Computer Science Tripos

Part II Project Proposal Coversheet

Please fill in Part 1 of this form and attach it to the front of your Project Proposal.

Part 1

Name: Weston Metzler

CRSID: wm307

College: Homerton

Overseers: (Initials) pl-pdj

Title of Project: Compiling OCaml to WebAssembly

Date of submission: 22/10/2020

Will Human Participants be used? No

Project Originator: Timothy Jones

Signature: ----------

Project Supervisor: John Fawcett

Signature: ----------

Directors of Studies: John Fawcett

Signature: ----------

Special Resource Sponsor:

Signature: ----------

Special Resource Sponsor:

Signature: ----------

Above signatures to be obtained by the Student

Part 2

Overseer Signature 1: ----------

Overseer Signature 2: ----------

Overseers signatures to be obtained by Student Administration.

Overseers Notes:

Part 3

SA Date Received:

SA Signature Approved:

Weston Metzler
Homerton
wm307

Computer Science Tripos Part II Project Proposal

Compiling OCaml to WebAssembly

October 22, 2020

Project Originator: Timothy Jones

Project Supervisor: John Fawcett

Director of Studies: John Fawcett

Overseers: Pietro Lio, Jeremy Yallop

1 Introduction

My project proposal is to create a compiler from OCaml to WebAssembly to gain experience implementing compilers and gain familiarity with both OCaml and WebAssembly.

1.1 Why WebAssembly?

Web applications have unique portability constraints since it is not known ahead of time what architecture the application will run on. The web scripting language JavaScript was invented to solve this problem with interpreters built into every web browser. But even with modern engines to JIT compile JavaScript, certain aspects of the language, particularly dynamic typing limit the efficiency of web applications. WebAssembly is a virtual instruction set architecture created to allow web browsers to run assembly-like code. Running WebAssembly code sent to the browser takes 3 steps. First it must be decoded from binary, next it is validated (giving us some safety guarantees), and then we can execute it at near-native efficiency. The work of compiling a high level language (parsing, optimizing, etc.) has to be done just once before the client ever requests the web page. WebAssembly was also designed to be a secure sandbox for executing code with access to the environment (such as for I/O) controlled by the embedding environment (such as the browser). So, while WebAssembly was designed to be executed in browsers, it is a general spec having applications in other areas particularly when portability and security are necessary.

1.2 Why OCaml?

OCaml would be an interesting choice of language to compile to WebAssembly. OCaml shares some similarities with JavaScript, both supporting functional programming while having imperative features. OCaml's powerful type inference system gives the feel of a dynamically typed language like JavaScript, while still being statically typed to take advantage of compile-time type checking for generating efficient WebAssembly. I also chose OCaml because I'm interested in developing my skills and understanding of the language.

2 Description of the Work

This project will be to write a compiler from a core-subset of OCaml to WebAssembly. The compiler itself will be written in OCaml, taking advantage of the `ocamllex` and `ocamlyacc` compiler generators to generate the front-end. There will also be a semantic analysis phase for type inference and type checking. Then the back-end of the compiler responsible for code generation as well as a WebAssembly runtime will be needed to create executable WebAssembly. As an extension I would like to add support for algebraic effect handlers from the multicore extension of OCaml.

3 Starting Point

My starting point is fairly basic. I have done some research on WebAssembly, OCaml, and Multicore OCaml to write this proposal, but much more is needed in order to begin work. I also have knowledge of ML and compilers from the Part I courses Foundations of Programming Languages and Compiler Construction respectively. I will be using the `ocamllex` and `ocamlyacc` compiler generators to generate code for the lexical and syntactic analysis phases of the compiler. The compiler will target the WebAssembly text format, so in order to get executable code in the WebAssembly binary format the WebAssembly Binary Toolkit will be used as an assembler. Other than that the plan is to write the type checker, code generator, and runtime from scratch.

4 Substance and Structure of the Project

4.1 Core Subset of OCaml

Here are the features of OCaml I plan on supporting:

- Primitive types of `int`, `bool`, `float`
- Tuple types such as `int * bool`
- Variant types declared with `type` to give us abstract data types including lists.

  ```
  type IntTree = Leaf | Branch of IntTree * int * IntTree
  ```

- `let` and `let rec` for defining variables and functions (not including currying syntax). This means support for function types as well.
- Pattern matching using `match-with`. This is a generalization of `if-then-else`, so it is not included.

4.2 Project Structure

1. Formally specify the language which is a subset of OCaml that I am compiling. This means grammars and typing rules.
2. Write some sample programs in the subset language which exercise a range of functionality. This isn't the full extent of testing, but it will be part of the success criteria. Some good test programs might include be Fibonacci, 8 Queens, and Binary Search Tree implementations.
3. Use the grammar created along with `ocamllex` and `ocamlyacc` to generate a lexer and a parser.
4. Implement a type inference algorithm (Hindley-Milner) to infer types and simultaneously do type checking.
5. Implement code generation from our AST all the way down to WebAssembly. This is the backend of the compiler.
6. Write tests for the compiler. Unit tests, integration tests, and regression tests are important for maintaining confidence our software works. The programs from the above step could also be used for end-to-end testing. The Dune build system for OCaml would be useful for this project and includes a test framework.

5 Success Criteria

This project is a success if it can be shown to correctly compile programs in the language of the core subset described above into working WebAssembly. We can evaluate correctness by comparing the results of the sample programs executed in a suitable WebAssembly embedding (possibly Chrome) to the results of running it through the OCaml compiler.

6 Evaluation

OCaml is a very flexible language which gives the feel of dynamic typing while having a solid static type system. The goal of this project is to bring that

flexibility to the web and take advantages of WebAssembly's sandboxed environment with security and safety verifications. We can evaluate the compiler by comparing it to alternative compilers for the web. A sensible comparison will be selected later with performance measurements such as execution time, binary sizes, and memory usage being taken between my compiler and an alternative for the web. Due to the limited scope of a part two project and my lack of focus on compiler optimisations, comparisons of absolute performance to other WebAssembly compilers isn't always appropriate. Comparing the measured time complexity can be used in some cases to verify my compiler matches the functionality of others.

7 Extensions

I am very keen to add support for algebraic effects. Algebraic effects are the main concurrency mechanism of Multicore OCaml, and enable concurrency and parallelism in the language. Algebraic effects and handlers are a novel programming feature that would be fun to support. They enable programs to make use of exceptions (which our subset of OCaml lacks), impure operations such as I/O, and some methods for concurrency. It's important to note the separation of concepts of concurrency, which is a style of program where execution is interleaved, and parallelism, which is simultaneous execution. The goal of supporting algebraic effects does not require simultaneous execution although that would be an interesting further extension. Some combination of Web Workers and support for threading and atomics features in WebAssembly may allow a faithful compilation of Multicore OCaml for the web.

8 Timetable and Milestones

To manage risk involved in this project and meeting the hard deadlines, an iterative approach to the work packages will be taken. At each stage of the project, the goal is to have a simplified but testable compiler that works from end to end, filling it out with more features each work package.

Slot 0 - *10th October – 23rd October*

Deadlines

- Phase 1 Proposal Deadline – 12th October, 3 PM.
- Draft Proposal deadline – 16th October, 12 noon.

- Proposal Deadline – 23rd October, 12 noon.

Slot 1 - *24th October – 6th November* **Work Package 1**

Research OCaml. Gain familiarity with the language since I'll be writing the compiler in it, particularly Multicore OCaml and effect handlers for the extension.

Milestone 1: A draft dissertation chapter describing research into OCaml has been delivered.
Milestone 2: A suite of test programs for evaluating success exists.

Slot 2 - *7th November – 20th November* **Work Package 2**

Research WebAssembly. Gain familiarity with the target language and tools for using it. Consider how can garbage collection, fibers, and domains be implemented?

Milestone 3: A draft dissertation chapter describing research into WebAssembly has been delivered.

Slot 3 - *21st November – 4th December* **Work Package 3**

Implement a very simple compiler for simple numeric expressions.

Milestone 4: The compiler can handle a simple program `2 + 2` and produce WebAssembly which is shown to compute `4`

Slot 4 - *5th December – 18th December* **Slack Time**

Slot 5 - *19st December – 1st January* **Work Package 4**

Extend the compiler to handle `let..in` (not functions yet) and `match..with` expressions as well as `int`, `bool`, and `float` primitive types and their operations.

Milestone 5: The sample programs which use the implemented features can be compiled and executed (with the correct behaviour).

Slot 6 - *2nd January – 15th January* **Slack Time**

Slot 7 - *16th January – 29th January* **Work Package 5**

Extend the compiler to handle the remaining features of the core subset: functions, tuples, and variant types.

Milestone 6: All sample programs can be compiled and executed with correct behaviour.

Slot 8 - *30th January – 12th February* **Work Package 6**

Milestone 7: The Progress Report is complete **Deadlines**

- Progress Report Deadline – 5th February, 12 noon.
- Progress Report Presentations – 11th, 12th, 15th, 16th February, 2 PM.

Slot 9 - *13th February – 26th February* **Slack Time**

Slot 10 - *27th February – 12th March* **Work Package 7**

Work on the extension, researching to understand effect handlers.

Milestone 8: A draft dissertation chapter describing research into Multicore OCaml has been delivered.
Milestone 9: The set of sample programs has been extended to include programs using algebraic effects.

Slot 11 - *13th March – 26th March* **Slack Time**

Slot 12 - *29th March – 11th April* **Work Package 8**

Add compiler support for algebraic effects.

Milestone 10: All sample programs from the extended set can be compiled and executed with correct behaviour.

Slot 13 - *12th April – 25th April* **Work Package 9**

Prepare the dissertation document and write the first three chapters.

Milestone 11: The first two chapters of the dissertation are ready for review.

Slot 14 - *26th April – 9th May* **Work Package 10**

Gather evaluation data and finish the dissertation.

Milestone 12: The dissertation is finished and ready for submission.

Deadlines

- Dissertation Deadline – 14th May, 12 noon.
- Source Code Deadline – 14th May, 5 PM.

9 Resources Declaration

I will use my personal laptop for all project work. It's a modern thinkpad with an i7 running Ubuntu. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. All project files will be in version control and backed up to Github, so the chances of losing files is minimal. In the case that my laptop becomes unusable, other computing resources such as MCS machines or Homerton computing resources are available as backup.