which is either a pointer to the next node or the OCaml value `[]`, represented by the integer 0. Floating point numbers must be boxed to be stored in a block, so a new block is created for each one. These blocks have the special tag 253, which indicates that their contents is a floating point number.

point numbers as immediates as well as integers. This approach was not chosen because of the relative complexity of implementation, extra reduced precision for integers, and poor interopability with existing conservative GC implementations.

Polymorphic values

Now that we have one consistent representation for integers and arbitrary blocks, which always takes up one word, it's very tempting to use this to represent polymorphic values as well. If all values can be represented using a word-sized structure, this solves the problem of not knowing what size a polymorphic value is.

This is because the only operation that could be performed on a polymorphic value is copying it somewhere or passing it as a parameter to another function – if the type was more concretely known at any point, it would've been possible to obtain a more concrete type from the `Lambda` IR.

The solution is therefore to "box" all types as the tagged pointer representation. Since the pointer part of the tagged pointer representation always points to an OCaml block, we create special OCaml blocks for holding other primitive types such as floating point numbers and strings. To disambiguate these blocks from other normal blocks so that at runtime, the debugger is able to see what type they really are, we assign a special tag to them which marks the block as containing only floats or strings or etc.

3.4.3 Representation in C

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

```
typedef struct _block {
    uintptr_t size : 56;
    uintptr_t tag : 8;
    any_type data[];
} block;
```

Recall that a block needs to store its size, an integer tag, and any number of other OCaml values. On a standard 64-bit architecture the address space is only 2^{48} bytes large, and considering that we only need to store words which are 8 bytes on 64-bit Linux the maximum that `size` could be is 2^{45}. In addition, OCaml tags only range between 0 to 255[1]. We can therefore reduce the amount of space used for the header using C bit-fields, allocating 8 bits to `tag` and the remaining 56 bits to `size`, meaning the header is only one word long.

Now we have a block definition, we can implement actual OCaml values as being a simple union type:

```
typedef union _value_type {
    intptr_t i;
    block* block;
} value_type;

#define BOX_INT(v) ((value_type){.i = (intptr_t)(v) << 1 | 1})
#define UNBOX_INT(v) ((v).i >> 1)

#define BOX_BLOCK ((value_type){.block = (v)})
#define UNBOX_BLOCK ((v).block)

#define IS_INT(v) ((v).i & 1)
```

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

One final thing to talk about is the purpose of `any_type`, which one may have noticed in the definition of `block`. Blocks normally hold an array of `value_type`s, but there are special blocks which hold strings and floating point numbers. This means that blocks actually need to hold a union type, which is defined thusly:[2]

[1] In fact, the native OCaml runtime restricts the number of possible variants to 246, using the tags 0-245 for variants. It uses the last 10 possible tags for special tags, such as tagging floats, strings, abstract values etc.

[2] For clarity, we're ignoring the problems associated with recursive type definitions in C, and assuming that we have the correct forward declarations.

```
typedef union {
    intptr_t i;
    uintptr_t u;
    double fl;
    char* str;
    value_type value;
    closure_type* closure;
    (void*)(*f)();
} any_type;
```

We use a union for this kind of type-punning instead of doing a pointer cast with a `void` pointer for example to avoid violating the strict aliasing rule, which allows the output of our compiler to be optimised more greatly by C compilers. Another detail is that the `any_type` union also holds integers, function pointers, and closures as it is reused for the closure representation, which will be discussed in section 3.5.

3.4.4 Type casting

While the `Lambda` IR has already gone through type inferencing and thus types should have been already determined and statically checked, there are still a few cases in which a cast is necessary from one type to another. These cases are:

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

A very simple algorithm is used therefore whenever there is a type mismatch between two variables in the cases of assignment and passing arguments to functions or closures.

1. If the source and target types are the same, do nothing.
2. If the source type is `any_type`, unpack the union according to the target type.
3. If the target type is `any_type`, pack the value into another `any_type`.
4. If the source type is `value_type`, unbox the value according to the target type.
5. If the target type is `value_type`, box the value into anotehr `value_type`.
6. Otherwise, perform a C cast between source type and target type.

3.4.5 Pattern matching

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

`Lswitch` expressions

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

```
type color =
  | Red
  | Green
  | Blue
  | Gray of int
  | RGB of int * int * int

match c with
  | Red -> 0
  | Green -> 1
  | Blue -> 2
  | Gray x -> 3
  | RGB (r, g, b) -> 4
```

In the type `color`, `Red`, `Green` and `Blue` will be compiled into the OCaml integers 0, 1, and 2, and `Gray` and `RGB` are compiled into blocks with the tags 0 and 1 respectively. This match statement is compiled into the following `Lambda` expression:

```
(switch* c/xxxx
  case int 0: 0
  case int 1: 1
  case int 2: 2
  case tag 0: 3
  case tag 1: 4)
```

This looks fairly similar to a switch statement in C, but there's one important difference: each the cases predicate on both whether the value is an integer or a block and the value of its value/tag.

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

`Lstaticraise` and `Lstaticcatch`

The `Lambda` IR actually has support for static exceptions, which are exceptions that can only occur across a local scope (in contrast to normal exceptions, which unwind the stack and can jump to an error handler in an enclosing scope). In particular, the OCaml pattern-match compiler may opt to use a static exception to avoid duplication of code for certain branches. For example, consider this match expression:

```
match c with
  | Red -> 0
  | Green -> 1
  | Gray x -> 2
  | _ -> 3
```

This match expression contains a wildcard expression which matches both `Blue`, which is an integer, and `RGB`, which is a block. The OCaml pattern-match compiler thus compiles it into the following `Lambda` expression:

```
(catch
  (switch* c/xxxx
    case int 0: 0
    case int 1: 1
    case int 2: (exit 1)
    case tag 0: 2
    case tag 1: (exit 1))
  with (1) 3)
```

The branches for `Blue` and `RGB` have been compiled into a static exception, to avoid duplicating the expression in the branch. Luckily in C this can be very simply emulated with gotos. `Lstaticcatch` expressions can be compiled simply into an `if(0)` and a label, and `Lstaticraise` expressions are simply compiled into a goto.

3.5 Function compilation and closure conversion

Functions and closures are the last hurdle to overcome for compiling OCaml into C. As OCaml is a functional language, functions have many more features than in C, including creating closures, freezing the environment of a function into a value.

OCaml has support for first class functions, allowing functions to be used as values. C does not have support for this – it supports a limited form of this using function pointers, but this is not sufficient for the representation of closures in C. The following sections will discuss the motivations for a closure representation and the implementation of such a representation.

3.5.1 Local functions

All functions in C are required to be defined at top-level, which means that nested function declarations are not possible. This is in contrast to OCaml, where function definitions are

simply expressions like any other, and thus can be defined locally and passed as values. In order to emulate this behaviour in C, whenever we come across a function definition we lift the function definition to the toplevel scope, compile the function there, and then return its function pointer when we come back.[3]

3.5.2 Function Typing

In OCaml, one way of thinking about functions is to view them as only ever being functions of one argument, which return other functions. As an example, the function type `int -> float -> int` represents a function that takes an integer and returns a `float -> int`, which is a function that takes a floating point number and returns an integer. It can be seen that `->` is really a right-associative operator on types; a function type `'a -> 'b -> 'c -> 'd` should be bracketed as `'a -> ('b -> ('c -> 'd))`.

While this is a correct high-level view of functions and explains how partial application works nicely, this is not suitable for a low-level implementation at all – it is much more efficient for functions to actually accept multiple arguments, instead of returning a succession of other functions for which each need to be successively constructed and have the just-applied value saved to the environment of the function. In addition, successively calling functions increases the amount of stack manipulation that takes place, and not passing multiple arguments at once means that the compiler is not able to take advantage of optimisations where arguments are passed by registers instead of stack-spilling.

In C by contrast, a function declaration consists of a return type, and a list of input arguments. We therefore need a consistent conversion between OCaml function types and C function types, so we pick the most obvious one – all the types in the OCaml function type are the arguments to the C function, and the final type is the return type of the C function. Thus, a function of type `int -> float -> str` is compiled to `char* (*)(int, double)` in C.

Functions that return other functions

While functions in OCaml are usually declared as if they take in multiple arguments and return a value, it is possible and often useful to define functions that return other functions. As a simple example, consider the function:

```
let f x = (fun y -> x + y)
```

The type of `f` is `int -> int -> int`, but it's obvious that this isn't translatable into a C function `int f(int x, int y)` – the body of the function would return a function pointer, and not an integer.

This is actually a rather simple problem to fix via a technique known as eta-expansion, and can be done at the source code level. Simply add extra parameters to match the number of parameters in the type, and apply them to the old result of the function.

```
let f x y = (fun y -> x + y) y
```

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

3.5.3 Closure representation

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

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

```
typedef struct _closure_type {
    void* (*f)();
    struct _closure_type* next;
    any_type args[];
}
```

As you can see, this definition contains a function pointer, and a variably-sized array of arguments, which we reused `any_type` from section 3.4.3 to implement. One interesting choice taken here is that closures also have a `next` field which points to another `closure_type`, allowing them to act like a linked list – the motivation of this is discussed in partial application compilation, in section 3.5.4.

Promoting functions to closures

Now that we have a closure object, we need some way for functions to actually access the contents of the closures. The way this is implemented is to add an extra parameter to the end of the arguments of any closure, `closure_type* closure_obj`, which stores the pointer to the current closure object. From there, the function is able to access closure values by indexing `closure_obj->args`.

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

```
return_type promoted_closure(...args, closure_type* closure_obj) {
    return f(...args);
}

closure_type* cl = MALLOC(sizeof(closure_type));
cl->f = (void*(*)())&promoted_closure;
cl->next = NULL;
```

Calling closures

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

```
cl->f(...args, cl);
```

As one can see from the previous section, `cl->f`, which is `&promoted_closure`, correctly translates this call into a call of `f` with the correct arguments.

Unification of closures and functions

Closures and functions are indistinguishable and practically the same thing in OCaml, so for ease of representation, it was decided to unify functions and closures in C as well. This means that all functions are promoted to closures, and all function applications in OCaml are compiled into closure calls in C.

While this simplifies the implementation, it does come at a large performance cost of needing to allocate extra memory whenever new closures are created, as well as causing function calls to go through an extra layer of indirection. Surprisingly however, experiments have shown that modern C compilers are able to successfully detect references to other functions in this representation, and even perform tail-call optimisation in spite of this.

3.5.4 Partial application compilation

Partial application is one of the major use-cases for closures, whereby programmers can create new functions by not passing all required parameters to a function. Operationally, this means that the value of the parameter is saved in a closure, and once enough arguments have been applied to the closure the parameter is passed as the first argument along with the other arguments to the original function.

This is modelled using a closure object. Whenever there is a partial application, a function is created instead that takes remaining parameters, and a closure object is created to store the partially-applied arguments. The newly created function simply obtains the previously partially applied arguments from the closure object and passes them along with its own arguments to reconstruct the original function call. As an example, observe the following C-style pseudocode, where a closure `cl_f` (which requires three integer arguments and returns an integer) is partially applied with two integers, `a` and `b`, and another closure `cl_g` is returned[4].

```
int g(int c, closure_type* closure_obj) {
    int arg_0 = closure_obj->data[0];
    int arg_1 = closure_obj->data[1];

    closure_type* cl = closure_obj->next;
    cl->f(arg_0, arg_1, c, cl);
}

closure_type* cl_g = MALLOC(
    sizeof(closure_type) + sizeof(any_type)*2);
cl_g->f = &g;
cl_g->next = cl_f;
cl_g->data[0] = a;
cl_g->data[1] = b;
```

[4]Cast operators have been omitted for simplicity.

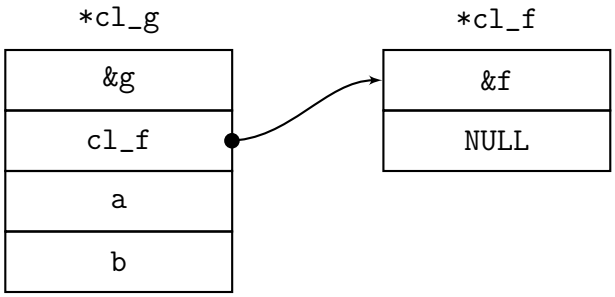

Figure 3.4: Example result of a partial application of a function `f(a, b, c)` with the arguments `(a, b)`. The arguments `a` and `b` are pushed onto a new closure object, which contains a function pointer to `g` and a pointer to the previous closure object. When `cl_g` is called with an argument `c`,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

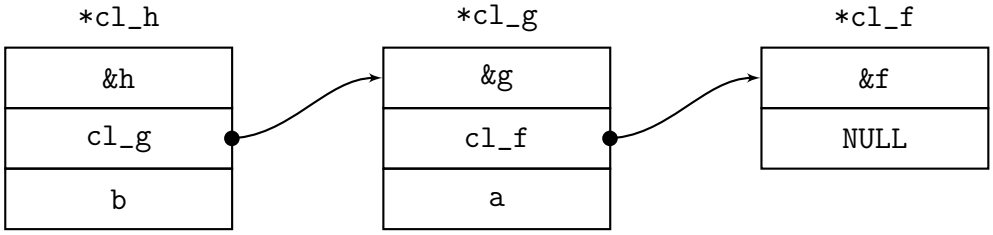

Figure 3.5: Example of a chained double application. In this case, the function `f(a, b, c)` has been applied with `a` and `b` separately, creating two closure objects, `cl_g` in the first application, and `cl_h` in the second. Here, `h` translates a call `h(c)` into `g(b, c)`, which in turn translates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

Chaining partial applications

An important aspect of this code transformation is that the resulting closure operates exactly like a promoted function closure, and is invoked in the same way. This means that it is also possible for chain partial applications – i.e. to perform partial application on a closure resulting from a partial application.

This requirement is also the justification for why closures are implemented as a linked list – each closure needs to know the function pointer of the next function they need to invoke, but that isn't always possible to determine statically[5]. Thus, each closure resulting from partial application points to the closure from which it was derived from so their function can figure out what function pointer to invoke.

3.5.5 Lexical closure compilation

Lexical closures are another common use of closures, and occur when a function refers to values in an enclosing scope. This presents a problem in C, as it does not allow nesting functions, which means that these values will have to be passed to the function somehow.

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

Common solutions to lexical closures are the techniques of lambda lifting or static links, but these do not work when functions are treated as first-class values. This is because the locally created function may be passed out of the scope where it has been created, so a closure object must be created to hold a copy of all the lexically bound variables along with the function pointer.

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

This process produces a closure object, which again, behaves the same and is invoked in the same way as a closure created from promoting a function or partial application.

As an example, consider the compilation of the following OCaml code:[6]

```
let f x y =
    let g z = x + y + z in g
```

This is compiled into the following C-style pseudocode:[7]

```
int f(int x, int y, closure_type* closure_obj) {
    closure_type* cl_g = MALLOC(
        sizeof(closure_type) + sizeof(any_type)*2);
    cl_g->f = &g;
    cl_g->next = NULL;
    cl_g->data[0] = x;
    cl_g->data[1] = y;

    return cl_g;
}

int g(int z, closure_type* closure_obj) {
    int x = closure_obj->data[0];
    int y = closure_obj->data[1];

    return x + y + z;
}
```

3.5.6 Function polymorphism

Function-level polymorphism is an important feature in OCaml, allowing programmers to write general code without knowing the type of its arguments in advance. As stated before, C has no concept of polymorphism, so instead we use the `value_type` representation of polymorphic values when defining polymorphic functions.

[6] In an actual compilation, this will get eta-expanded to make the function types consistent (see section 3.5.2) but we suppose this doesn't happen in this example for simplicity.

[7] We're also omitting the closure promotion process here.

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

3.5.7 Casting polymorphic functions

Passing other functions (closures) into polymorphic functions is more problematic. Suppose one of the arguments of a function is `'a -> 'b` – how do you pass another function to it? There are two things to consider here.

Erasing function types

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

```
value_type g(value_type x, closure_type* closure_obj) {
    double new_x = UNBOX_FLOAT(x);

    closure_type* cl = closure_obj->next;
    double result = cl->f(new_x, cl);

    value_type new_result = BOX_FLOAT(result);
    return new_result;
}

closure_type* cl_g = MALLOC(sizeof(closure_type));
cl_g->f = &g;
cl_g->next = cl_f;
```

This in fact needs to happen any time it's possible to "lose track" of the actual type of a function, i.e. it's possible the function will be used in a situation where its type cannot be statically determined. This includes adding functions to an OCaml block (because the data structure may be passed to a function polymorphic in a subtype of the data structure), or accessing functions from an OCaml block[8], or storing and retrieving functions from closure data.

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

1. If the source type is `value_type` or `any_type`, when casting back assume all of its argument and return types are `value_type`, and then cast it back to the expected type using the technique from the previous section.

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

2. If the target type is `value_type` or `any_type` and the source type is a function, first erase types from that function by rewriting all its argument and return types to `value_type` using the technique from the previous section before proceeding with the cast.

"Downcasting" functions

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

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

The problem therefore has to be solved on the caller's side. In essence, we want to transform a C function signature from:[9]

```
value_type f(value_type x, value_type y);
```

into:[10]

```
closure_type* g(value_type x);
// where the return value contains a function
value_type h(value_type y);
```

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

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

`g`'s role is to act as the resulting function, that will be passed into whatever polymorphic function. Its job is to set up a closure for `h` based on its arguments – its essentially a wrapper around a partial application.

```
closure_type* g(value_type x, closure_type* closure_obj) {
    closure_type* cl = MALLOC(
        sizeof(closure_type) + sizeof(any_type));
    cl->f = &h;
    cl->next = closure_obj;
    cl->data[0] = x;

    return cl;
}
```

[9] We're omitting the `closure_type* closure_obj` parameters which should go on the end.

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

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

```
value_type h(value_type y, closure_type* closure_obj) {
    value_type x = closure_obj->data[0];

    closure_type* cl = closure_obj->next;
    value_type result = cl->f(x, y, cl);

    return result;
}
```

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

```
closure_type* cl_g = MALLOC(sizeof(closure_type));
cl_g->f = &g;
cl_g->next = cl_f;
```

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

Polymorphic values being able to take the place of part of the type of a function cause more trouble later, in section 4.1.1.

3.6 Garbage Collection

During the execution of an OCaml program many objects will be allocated on the heap, and often – an allocation happens whenever a new data structure is made, whether that is a variant, a record, or even a tuple, and whenever a new closure is made. It is incredibly difficult to track the lifetimes of these objects, or in any function language for that matter, so for that reason the OCaml language employs a garbage collector for cleaning up allocations that are no longer used.

C unfortunately does not have a garbage collector, but there exist implementations of GCs for C, the most popular of which is the Boehm garbage collector. This is a conservative GC, which means that it treats anything that potentially looks like a pointer to a valid allocation as a pointer. This occasionally leads to false positives, where the GC does not free some memory which should be inaccessible but by chance there is a value that looks like it is a pointer pointing there.

The good news is that because of the way we have defined the implementation of OCaml values, all values potentially representing pointers will be unchanged in their actual binary value. This means that adding a garbage collector to our executables is as simple as using the Boehm GC as a drop-in replacement of `malloc`, only requiring:

```
#include "gc.h"
#define MALLOC(v) GC_MALLOC(v)
```

TODO: move some of this to the preparation?

Chapter 4

Evaluation

4.1 Regression tests and feature completeness

4.1.1 Polymorphism and function application

4.2 Benchmarks

4.2.1 Instrumentation

4.2.2 Allocation time

Removing extra allocations

Polymorphic comparisons

4.2.3 Closure creation

4.2.4 Garbage collection

4.3 Observability

4.3.1 Sample GDB sessions

4.3.2 Line number matching

4.3.3 Variable printouts

Chapter 5

Conclusion

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] C. Sun, *An Observable OCaml with liballocs and C*, May 2017.

[3] INRIA, *The OCaml Core System*, fetched at 2017-09-27.
`https://github.com/ocaml/ocaml/tree/4.05`

[4] A. Madhavapeddy, J. Hickey, Y. Minsky, *Real World OCaml*, O'Reilly Media, November 2013.

[5] S. Kell, *Towards a Dynamic Object Model within Unix Processes*, October 2015.

Appendix A

Project Proposal

Introduction and Description of the Work

This project involves the implementation of a compiler from OCaml to C in such a way that will allow the resulting executable to be compatible with DWARF-based debugging tools, such as `gdb`. This project will use the existing front-end of the OCaml compiler, but implement a very different back-end which outputs valid C code that can be compiled using a tool such as `gcc` into a debuggable executable. The focus of the compiler is to optimise for debuggability, or "observability", by taking advantage of the numerous debugging tools available of the C compilation toolchain.

Since the implementation of the full OCaml language is likely too ambitious for one project, the core project will instead revolve around the implementation of an interesting subset of OCaml which still provides technically interesting challenges in its implementation, while extension tasks may extend the implementation to cover a wider subset of the entire language. An informal description of said subset will be given in the Substance and Structure section.

The "observability" of the compiler means that where possible the debugger should behave *as if* it was debugging the OCaml code. This means that the debugger should consider sections of the compiled machine code to be mapped to the appropriate lines of code in OCaml. Furthermore, locally bound variables should be visible from the debugger when execution has reached the relevant point, meaning that local variables in the C code should mirror the variables in the OCaml code.

In addition, features such as closures and parametric polymorphism must be implemented carefully since these features don't have any similar structures in C for which they could've been mapped to, but also to maintain "observability" these must be "transparent" to the debugger (for example, a polymorphic function of the type `'a -> 'a` that has been instantiated as the type `int -> int` should have this quality be visible from the debugger). These qualities can be implemented using user-defined commands for `gdb`.

The resulting compiler can be evaluated in mainly two contexts. One of these is a straightforward performance comparison, where a testbed of OCaml programs are benchmarked across the OCaml native compiler, the OCaml bytecode compiler, and this project. The other context is the "observability" of the compiler; where programs are stopped at a certain point of their execution and inspected, to see how much of the in-

ternal state of the stack is recoverable. This can be made in comparison with OCaml's bytecode debugger, `ocamldebug`.

Resources Required

No special resouces are requred that are not open source and freely available for download.

Starting Point

The project, at least initially, will be a fork of the OCaml compiler obtainable at `github.com/ocaml/ocaml` which will take the same front-end as the existing compiler, but will implement a new back-end for compilation into C. The work may use other libraries such as `libffi` and `liballocs` but no work of similar scope to the project.

Substance and Structure of the Project

The core of the project will be devoted into the implementation of three incrementally expanding subsets of OCaml. These subsets have been identified such that each is sufficient to write nontrivial programs in, and features have been grouped by relation to each other. In this manner work can be easily split into 3 discrete blocks, each of which will support a new collection of programs by their completion.

Subset 1

Subset 1 will be a very simple language with only a limited number of types and language constructs. The subset will contain only the basic boolean, integer, floating point and `unit` types, basic string support (for input/output), top level function declarations with `let` and `let rec`, which only accept one argument (so as to sidestep the problem of partial application and closures), `if`, `for`, `while`, `ref`, as well as basic arithmetic and boolean operations.

These features were chosen because because they very easily make for a small minimalist language, and all constructs more or less have direct analogues in C meaning translation from AST to C code should be fairly easy. Nonetheless, this small subset is sufficient to write simple programs.

Subset 2

Subset 2 will focus on the implementation of types and polymorphism, and will support tuples, lists, parametric polymorphism, algebraic data types, record types, and `match` expressions.

The implementation of this subset will require the design of an object representation for OCaml values. Tuples and record types map easily to structs in C, while a tagged union can be used for OCaml's "sum of products" approach to algebraic data types. On

the first pass, it is quite likely that a naive "tagged pointer" type will be used for support for polymorphism, but extension tasks may experiment with eliminating this need via some method.

The addition of algebraic data types greatly increases the scope of supported programs, and allows for functions taking multiple arguments (via tuples), list processing functions, and more complex data structures such as binary trees.

Subset 3

Subset 3 will finally add treatment of functions as first-class values. This will include lambdas, lexical closures, and partial application of functions.

The implementation of this subset will require some representation of closures. While C does support function pointers, it has no lexical closures and as such a representation where closures are function pointers with an extra `void *` pointer that refers to data in the closure is likely to be the implementation on the first pass. As an extension task, more performant ways of implementing closures may be investigated, such as potentially dynamically generating entry points that push the required arguments before jumping to the body of the function.

The addition of these features will make higher-order functions representable, and make available functions such as `map`, `filter`, function composition etc. At this point subset 3 can be considered a fairly minimal but full-featured functional language.

Evaluation

Evaluation will require the creation of a small library of test programs. These programs should be novel and exhibit the full range of language features supported by the subsets.

Two main methods will then be used to evaluate the compiler. One is benchmarking performance of the test programs with the existing OCaml native compiler and the OCaml bytecode compiler. While it is not expected that the C compiler will match the performance of OCaml official compilers, the goal is to be within a reasonable margin so that the performance of compiled C code is comparable. The performance evaluation will be a simple measurement of the runtime of the compiled executable in relation to the running time of the executable produced by the OCaml native compiler and the running time of the interpreter on the bytecode produced by the OCaml bytecode compiler. The test suite for the performance evaluation will have to be partly written as part of the project, and partly adapted from the existing OCaml compiler test suite (`github.com/ocaml/ocaml/tree/trunk/testsuite`) or possibly the Computer Language Benchmarks Game (`benchmarksgame.alioth.debian.org`). Since the compilation source is only a subset of OCaml, not all of the existing test suite programs will be relevant, and others will require some adaptation for them to work for the project.

The other method will be the evaluation of the "observability" of the C code. This will take place in two parts: firstly, we can do a test for "observability": this will entail a minimum list of requirements, such as ability to step through OCaml code, ability to print locally bound values, ability to print types of locally bound variables, and ability to show

instantiation of polymorphic functions into specific types. The second is to compare the debug outputs of the C code and the OCaml code. The native OCaml compiler does not allow observation of locally bound values at all, so our C compiler should beat this easily; although a more interesting comparison would be with the OCaml bytecode debugger. Breakpoints could be set at randomly chosen call sites, and the stack traces, visible locals etc. could be compared between `ocamldebug` and the debug output of the C code.

Extension Tasks

This project leaves a lot of room for extension tasks. An obvious extension is to extend the subset to be a more complete subset of OCaml. The next steps taken would likely be module and exception support, although both are far more complex features to attempt for implementation in C.

More realistic extension tasks would be changes to the existing code in order to make it more performant. This would involve improvements to the implementation of polymorphic functions and closures, as described in their respective sections prior.

Another plausible extension task is to improve the debug output of OCaml values. Since C does not have an exact analogue for all OCaml values the debug output from `gdb` for example will not look exactly like their OCaml representation. However, `gdb` supports the use of custom formatters written in Python for types and a task to implement a way to generate formatters to print OCaml values correctly is a suitable extension task.

Success Criteria

The following criteria should be achieved at the completion of the project:

- A working compiler, that is able to compile the language as described in Subset 3 into C code correctly;
- An evaluation of performance between the output of the compiler and the OCaml native and bytecode compilers;
- An evaluation of "observability" in the compiled executable with tools such as `gdb`.

  The evaluation should focus on the following:

  - Ability to set breakpoints and step through the compiled executable as if it was OCaml code;
  - Ability to view locally bound variables, and their types, although perhaps only as a C "version" of the resulting data;
  - Ability to print more complex data structures, such as ADTs, record types and lists;
  - Similar and comparable output between the debug output and the same code running in `ocamldebug`.

Further success criteria, which are not expected to be achieved through the core project, but interesting as extension tasks may be:

- Ability to view the instantiated types of polymorphic functions;
- Ability to view data formatted as OCaml values, using specially written formatters for `gdb`;
- Any number of optimisations on the generated code, resulting in better performance benchmarks.

Timetable and Milestones

20 October — 3 November

Initial experiments and reading. Familiarise with the OCaml compiler source code and obtain typed AST. Set up development environment. Write testbed of evaluation programs so that features can be tested and evaluated immediately after their implementation.

Milestones: A completed small library of OCaml programs, plus a way of obtaining the typed AST from the OCaml compiler.

4 November — 17 November

Implementation of Subset 1. Evaluation of Subset 1 performance with respect to the OCaml native and bytecode compilers.

Milestones: A collection of programs now compileable by the Subset 1 compiler, along with their running times collected in comparison with the OCaml compilers.

18 November — 15 December

Implementation of Subset 2. This is expected to take longer than the implementation of Subset 1 to take into account that special care needs to be taken when designing object representations of OCaml values and implementation of parametric polymorphism. Evaluation of Subset 2 performance with respect to OCaml compilers.

Milestones: A larger collection of programs now compileable by the Subset 2 compiler, along with their performance results in comparison with the OCaml compilers.

16 December — 29 December

(Two week break for Christmas)

30 December — 12 January

Implementation of Subset 3. Evaluation of Subset 3 performance, as well as comparison between the debug output of the compiled code with `ocamldebug`.

Milestones: Completion of core project with the implementation of Subset 3. Some evaluation results (although perhaps not complete) for the core project.

13 January — 26 January

Write up of the Progress Report. Perform any new evaluation tasks that were missed but become apparent in the write up of the Progress Report. Review the remainder of the project in context with the existing work and determine which extension tasks are best to pursue.

Milestones: Completed core project with evaluation, with a completed Progress Report ready for submission. Entire project reviewed with supervisor and overseers.

27 January — 9 February

Submission of Progress Report. Revise evaluation tasks with feedback from Progress Report. Prepare Progress Report presentation.

10 February — 23 February

Start work on dissertation, as well as the implementation of ways of making the compiled code more performant. Investigation and use of perhaps `libffi` and `liballocs` to improve performance and observability.

Milestones: The introduction and preparation sections of the dissertation complete. Improvements on the compiler made, with evaluation results to show improvements in performance of the compiled output.

24 February — 9 March

Start writing the implementation section of dissertation. Implementation of the generation of custom formatters for `gdb`, which will make the debug output easier to read and interpret as OCaml values.

Milestones: Implementation section of the dissertation at least half complete. Improvements to debug output of debuggers run on the compiled executables.

10 March — 22 March

Finish writing implementation section of dissertation. Start evaluation section. Pick up any further evaluation tasks at this point that become apparent.

Milestones: Finished implementation section, evaluation section at least half complete. Revised evaluation tasks in accordance with the write up of the evaluation section.

23 March — 5 April

Finish evaluation and conclusion sections of the dissertation. Deliver draft dissertation to supervisor and director of studies for feedback. Spend time working on whatever is in greatest need of attention.

Milestones: Submit a complete draft of the dissertation for review to supervisor and director of studies.

6 April — 19 April

Revise dissertation based on feedback received. Finish any last tasks that require performing, which are hopefully minor at this point in time. Send revised drafts for further rounds of review.

20 April —

Final adjustments and revisions to dissertation and the body of code. Prepare for submission of dissertation.

Milestone: Submission of dissertation.

Resource Declaration

No special resources are required for this project other than those that are not already available as open source code.