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:	Ștefan - Alexandru Pițur	CRSID:	sap86
College:	Robinson	Project Checkers:(Initials)	djg11, djw1005
Title of Project:	Fully In-Place Updates in Functional Programming		
Date of submission:	15 October, 2023	Will Human Participants be used?	No
Project Originator:	Dr. Jeremy Yallop		
Project Supervisor:	Dr. Jeremy Yallop		
Directors of Studies:	Dr. Richard W Sharp		
Special Resource Sponsor:	N/A		
Special Resource Sponsor:	N/A		

Part 2

Project Checkers are to sign and comment in the students comments box on Moodle.

Part 3 For Teaching Admin use only

Date Received:		Admin Signature:	

Computer Science Tripos – Part II – Project Proposal

Fully In-Place Updates in Functional Programming

Ștefan - Alexandru Pițur, Robinson College

Originator: Dr. Jeremy Yallop

October 11, 2023

Project Supervisor: Dr. Jeremy Yallop

Director of Studies: Dr. Richard W Sharp

Project Overseers: Dr. David J Greaves & Dr. Damon Wischik

Introduction and description

Functional programming has lots of attributes that make it so popular among us, Computer Science enthusiasts. Unfortunately, whenever we are coding, we have a choice to make. We can either maintain a purely functional coding style, or aim for efficiency by using in-place updates, therefore abandoning the pursuit of data immutability and referential transparency. This problem is identified and tackled in a research paper[1], forming the foundation of my project.

My project will implement the ideas presented in the paper, aiming to validate the hypothesis of achieving optimised purely functional programming under specific conditions. Therefore, the end result of my project will be a programming language able to solve the previously mentioned problem, targeting OCaml's back-end, Flambda[2]. Hence, we may divide the project into two distinct parts: core and extensions.

The core of the project is focused on creating a programming language which enables functions to be executed fully in-place, where that is possible. The cited paper argues that this is achievable given that all of a function's owned parameters are unique and not shared. Furthermore, it performs this by dynamically checking using Perceus reference counting. The ingenuity of this project rests in being able to perform the check statically and executing the function fully in-place if the check is passed. In order to achieve this, we firstly need to build the compiler's front-end, mainly the lexer and parser, followed by connecting with OCaml's back-end in order to be able to execute arbitrary code. The details of how to implement the static check are not explored by the supporting research paper, but it suggests using a uniqueness type system, similar to the one present in Clean[3].

After building the core of the project, we will be able to design experiments for evaluating language's performance. It would be insightful to compare the in-place execution against its non- in-place variant from OCaml, Haskell, Scala and even C on relevant metrics, such as execution time, peak memory usage, number of allocations and more, using profilers. These experiments will consists of recording the specified execution metrics on different algorithms and data structures, such as Red-Black Tree, Finger Tree, Quicksort, Mergesort and mapping a function onto a generic tree.

For extensions, I plan on re-implementing the dynamic checking using Perceus reference counting, as done in the research paper. By doing so, we may evaluate the performance of the execution using the static check against the dynamic one on the designed experiments and expect to observe that the former approach is faster due to the overhead on the latter. Furthermore, this allows us to investigate if the static analysis validates as many functions to be executed in-place as the dynamic one does. Other extensions that I consider implementing include introducing OCaml's equivalent of references in the new language, Tail Recursion Modulo Cons optimisation (or even more generalised Tail Modulo Constructor), as well as enabling polymorphism for the typing system. It is worth mentioning that I do not expect to implement all of the extensions, due to the time limit imposed by the project, but will choose nearer the time which to focus on.

Starting point

As mentioned, my project expands on the findings of a research paper[1], providing an implementation of its theoretical details. Furthermore, open source projects such as Flambda[2] and Malfunction[4] could be used when building the compiler targeting the OCaml's backend.

Nevertheless, concepts presented in courses such as **Compiler Construction** and **Optimising Compilers** will be used as sources of information.

Resources required

Throughout the project, I will be using my personal computer (Mac Book Pro 2021, 10-Core M1 Max, 64GB RAM), for both code development and dissertation writing. In case of any unexpected failures, I plan on using another personal computer, without losing any of the current progress.

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

The project will always be saved in Cloud using the Git technology in a private repository on GitHub. Furthermore, copies of the project will periodically be saved offline, on USB drives. A similar approach will be taken with respect to the dissertation paper, continuously uploading it into GitHub, Google Drive and Overleaf, as well as regularly creating offline copies.

Success criteria

The project will be a success if the resulting functional programming language is able to correctly identify and run functions that can be executed fully in-place, using the static checking method, under specific conditions such as the ones hypothesised by the supporting research paper.

Apart from this, the project should be able provide information about well defined execution metrics and report significant findings when comparing the fully in-place execution against the non- in-place variants from other functional languages, but not limited only to.

Therefore, we may list the key-points of a successful core project into the following:

- Design a grammar and implement a lexer and a parser for the new simple functional programming language.
- Implement the static analyses to determine opportunities for in-place updates.
- Implement a compiler targeting Ocaml's back-end that takes advantage of the in-place update opportunities.
- Evaluate the performance of the generated code with and without in-place update optimisations.

Similarly, we may define key-points to be achieved by the proposed project extensions. As previously mentioned, it is hard to predict upfront how many of these extensions will be attempted due to the time constraint imposed by the project, but I can confirm that I will prioritise the first one over the others:

- Implement the dynamic analyses checker using Perceus reference counting.
  - Evaluate performance of the static checker against the dynamic one.
  - Provide insight into the relative strength of the static check compared the dynamic one.
- Implement the OCaml equivalent of references.
- Implement Tail Recursion Modulo Cons optimisation. Alternatively, Tail Modulo Constructor optimisation could be attempted as well.
- Enable polymorphism for the typing system.

Timetable

1. **Michaelmas weeks 1–2 (3^{rd} October - 16^{th} October)**

   Planned work

   - Setup GitHub repository, configure OCaml and IDEs for local development.
   - Further research into the supporting paper, gathering a deeper understanding of the semantics and ideologies presented.
   - Deciding on a language grammar and further research into pre-requisites for building the lexer and the parser.
   - Research on Flambda and Malfunction and how will my compiler be able to make calls to OCaml's back-end.

   Milestones

   - Submit draft and full proposal.
   - Having a fully configured initial development environment.

Deadlines

- 6th October - Full Draft Proposal submission
- 16th October - Final Proposal submission

2. **Michaelmas weeks 3–4 (17th October - 30th October)**

   Planned work

   - Settle on a language grammar.
   - Build the lexer for the compiler.

   Milestones

   - Have a fully functional lexer which is able to tokenize the project's new programming language.

3. **Michaelmas weeks 5–6 (31st October - 13th November)**

   Planned work

   - Fix any of the emerging issues with the lexer.
   - Build the parser for the compiler.
   - Further investigate how to connect with OCaml's back-end, Flambda.

   Milestones

   - Have a fully functional parser, therefore completing the compiler's front-end.

4. **Michaelmas weeks 7–8 (14th November - 27th November)**

   Planned work

   - Further research into ways of implementing the static analysis for determining whether a function can be executed in-place.
   - Initial steps of connecting the emerging programming language with OCaml's back-end.

   Milestones

   - Better understanding of possible approaches to performing the static check, along an initial implementation of some of the researched ideas.
   - Partial targeting to OCaml, enabling simple code to be executed.

5. **Michaelmas week 9 – Christmas Vacation (28st October - 11th December)**

   Planned work

   - Start implementing ideas found as part of the research.
   - Connect compiler's front-end with OCaml's back-end for arbitrary code execution.

Milestones

- Finalise the core part of the compiler.
- Create initial experiments and record execution metrics.

6. **Christmas Vacation (12th December - 15th January)**

   Planned work

   - Further research into the static check.
   - Finalise the static check implementation.
   - Fully enabling programs to be executed via OCaml, including optimised in-place execution for functions that pass the static check.
   - Implement part of the evaluation experiments, record execution metrics and prepare intermediate report for comparison with other programming languages.

   Milestones

   - Finalise the core part of the compiler.
   - Create initial experiments and record execution metrics.

7. **Lent weeks 1–2 (16th January - 29th January)**

   Planned work

   - Fix any of the emerging issues with the static check.
   - Finalise implementing all the experiments and perform recordings as planned.

   Milestones

   - Finalise the core part of the project.

8. **Lent weeks 3–4 (30th January - 12th February)**

   Planned work

   - Start researching ways of approaching the first extension, implementing the dynamic check presented in the support paper using Perceus reference counting.
   - Write and finish the Progress Report.
   - Prepare for Progress Report Presentations.

   Milestones

   - Have a deeper understanding of the requirements and design of the first extension and be prepared for implementation.

   Deadlines

   - 2nd February - Progress Report submission

9. **Lent weeks 5–6 (13th February - 26th February)**

   Planned work

   - Finalise first extension implementation, have the dynamic checker ready and connected with OCaml's back-end.
   - Perform experiments on the new functionality, comparing execution metrics against static checking version and other programming languages.

   Milestones

   - Finish the first extension.
   - Gather new execution metrics and prepare initial reports for *Evaluation*.
   - Begin working on the dissertation, starting with the *Introduction* chapter and producing an initial version of it.

10. **Lent weeks 7–8 (27th February - 11th March)**

    Planned work

    - Decide on the second extension to be implemented.
    - Research into the second extension.
    - Initial implementation of the second extension.
    - Begin writing the *Preparation* chapter of the dissertation.

    Milestones

    - Finish the second extension.
    - Finalising the *Introduction* chapter, in light of feedback.

11. **Lent week 9 - Easter Vacation (12th March - 25th March)**

    Planned work

    - Fix any of the emerging issues with the second extension.
    - Modify the *Preparation* chapter, in light of received feedback.
    - Begin writing the *Implementation* chapter.

    Milestones

    - Have a perfectly functioning second extension.
    - Completely finalise the *Preparation* chapter.

12. **Easter Vacation (26th March - 22nd April)**

    Planned work

    - Further implementation on remaining extensions.
    - Full completion of the *Implementation*, *Evaluation* and *Conclusions* chapters, incorporating any received feedback.

Milestones

- Finalise the *Implementation*, *Evaluation* and *Conclusions* chapters of the dissertation.

13. **Easter weeks 1–2 (23th April - 6th May)**

    Planned work

    - Fixing any last minute emerging problems with any extensions.
    - Integrating the changes into the correct chapters of the dissertation.
    - Prepare further documents required to be appended to the dissertation.
    - Perform any last changes to any of the chapters, in light of new feedback.

    Milestones

    - Finalise a penultimate version of the dissertation and have it ready for submission.

14. **Easter weeks 3 (7th May - 13th May)**

    Planned work

    - Perform any last changes by incorporating any feedback left.
    - Further proof reading followed by on-time submission.

    Deadlines

    - 10th May - Dissertation Deadline (electronic)
    - 10th May - Source Code Deadline (electronic copies)

References

[1] `https://www.microsoft.com/en-us/research/uploads/prod/2023/07/fip.pdf`

[2] `https://github.com/ocaml-flambda/flambda-backend`

[3] `https://wiki.clean.cs.ru.nl/Clean`

[4] `https://github.com/stedolan/malfunction`