William Robson

A JIT compiler for OCaml bytecode

Computer Science Tripos - Part II

Selwyn College

14 May 2021

Proforma

Name:	**William Robson**
College:	**Selwyn College**
Title:	**A JIT compiler for OCAml bytecode**
Examination:	**Computer Science Tripos Part II, 2021**
Word Count:	**TODO**
Project Originator:	**William Robson & Timothy Jones**
Project Supervisor:	**Timothy Jones**

Original Aims of the Project

These were the original aims

Work Completed

This was what was completed

Special Difficulties

This was difficult

Declaration

I, William Robson of Selwyn 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 William Robson

Date 14 May 2021

Contents

List of Figures

Awknowledgements

Something else

Chapter 1

Introduction

1.1 Overview

The OCaml compiler has two main backends: an optimised target-specific native code compiler and a compiler to bytecode which is later run by the interpreter, `ocamlrun`.

My project set out to create a just-in-time (JIT) compiler from OCaml bytecode to x86_64 machine code with the hope of achieving performance benefits.

The project achieved on schedule three core goals as set out in the proposal:

1. There is a JIT compiler implemented into the existing OCaml source replacing the interpreter with all functionality but debugging and introspection.
2. There is a comprehensive and automated suite of benchmarks built comparing its performance to other alternatives.
3. It performs favourably to the interpreter when discounting initial compile time for all programs and overall for longer running programs.

This gave some time for a significant extension project; this consisted of creating a slower but more optimising compiler being conditionally used for functions which are called often.

I will refer to the faster but less intelligent compiler as the 'first' or 'initial' compiler and the slower but more optimising compiler as the 'second' or 'optimising' compiler.

1.2 OCaml bytecode

Using the bytecode interpreter is significantly slower than the native code compiler - however the design of the interpreter is very sophisticated. Its design descends from that of the ZINC Abstract Machine (ZAM) [**zinc**].

Although most users of OCaml use the native code compiler nearly exclusively, the bytecode compiler is still useful. It is used primarily:

1. To run the OCaml REPL ('toplevel')

2. As a more stable binary format for tools such as `js_of_ocaml` to use as input
3. As a portable executable format for different machines
4. To run programs under a debugger
5. To allow language developers to experiment with new language features without needing to write target specific code generation

1.3 Related work

This project has been attempted at least twice: with OCamlJit in 2004 [**ocjit1**] and OCamlJit2 in 2010 [**ocjit2**]. On advice of my supervisor I decided not to closely read these papers until after implementation - my first compiler independently ended up working in a similar way to OCamlJit2 and my second was more sophisticated than either project.

This project fits into the space of JITs more generally as a method-based (as opposed to trace-based) compiler [**pyket**]. The first compiler uses `dynasm-rs` [**dynasmrs**] which works as a Rust macro turning assembly instructions into pre-assembled code and calls into a simple runtime for relocation. Its design is based on that of `DynASM` used in the `LuaJIT` project [**dynasm**].

The second compiler uses the `cranelift` [**cranelift**] library which is a low-level retargetable code generator with an emphasis on use in JITs[1]. It is somewhat similar to LLVM but lower level and much simpler. It cannot perform many optimisations LLVM can but has significantly lower compilation time.

1.4 Implementation constraints

In order to limit the scope of the project some decisions were made upfront.

1. This project only modifies the runtime components of the OCaml compiler and does not modify the bytecode format or behaviour of the compiler, even if this would make life simpler
2. The only target considered is x86_64 Linux
3. Correctness should be prioritised over completeness and existing OCaml semantics should be preserved

Despite the third item, the initial compiler is both correct and can run fully JIT-compiled every OCaml program I have tested. The features not supported are the debugger and backtraces - however these are optional things that need to be explicitly requested by the user. A significant milestone achieved was allowing the compiler to self-host.

[1]It's largest use is for JIT-compiling webassembly as part of `wasmtime` and Firefox

Chapter 2

Preparation

2.1 The OCaml bytecode interpreter

2.1.1 Core design decisions

The interpreter has its roots in the ZAM[**zinc**] in many ways. The most significant of these for this project was the model used for calling functions and dealing with the functional language concepts of tail calls and partial application. In a 2005 talk at the KAZAM workshop [**xavtalk**], Xavier Leroy describes the concept of a distinction between the 'eval-apply' and 'push-enter' models. Leroy attributes this distinction to Simon Peyton Jones.

Consider a simple de Bruijin-indexed lambda calculus:

$$\begin{aligned} e ::= & \;|0|1|\ldots && \text{Variable} \\ & |\lambda e && \text{Abstraction} \\ & |e\, e && \text{Application} \end{aligned}$$

Simple scheme

A simple scheme[1] is to define the following:

$$\begin{aligned} \mathcal{C}[\![n]\!] &= \mathtt{ACCESS}(n) \\ \mathcal{C}[\![\lambda e]\!] &= \mathtt{CLOSURE}(\mathcal{C}[\![a]\!];\mathtt{RETURN}) \\ \mathcal{C}[\![e_1\, e_2]\!] &= \mathcal{C}[\![e_1]\!];\mathcal{C}[\![e_2]\!];\mathtt{APPLY} \end{aligned}$$

The machine has code, an environment and a stack (where all behave someone like linked lists). The transitions are:

[1]Similar to that of the Part IB Compiler Construction course

Before			After		
Code	Env	Stack	Code	Env	Stack
ACCESS(n);c	e	s	c	e	$e(n);s$
CLOSURE(c');c	e	s	c	e	$\langle c',e'\rangle;s$
APPLY;c	e	$v;\langle c',e\rangle;s$	c'	$v;e$	$c;e;s$
RETURN(n);c	e	$v;c';e';s$	c'	e'	$v;s$

$\langle c,e\rangle$ denotes a closure

While conceptually simple, this module has problems with multiple argument curried functions. Calling any multiple-function argument requires creating and then discarding multiple closures. In order to deal with this we need special support.

Eval-apply model

In Leroy's eval-apply model the callee always receives exactly the number of arguments it expects.

This is true of most imperative languages but is also a viable strategy for implementing functional languages:

When compiling the caller, the compiler knows of any arity mismatches and can insert the code for partial applications/tail calls at compile time. This is what the OCaml native code compiler does.

Push-enter

THIS IS CONFUSING

By contrast, in the push-enter model it is the job of the callee to check the arguments it has been given and deal with the cases itself.

In the OCaml interpreter this is done in a somewhat involved way. There is a machine register called `extra_args` which is set by the caller of a function. Arguments are passed on the stack followed by a return frame. `extra_args` contains 1 less than the number of passed arguments.

If a function takes more than 1 argument, the first instruction is GRAB(n). If `extra_args` $\geq n$, it decrements `extra_args` by n.

However, if this is not the case then the function has been partially applied and `extra_args` $< n$. The instruction instead returns a closure containing in its environment the arguments that were passed and a special code pointer to an instruction called RESTART.

When RESTART is executed it pops those arguments back on to the stack behind the arguments now provided but before the return frame and jumps to try the GRAB again.

This represents partial application. Consider the OCaml function:

returns an integer but returns a new partially-applied function that remembers the 1 passed to instruction.

Implications

The push-enter/'callee deals with arity mismatch' model was one of the biggest complexities involved with making this JIT. In the bytecode compiler every single call is an indirect lookup through a closure, whereas the eval-apply model allows calls to be translated down to simple direct function calls.

2.1.2 Data representation

OCaml has a uniform data representation for its values. Values are 64 bits long. Pointers to heap-allocated values are stored directly - but due to alignment they are guaranteed to have a 0 in the LSB. Integers are 63 bits long with the unused LSB storing a value of 1.

Every heap allocated value has a `u64` header containing the number of words stored (called the `wosize`) and a `u8` tag. Most tag values correspond to a block which can be thought of as a tagged tuple containing `wosize` fields. Each field is treated as an OCaml value by the garbage collector.

There are special tag values and cases in the garbage collector for things like floating point numbers[2], closures, objects and `f64`/`u8` arrays.

Registers

The OCaml abstract machine uses five registers:

- `sp` is a stack pointer for the OCaml stack which is used extensively in the bytecode.
- `accu` is an accumulator register used for the return values of functions and primitives as well as for most arithmetic operations.
- `env` holds a pointer to the current closure (an OCaml block) which is used for referencing closure variables (stored as fields in the block)
- `extra_args` is used for the interpreters implicit method for currying and tail-call explained in section TODO
- `pc` contains the pointer to the bytecode instruction to be interpreted next

In addition to these registers there is the `Caml_state` struct whose fields can be considered as another 30 or so registers. They are used by the interpreter mainly for supporting the garbage collector, exceptions and growing and reallocating the OCaml stack.

[2]which due to the uniform representation must be stored boxed on the heap

2.1.3 Garbage collector

OCaml is a garbage collected language - memory is managed by the runtime and released once it is no longer reachable from any other live object.

As it is a functional language, short-lived immutable values are created and dropped very frequently. For this reason OCaml uses a generational garbage collector: there is a minor heap and a major heap. Allocations are done by pointer bump in the minor heap until it becomes full. At that point the runtime branches into the garbage collector which compacts anything alive in the minor heap.

After something survives for a while they are moved to the major heap which is collected much less frequently.

OCaml also requires a write barrier for every assignment to a field.

Safepoints

A useful abstraction in the implementation of code interacting with a GC is that of the safepoint. This is a point in the program (usually a function call) where pointers might end up relocated.

For OCaml this can happen:

- when the minor heap is full during an allocation and the allocation routine branches into the GC
- when a C primitive is called (which may itself allocate memory and trigger the GC)
- when responding to signal handlers

As the garbage collector relocates objects, it is important the runtime can find all GC roots (pointers to heap-allocated values). For the interpreter, all roots are stored on the stack. For this reason the interpreter spills the accu and env to the stack at every safepoint.

Instruction format

There are 149 opcodes defined. Each opcode takes a certain[3] number of arguments. Opcodes and arguments are stored as `i32` values.

However many opcodes are the composition of a few simpler opcodes to save space in the format. For example `PUSHGETGLOBALFIELD(x, y)` can be expanded to `PUSH, GETGLOBAL(x), GETFIELD(y)` or `ACC2` (only opcode) is the same as `ACC(2)` (with the 2 as the operand).

For my JIT I did these expansions to arrive at 62 instructions[4].

[3] or in the case of `SWITCH`, variable

[4] where all binary arithmetic and comparison instructions use the same instruction type – so in practice about 80 distinct instructions

Most operations are fairly standard for a stack based machine with an accumulator - there are commands for stack manipulation and loading and storing the accumulator to the stack, performing arithmetic using the accumulator and the stack, conditional branches, switch statements, calling C primitives and allocation of OCaml blocks.

2.2 Compiler concepts

Reference basic blocks, dataflow analysis, stack based VMs

2.3 Primer on x86_64 assembly

Talk about CISC design + addressing modes, registers,

2.3.1 System V calling convention

2.4 Starting point

2.4.1 Before formal start

Before formal project start I forked the OCaml compiler (version 4.11.1) and implemented a proof of concept bidirectional FFI between C and Rust.

I also wrote a parser from the bytecode into Rust data types and a simple disassembler that uses it.

2.4.2 Dependencies forked during the project

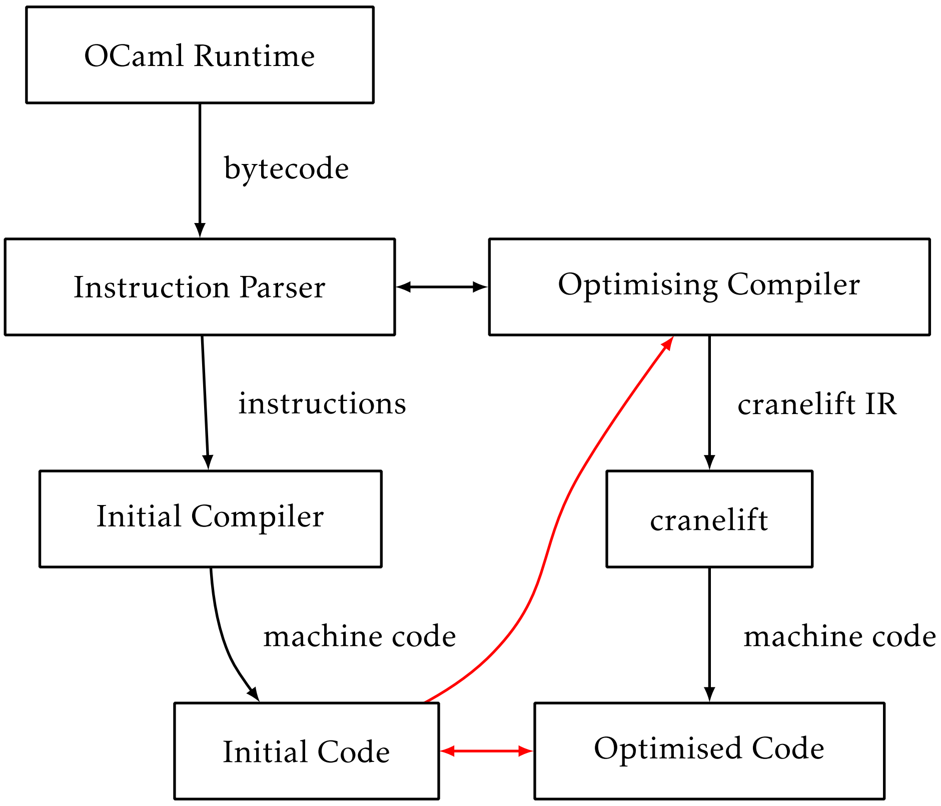
I use a lot of open source libraries/crates (as is usual to do in Rust). Some dependencies needed to be tweaked slightly and where that is the case they are vendored in to the source tree.

I based my benchmark suite from the excellent Sandmark project by OCaml Labs at Cambridge but made a large amount of mechanical modifications to modify the tool to support bytecode benchmarks.

Chapter 3

Implementation

3.1 Architectural overview



I have implemented a Rust static library which is linked in to the OCaml runtime library and `ocamlrun` (the interpreter). `ocamlrun` hooks into this library when it loads bytecode and when it starts interpreting it. If the JIT is enabled (either by setting an environment variable or enabling it by default at compile time), when the bytecode load hook is run the **initial compiler** will execute.

The initial compiler parses the bytecode into a stream of instructions and for each bytecode instruction it emits assembly with the same semantics to a buffer. After all code has been emitted (including headers and footers with shared routines) relocations are performed and the buffer is marked as executable.

When `ocamlrun` then calls the hook to start interpretation, the library instead jumps to this assembly code which then performs the same operations as the interpreter - except that every instruction has effectively been inlined.

When an OCaml closure is applied[1] a closure execution count is incremented. Once this count passes a configurable threshold the code instead branches in to the **optimising compiler**.

The optimising compiler operates on the level of a single function. It starts by doing a depth-first search to discover all the basic blocks in the function. It then iterates over this again producing Cranelift IR Format (CLIF) which is passed to the machinery of **cranelift**. The CLIF code produced abstracts away the use of the stack allowing cranelift to perform its register allocation. The function's code pointer is updated and any future calls (including the originally triggering call) will use the optimised implementation.

3.2 Major milestones

The focus throughout all aspects of the project was focusing on working code that could be tested even if the total wasn't complete. However it is worth noting the major milestones where some degree of completeness was achieved. These entries are listed in chronological order.

1. Execution of Hello World with the JIT
2. Passing the OCaml test suite and bootstrapping the compiler
3. Implementing the benchmark suite
4. (Extension) Developing the advanced disassembler tools
5. (Extension) Implementing the optimising compiler

3.3 Repository overview

A more complete overview of the repository is given in section **??**. However, it is worth drawing attention to which components are solely, partially or not at all my own work:

```
/
├── benchmarks
│   ├── sandmark ............................ fork of Sandmark with large patches
│   └── analysis ................. Jupyter notebooks analysing benchmark results
├── docs ..................... LaTeX source of proposal, report and this document
└── scripts ........................... scripts to run tests and graph basic blocks
```

[1]all non-primitive function calls are translated to closure application

```
│
├── src
│   ├── rust
│   │   ├── ocaml-jit-shared ......... shared library between the other two crates
│   │   ├── ocaml-jit-staticlib ............... crate linked in to OCaml runtime
│   │   └── ocaml-jit-tools ... binary crate with standalone development/testing
│   │       tools
│   ├── ocaml ..................................... fork of OCaml compiler 4.11.1
│   │   └── runtime ......................... contains most large patches for the JIT
│   └── vendor ................... vendored rust dependencies with small patches
├── test-programs ....... OCaml source for the test programs used by the scripts
└── vendor/no-aslr ........... A simple wrapper to run a program without ASLR
```

3.4 Initial compiler

The initial compiler was developed first before any work on the optimising compiler was done. In the process of implementing the initial compiler some modifications had to be made. These modifications will be covered later in section **??** and this section will mainly cover the initial state

3.4.1 Overview

The basic compiler consists of two larger components - an instruction parser and an assembly emitter.

Most of the rest of the files are support for these components and deal with the FFI from and into the existing runtime.

Mapping of OCaml abstract machine

The initial compiler operates by converting every bytecode instruction into a stream of assembly instructions with the same semantics. Abstract machine registers are mapped to x86_64 registers. A pointer to the `Caml_state` struct containing other global interpreter state is also stored in a register called `r_cs` for access to its fields with small code size. The mapping used is shown in Table **??**.

Note that all x86_64 registers mapped are callee-saved in the System V C calling convention. This means that no work needs to be done spilling and restoring them from the C stack to call C/Rust primitives (either as part of `CCall*` instructions or supporting primitives written for the JIT).

Mapping of the PC

Bytecode pointers are mapped to native code pointers and the interpreter's PC is replaced with the system's instruction pointer. This change is responsible for nearly all of the performance improvements of the JITed code. The benefits are

OCaml register	x86_64 register
`r_env`	`r12`
`r_accu`	`r13`
`r_extra_args`	`r14`
`r_sp`	`r15`
`r_cs`	`rbx`

Table 3.1: Mapping of OCaml to x86_64 registers

- Memory accesses are reduced - rather than loading operands and opcodes, they are baked in to the machine code
- The CPU can predict execution and fill its pipeline further than it can with the use of native code pointers
- Branch prediction in hot paths become more effective as each branch can be predicted differently
- Likewise, the CPU can schedule out-of-order execution more effectively across different instructions

However, this approach does lead to more contention on the instruction cache - rather than only containing a single copy of the implementation of each instruction there are many.

3.4.2 Hooks from OCaml

Code at `src/rust/ocaml-jit-staticlib/src/c_entrypoints.rs`

The existing OCaml runtime was modified to hook into the JIT at three points:

1. After a bytecode section is loaded
2. Before a bytecode section is released
3. When the interpreter is called

In most programs there are only 1[2] sections loaded. However, the interpreter is used in other places such as the toplevel REPL which motivates the more general multiple section support.

Compilation happens after the first hook and an entry is written to a global table containing the pointer to the buffer. The compiled code takes the form of a single large C function taking no arguments.

[2]or 2 if callbacks from C to OCaml are used

When a section is released this entry is dropped which frees the memory.

The interpreter call hook then calls the JIT-compiled function representing that section.

3.4.3 Instruction parsing

The first stage in the pipeline is to parse the bytes into a stream of elements of the `Instruction` type.

The `Instruction` type
Code at `src/rust/ocaml-jit-shared/src/instructions/types.rs`

OCaml has 149 opcodes, each of which can take different number of operands. Arguments and operands are all stored as `i32` values. Rust has good support for algabreic sum types called enums (which are much more general than the enums in a language like C).

Some of these opcodes are code-size optimisations which represent the composition of multiple consecutive instructions or have a specific hard-coded operand value. For example, `PUSHACC2` represents `PUSH` then `ACC2`. `ACC2` itself is a specialisation of the more general `ACC(n)` instruction. These peephole-optimisation enabling instructions make less sense for the purposes of our interpreter so only the simplest instructions are considered.

Likewise, there are instructions like integer comparisons and conditional branches that have many different variations changing only on the exact conditional used.

The instruction type is polymorphic over the type of the label to allow for different label representations.

Parser
Code at `src/rust/ocaml-jit-shared/src/instruction/parse.rs`

The parser is implemented using Rust iterators; the parser takes an iterator of `i32`s and produces an iterator producing values of type `Result<Instruction, ParseError>`[3].

This use of iterators makes the instruction parser consistently performant for large bytecodes - consumers of the instruction stream take each instruction as they need it.

Labels are stored in the OCaml bytecode as indirect offsets relative to the current PC. To simplify later uses, these are converted to absolute offsets relative to the start of the section.

As mentioned, single OCaml instructions may parse to more than one `Instructions`. To support this, the type contains a pseudo-instruction called `LabelDef` which is emitted at the start of every OCaml instruction. This also allows building a map from OCaml bytecode absolute offsets to parsed `Instructions`.

[3]standard rust error handling type defined as enum `Result<T,E> {Ok(T), Err(E)}` like OCaml's `type ('a, 'b) result = Ok of 'a | Err of 'b`

3.4.4 Code generation

Code at `src/rust/ocaml-jit-staticlib/src/compiler/emit_code.rs`

The code generation is the largest aspect of the compiler. It makes use of the `dynasm-rs` library. This library manages emitting machine code and relocation information, relocating and using `memmap` to mark the code as executable.

The compiler is triggered on the first time a 'section' is loaded - for normal programs this is at startup and for programs using the OCaml toplevel REPL this is after every statement is typed.

The compiler first emits a standard function header entrypoint which saves callee-saved registers used by the compiler and aligns the C stack. A longjmp handler is set up for exceptions and then for each bytecode instruction assembly with the same semantics is emitted.

During this process `dynasm-rs` dynamic labels are used to set up relocations: these labels are defined before every bytecode instruction and can be referenced by any other instruction. DynASM translates these at runtime into pc-relative jumps.

After all instructions are done, some shared code used by the instructions is emitted. `dynasm-rs` then performs relocations and uses `mmap` to mark the region of code as executable.

The overall signature of the assembly produced by this process is a single function taking no arguments and returning an OCaml value. OCaml closure applications (function calls) do not produce a stack frame on the C stack - the existing machinery using the OCaml stack is used instead.

Simple example

The main code of the compiler itself is contained in a 2000 line file (`src/rust/ocaml-jit-staticlib/src/compiler/emit_code.rs`). Most of this is taken up by a Rust large pattern match for each of the bytecode instructions. As a very simple example of what the code looks like consider the implemenation of the `Add` instruction. It adds the value at the top of the OCaml stack to the accumulator and stores the result in the accumulator. Note that the OCaml integer format means a decrement is required. In the original assembler it is implemented as so:

In the compiler the case becomes:

This has exactly the same semantics. Note `r_accu` and `r_sp` are aliases for `r13` and `r15`.

At compile time, the macro component of `dynasm-rs` translates it to:

Most assembly work is done at compile time - the byte string above contains the machine code for those instructions. This helps a lot with the performance of the compiler.

Branches

For example of the label and relocation support, consider the `BranchCmp(Comp, i32, L)` instruction. It compares the current value of the accumulator to the `i32` constant using the condition (of type `enum Comp Lt, Gt, ...`). If the condition is true it branches to the label, otherwise it passes to the next instruction.

This is implemented as so:

The macros translate it to:

This shows the assembler work still needing to be done at compile time - relocations.

Other cases

Most other cases follow these patterns. Some more involved instructions will call into C primitives I wrote instead (passing and then restoring the registers from the stack).

However, the basic structure of snippets of combined assembly and rust in a large pattern match statement remains for all cases.

3.4.5 Futher details

Although the basic structure as described holds, some details are omitted but appear in `emit_code.rs`.

- There is extensive optional support to support tracing instructions and events (described in section **??**)
- Callbacks from C to OCaml code require special handling
- The compiler returns a pointer to the first instruction to support OCaml's metaprogramming `ocaml_reify_bytecode` program used in the implementation of things like the toplevel `ocaml` program.
- Function application is a fairly involved process and there are checks to resize the stack and check for signals as well as the fundamental complexity of the push-enter model.
- Registers need to be saved and restored at safepoints to allow the garbage collector to use them
- Certain operations are involved enough that instead of inline the definition in hand-written assembly, I push the registers to the C stack and call a C primitive to implement the operation (taking the registers as a struct)
- The compiler stores a persistent data structure of the sections to allow mapping bytecode addresses to machine code addresses and allow for cleanup after a section is freed.

3.5 Optimising compiler

3.5.1 Motivation and design

Once the initial compiler was complete there was time to dedicate to a significant extension. Although the initial compiler was performant, it lacked any optimisations between multiple instructions.

The proposal lists a few ideas of extensions but does not commit to any one of them. Of these, the decision made would be to focus on combining two concepts:

- Dynamic recompilation of hot functions with more optimised forms
- Replacing the use of the stack with registers

The details of the first extension are explained later in section **??** and the second extension forms the basis of the optimised compiler the first extension calls.

The time left meant building a complete x86_64 compiler backend (with register allocation, instruction selection, etc.) from scratch was infeasible. The usual toolkit used to avoid replicating this work is LLVM [**llvm**]. Although is more typically known for its use in ahead-of-time compilation, there is support for use in JITs.

Problems with LLVM

I initially decided to go with LLVM but as the implementation started I ran into some limitations.

- LLVM is very large and bloated - compiling it with multiple jobs caused my machine to run out of memory and linking it in massively bloated the binary size of `ocamlrun`
- The safepoint GC support while somewhat mature is messy and would require writing patching the C++ source of LLVM to add new options as well as diving very deep into internal data structures
- Although there are some very good Rust bindings, not all features of the C++ api are exposed. This is especially true with the garbage collector support.
- LLVM has about 2 different JIT interfaces (`MCJit`, `orc`, `orcv2`), all of which had different limitations when used in a project such as mine and only `MCJit` had good Rust bindings
- LLVM is somewhat heavyweight and compilation is slow which means there is a large risk of slower overall performance even if only compiling hot functions.

Cranelift

After struggling with these issues I discovered the `cranelift` project. Like LLVM it is a retargetable code generator with an IR. However, it has a number of different design decisions which made it much better suited for my planned uses:

- It is written in Rust which means the API is fairly idiomatic when using it in Rust
- It is designed primarily for JITs and focuses heavily on compilation performance
- The project is actively developed and I could communicate with the developers when I had questions who were very responsive.
- Although the support for garbage collection is not as extensible, with the help of the developers I was able to come up with a model that is a good fit for the OCaml garbage collector's requirements.
- When I encountered any bugs or missing features I could easily submit and get merged patches to the project which uses a familiar Github workflow rathter than the LLVM project's legacy systems.

There were still some missing features in Cranelift which are described later but for the most part it was a good fit for my needs and was in a large part responsible for my being able to complete this ambitious extension.

3.5.2 Overview

At a high level the optimised compiler is a function taking two inputs and returning two outputs. The inputs are a slice to the code of a section of a bytecode, and the offset within that slice containing the first instruction in the function to be optimised.

The first output is a pointer to a compiled function that can be called to execute the code represented by that function. The second output is a vector of stack map entries which are used to support finding GC roots (see section **??**).

The compiler consists of three major components.

1. Basic block conversion and stack start analysis (section **??**)
2. IR generation from the basic blocks using the analysis (section **??**)
3. Compilation of IR to x86_64 assembly (done by Cranelift)

As described later in section **??**, this compiler is triggered once a function is called enough times to become 'hot'. After the function is compiled the optimised implementation will always be used.

3.5.3 Changes

The largest change from the initial compiler to the optimised compiler is the removal of use of the OCaml stack and accumulator registers. This is done by replacing these uses of the OCaml stack with machine registers and the C stack.

Arguments are passed to a function according to the System V C calling convention - the first argument is always the closure environment, `env`, and the remaining arguments are passed in registers. Direct calls from optimised functions to optimised functions can avoid the use of the OCaml stack entirely. Other cases need more work (described in section **??**).

The method used to remove the use of the stack is to convert the OCaml bytecode into an SSA form by keeping track of the contents of a virtual stack and accumulator as the closure is translated. As an example consider the simple OCaml function:

This compiles to the sequence of bytecode instructions:

```
Grab(1)          # Ensure 2 arguments passed
Acc(1)           # Load 2nd element from top of stack to acc
Push             # Push acc
Acc(1)           # Load new 2nd element from top of stack to acc
ArithInt(Add)    # Add top of stack to the acc and pop it
Return(2)        # Pop 2 items, then return acc
```

Instead of translating all of these inefficient stack manipulations as the initial compiler does, the optimised compiler keeps track of the stack as it compiles. For an example of this, see table **??**. Ignore the `Grab(1)` for now as it is handled in a different way.

Initial state	Operation	Final state
`acc=?, stack=[b, a]`	`Acc(1)`	`acc=a, stack=[b, a]`
`acc=a, stack=[b, a]`	`Push`	`acc=a, stack=[a, b, a]`
`acc=a, stack=[a, b, a]`	`Acc(1)`	`acc=b, stack=[a, b, a]`
`acc=b, stack=[a, b, a]`	`ArithInt(Add)`	`acc=a+b, stack=[b, a]`
`acc=a+b, stack=[b, a]`	`Return(2)`	`acc=a+b, stack=[]`

Table 3.2: Translation of add

This method is effective when considering a single basic block at a time. However, in the presence of branching, joining and looping basic blocks more work is needed. Luckily the output of the OCaml compiler has the invariant that the stack size relative to the function's stack frame is constant at the start of each basic block, *no matter the path taken to reach it*. This happens to be the key to allow efficient SSA conversion, completely removing any explicit notion of a stack and accumulator from the generated Cranelift IR.

3.5.4 Conversion to basic blocks

Code at `src/rust/ocaml-jit-shared/src/basic_blocks/`

The typical data structure used in optimising compilers is the basic block. *Make sure I've defined this somewhere in preparation.*

However, the only information we have from the bytecode is a sequence of instructions with jumps. In order to move away from a heavy connection with the instructions the first step is to decompile to basic blocks.

Types

Code at `basic_blocks/types.rs`

A **`BasicClosure`** consists of a vector of **`BasicBlock`** along with some other data. A basic block has a vector of **`BasicBlockInstruction`** and a single **`BasicBlockExit`**.

Some `Instructions` (see section **??**) map to a `BasicBlockInstruction` and others map to a `BasicBlockExit`. For example, `Instruction::Push` becomes `BasicBlock-Instruction::Push` but `Instruction::BranchIf(label)` becomes `BasicBlockExit::BranchIf then_block, else_block .`

This enforces at the type level the basic invariant of basic blocks - one entry, one exit.

Stack starts

In addition to performing the conversion to basic blocks, the algorithm also keeps track of a concept I call the **stack start** of a basic block. Although, like all other aspects of the OCaml interpreter, it is not documented anywhere, inspecting the bytecode compiler's source and testing against all real programs finds an important invariant:

For all paths leading from the entry block of a function to an instruction, the absolute stack size relative to the start of the function's stack frame is the same.

During the conversion to basic blocks this invariant is validated and the size of the stack before the first instruction of a block (the stack start) executes. This is the **stack start** of the block.

Algorithm

Code at `basic_blocks/conversion.rs`

The algorithm consists of two runs of DFS. The first finds all bytecode offsets which form the start of blocks. The second visits all these blocks working out stack sizes.

In order to describe this algorithm without too many confusing details I will create a formalism just complicated enough to include all the complexity.

The first DFS pass is only necessary to deal with loop back edges. For simplicity let's assume this doesn't happen.

Let I be the set of all instructions. The program P is a list of instructions and can be indexed. For simplicity, assume indices are consecutive[4].

Each instruction has four properties:

- $\delta(i)$ is the change to the stack pointer after executing the instruction. For example `Push` which pushes the acc to the stack has $\delta(\texttt{Push}) = +1$
- exit(i) returns **true** iff the instruction causes the block to end (jumps, returns)
- fallthrough(i) returns **true** iff exit(i) and it can end up jumping to the following instruction. `BranchIf(branch_loc)` is fallthrough but `Branch(branch_loc)` is not.
- labels(i) returns a set of all labels the instruction could end up jumping to (not including fallthrough). For example labels(`BranchIf(branch_loc)`) = {branch_loc}. It is only relevant if exit(i)

Under this model, the algorithm to find all the blocks and their starts is shown in algorithm **??**. The actual algorithm is more complicated and calculates more things but not conceptually any different.

Additional operations performed during the search

The algorithm as presented in algorithm **??** shows only the basic logic of the search and storing of the stack starts for each block. The actual search also performs more work:

- When a block is visited a vector is created to hold the parsed instructions
- Blocks are numbered in reverse post-order (the natural order for basic blocks where with the exception of loop back edges, blocks are visited in a way consistent with execution order)
- Every time an instruction is visit any labels referencing other blocks are replaced with the block ids and a copy is added to the vector.
- At exit, all of the block metadata is stored in a `BasicBlock` struct and placed in a table of blocks referenced by the block number.

Example

3.5.5 Calling conventions

In the initial interpreter, all calling is done using the OCaml interpreter's existing model. This passes all arguments on the OCaml stack and has the caller deal with partial application and tail calls.

[4]they're not but it's fairly simple to handle

Algorithm 1 DFS to find basic blocks and their starts

```
1:  function FindBlocks($P, arity$)
2:      $seen \leftarrow \emptyset$
3:      $starts \leftarrow \{\}$
4:      function VisitBlock($initial\_index, initial\_stack\_size$)
5:          if $initial\_index \in seen$ then
6:              assert $starts[initial\_index] = initial\_stack\_size$
7:              return
8:          end if
9:          $starts[initial\_index] \leftarrow initial\_stack\_size$
10:         $i \leftarrow initial\_index$
11:         $s \leftarrow initial\_stack\_size$
12:         repeat
13:             $x \leftarrow P[i]$
14:             $s \leftarrow s + \delta(x)$
15:             if exit($x$) then
16:                 for $j \in$ labels($i$) do
17:                     VisitBlock(j, s)
18:                 end for
19:                 if fallthrough($x$) then
20:                     VisitBlock(i + 1, s)
21:                 end if
22:                 return
23:             end if
24:             $i \leftarrow i + 1$
25:         until forever
26:     end function
27:     VisitBlock(0, arity)
28:     return $starts$
29: end function
```

We would like to avoid the stack in hot paths, so instead use an eval-apply model. Arguments are passed according to the System V calling convention and a function is only called with all the arguments it expects.

TODO: flesh this out

3.5.6 IR generation

TODO: code ref

Example: Add

Consider the add function:

```
Arity: 2
Max stack size: 3

# Block 0 (stack_start = 2)
Acc(1)
Push
Acc(1)
ArithInt(Add)
Exit: Return(2)
```

```
function u0:0(r64, r64, r64) -> r64, i64 system_v {

block0(v0: r64, v1: r64, v2: r64):
    v4 = iconst.i64 0
    v6 = raw_bitcast.i64 v1
    v7 = raw_bitcast.i64 v2
    v8 = iadd v6, v7
    v11 = iconst.i64 -1
    v9 = iadd v8, v11
    v10 = raw_bitcast.r64 v9
    jump block1

block1:
    return v10, v4
}
```

```
0000000000000000 <arith_add>:
   0: 55                    push   rbp
   1: 48 89 e5              mov    rbp,rsp
   4: 48 01 d6             add    rsi,rdx
   7: 48 83 c6 ff          add    rsi,0xffffffffffffffff
   b: 48 89 f0             mov    rax,rsi
   e: 48 31 d2             xor    rdx,rdx
  11: 48 89 ec             mov    rsp,rbp
  14: 5d                   pop    rbp
  15: c3                   ret
```

Clamp

```
Arity: 3
Max stack size: 4

# Block 0 (stack_start = 3)
Acc(2)
Push
Acc(1)
IntCmp(Lt)
Exit: BranchIf { then_block: 1, else_block: 2 }

# Block 1 (stack_start = 3)
Acc(0)
Exit: Return(3)

# Block 2 (stack_start = 3)
Acc(2)
Push
Acc(2)
IntCmp(Gt)
Exit: BranchIf { then_block: 3, else_block: 4 }

# Block 3 (stack_start = 3)
Acc(1)
Exit: Return(3)

# Block 4 (stack_start = 3)
Acc(2)
Exit: Return(3)
```

Cranelift IR basics

Cranelift, much like LLVM, uses an IR with a binary, in-memory and textual representation. The basic concept is that of typed values where the types are things like

I64 and F32. When constructing IR as part of the frontend, cranelift provides some assistance with translating dynamic mutable variables to SSA values inserting block parameters where needed.

3.5.7 GC support

A key aspect of the OCaml runtime is its garbage collection support. In the bytecode interpreter no particular effort is needed for this as every value is stored on the OCaml stack. However, for my compiled code which does not use the OCaml stack it is necessary to build something more involved.

As some of the steps are a bit fiddly I will start with an overview of how the steps involved fit together:

1. During cranelift IR generation, I store everything that could contain a pointer to a GC-managed value (a **local root**) in a cranelift **reference type** (R64).
2. Cranelift performs LVA, spilling and restoring of registers to the C stack and emission of **stack maps** at every GC **safepoint**.
3. My runtime stores the stack maps in a hash map keyed by the return address at time of the safepoint.
4. During GC, my runtime walks entire the frame pointer (`rbp`) chain. By looking at the return address and using the hash map to discover any stack maps, my runtime tells the OCaml GC about the location and value of any local roots which could contain pointers.

I am incredibly grateful for the assistance Nick Fitzgerald, Chris Fallin and other Bytecode Alliance members who helped me in the Zulip chat to make all this work and reviewed my pull requests. I am reasonably certain that I am the only non-wasmtime user of cranelift's GC support so far.

IR generation

Cranelift has support for precise garbage collection. It was originally created for webassembly reference types but luckily mapped well to the needs of OCaml. Precise (sometimes called *exact* or *accurate*) means that the GC is able to determine all roots and only the roots when tracing. The opposite is a *conservative* collector which allows for an over-estimate.

In order to use it my IR generation has to mark roots by storing them in values which have the R64 type. The R64 type means 64-bit reference and is compiled identically to I64. However, cranelift support will use this type to differentiate GC-managed pointers from other values.

For this reason, anything which is an OCaml value and could be a pointer to the heap **must** be stored in a R64 type at every point the GC could trigger (a *safepoint*).

However, despite being almost equivalent in implementation to `I64`s not all operations that can be done using `I64` values can be done using `R64` values. This is only really a problem because of OCaml's mixed integer/pointer data representation. The integer add operation is not implemented for `R64`s but is needed to add two integers stored in OCaml's uniform representation.

The somewhat hacky solution I came up with is to use the `raw_bitcast` operation to temporarily convert `R64` values to `I64`, perform any arithmatic, and then convert back.

The key invariant I had to hold to make this work was that these casted `R64`-derived `I64` values were only temporary to the implementaiton of the operation that uses them.

What cranelift does

None of this section is my work (except for my pull requests to Cranelift fixing bugs I encountered).

Cranelift's GC support works using the concept of safepoints. A safepoint is a point in the program where the garbage collector could run. Cranelift treats every function call as a safepoint[5].

At every safepoint (here function call), cranelift will ensure:

1. All live `R64` values have a copy on the stack so the GC could find them
2. No code or optimisation assumes that the old value of a `R64` is still valid after the safepoint

In order to determine the live reference types at every safepoint, cranelift performs a live variable analysis (LVA) pass. It then will spill and restore (push and pop) any reference type-containing machine registers to the C stack (other values will be on the stack already).

This spilling takes place as part of register allocation. As is typical for optimised solvers of NP-complete problems, cranelift's register allocator is thousands of lines of incredibly dense algorithmic code that I do not fully understand.

Stack maps

The output of this step is calls to a callback handler I provide with (native code offset, stackmap) tuples.

The stack map is a set of offsets relative to the stack pointer at time of call where reference typed values are located. To store these entries efficiently cranelift uses a bitset data structure. The interpretation of these offsets is shown in the following diagram (where `x` is each member of the stack map's set):

[5] I planned a way to make this configurable per-function with the cranelift developers but decided it would take too long at the late stage of the project

```
         Stack
       +-------------------+
       | Frame 0           |
       |                   |
  |    |                   |
  |    +-------------------+ <--- Frame 0's SP
  |    | Frame 1           |
Grows  |                   |
down   |                   |
  |    | Live GC reference | --+--
  |    |                   |   |
  |    |                   |   |
  V    |                   |   x = offset of live GC ref
       |                   |   |
       |                   |   |
       +-------------------+ --+--  <--- Frame 1's SP
       | Frame 2           |
       | ...               |
```

*This diagram is directly copied from Nick Fitzgerald's blog post [***refblog***] describing cranelift's GC. License: CC-BY-SA.*

Integration with OCaml

After compilation is done, I translate the native code offsets to an absolute address by adding the pointer to the first instruction in the compiled function. I store all stackmaps in a hashmap indexed by the native code offset.

During garbage collection, the garbage collector has points where it scans all of the "local roots". It passes a callback function which takes two arguments: the address of the root in memory and the value at that address.

In order to use my return address map, it is neccesary to walk up the chain of stack frames in the same way as a debugger does. One way I initially tried was to use Apple's `libunwind` library which is capable of doing this and used for things like C++ exception handling or Rust backtraces on panic. However, integrating this with a JIT is an undocumented mess involving emitting DWARF debug information and registering them against the runtime. Wasmtime/cranlift have actually done this work for their own use of the GC. However, I would have to manually do it for every function in the initial compiler which looked horrific.

I instead settled on a less complicated solution: use frame pointers. Frame pointer chains are optional in `x86_64` calling conventions and usually omitted by optimising compilers. However, by telling the Rust compilers and GCC to not emit frame pointers, and manually adding the `rbp` chain assembly to my initial compiler's output I could ensure that a linked list repeatedly dereferencing the initial `rbp` value would walk the enitre stack. The code to do this is as follows:

The `rust_jit_lookup_stack_maps` function does a look up in the hashmap and will call the function `f` with any roots if it gets a hit on the return address (which is at `bp + 1`).

Summary

Garbage collection is incredibly difficult to get right. Luckily, cranelift had some support for it I could reuse. My use of it was non-standard but ultimately succesful. Once I had this I needed a way to walk through the C stack. Initial approaches were unsuccesful but ultimately falling back to `-fno-omit-frame-pointer` worked.

I paid some performance penalty by using frame pointers and my hash map for looking up stack frames was unoptimised. However, once all components were in place the system worked remarkably well.

3.5.8 Exception handling

As should be clear from other sections, explaining all the intracacies of how OCaml deals with exceptions would take many pages. In fact, the only bytecode instruction unsupported by the optimised compiler is the `PushTrap` operation which is used in implementing `try-catch` operations.[6]

Without going in to too much detail, I am proud of my solution to one of these problems which I will attempt to summarise here

At a very high level, exception support in the existing interpreter complicated by the fact the interpreter could call a C primitive that itself called back in to the interpreter in a different C frame. To support this, OCaml uses C `sigsetjmp` and `siglongjmp` functions. These functions work by saving and restoring the values of all machine registers and POSIX signal masks to a buffer allowing for 'long' jumps up the stack.

Most OCaml exception raises do not require a siglongjmp as they do not need to pass through the callback stack. However, my optimised compiler emits most functions as C functions and in general exception raises do require a longjmp.

The buffer used to store the state to restore is rather large and in the naive implementaiton would need to be allocated by *every* function on the stack. This was a clear issue needing solving if there was any hope of the optimised compiler actually being faster.

My solution was to note that I only needed two registers to completely restore my interpreter's state - `rbp` and the x86_64 instruciton pointer. This is because the only places the longjmp could point to are at specific locations where all other registers can be seeded by other means. I ended up writing my own jump code to do this.

[6]One particularly hard to debug problem led to a 2003 bug report in French where Xavier Leroy thanked the reporter for a particularly 'interesting' problem to fix. I had unknowingly reintroduced the conditions that led to that bug and unfortuantely did not find it as interesting to fix.

Note inline gcc assembly is AT&T not Intel syntax as I have used for the rest of the project:

3.5.9 Limitations

Despite allowing me to achieve this incredibly complicated system in a relatively short amount of time, Cranelift is a young project which is not without its flaws.

1. Cranelift has no support for tail calls yet. I had to instead perform tail calls in a hand-written wrapper function.

2. There is no way to mark a primitive (such as a call to OCaml's GC write barrier, `caml_modify`) as not requiring a safepoint which means the emitted assembly tends to have too many spills

3. Despite my `R64` $\leftrightarrow$ `I64` workaround mostly working, I managed to trigger some obscure bugs in the register allocator by the non-standard use of reference types.

4. Cranelift has limited support for other calling conventions, or operating outside of the context of a C function.

5. Exception/trap handling is difficult in cranelift and essentially requires the use of libunwind. Libunwind itself was a mess. As such I do not support catching exceptions in optimised code

However, despite this I had a much easier time integrating cranelift than LLVM. I think its core design decisions make it fill a unique niche where LLVM cannot. I have no hesitations in reccomending it for JIT projects, as long as the user is comfortable with digging around in cranelift's source code a bit.

3.6 Dynamic recompilation

3.6.1 Motivations

3.6.2 Closure table

3.6.3 Efficient exception handling

3.7 Testing

3.7.1 Trace comparison

3.7.2 Expect tests

3.7.3 OCaml test suite

3.7.4 Self hosting

3.8 Benchmark suite

3.8.1 About Sandmark

3.8.2 Modifications made

3.8.3 Analysis

Chapter 4

Evaluation

4.1 Overview

4.2 Correctness

The initial compiler fully covered the entire instructino set

Nearly all tests passed

The second compiler could not do everything the initial compiler could but this was ok because it fell back correctly.

4.3 The benchmark suite - Sandmark

The OCaml Labs team here at cambridge

4.4 Benchmark results

4.4.1 Initial compiler

4.4.2 Overhead of supporting recompilation

4.4.3 Optimsed compiler

4.5 Evaluation of results and design trade-offs

Chapter 5

Conclusions

This is a thing

5.1 Lessons learned

5.2 Possible extensions

Appendix A

Original proposal