Using Functor Categories to Generate Intermediate Code in Agda

Computer Science Tripos Part II Project Proposal

Jack Gao (yg410), Homerton College

Project Supervisors: Yulong Huang (yh419) and Yufeng Li (yl959)

Director of Studies: Dr. John Fawcett

October 2024

Introduction

In the paper “Using Functor Categories to Generate Intermediate Code”[1], John Reynold proposed a way of translating Algol-like languages (declarative programming languages with stores) to intermediate code, using their semantic interpretations in functor categories. By picking a suitable category of intermediate code, the interpretation process becomes compilation, which is “correct by construction”. Reynold concluded that he did not have a proper dependently typed language in hand, so his compiler is a partial function.

With the development of dependently typed languages and proof-based languages, we can implement his compiler as a total function and prove compiler correctness. In this project, I will use Agda to implement the compiler. Agda captures the source language’s intrinsic syntax with indexed families, which contains only well-typed terms. Therefore, it focuses on compiling the correct programs and rules out the ill-typed nonsensical input. The correctness of the compiler is supported by both theory and implementation. The theory of functor category semantics is based on the denotational semantics model, which offers a mathematically rigorous framework of understanding the meaning of programs. As a proof assistant language, Agda provides formal proofs with its strong type system, so the correctness of the compiler can be proved along its implementation. This also follows the increasing trend in formally proving the correctness of compilers and having verified compilers.

The project is to implement the compiler as described in [1] in Agda. The source language is an Algol-like language and the target language is an assembly-style intermediate language for a stack machine. The core part is compiling the basic instructions, variable declarations, and conditionals. John Reynold’s paper provides a detailed and precise definition of these components, which makes the implementation of the compiler very feasible.

Possible extensions include compiling advanced features such as open calls, subroutines and iterations, optimising the target code, writing proofs of correctness of the compiler, and exploring the equational theory of instruction sequences using the functor category semantics as a guide, which was not presented in the paper. Reynolds’ claims to build a functor category model, but at the end of Chapter 6 in [1] admits that he has actually done no such thing as the naturality laws do not hold unless instruction sequences are replaced by their image in some denotational model that is left unspecified. The extension could be exploring a reasonable equational theory that can be stated directly on the instruction sequences, under which the semantics actually becomes a true functor category model (i.e. naturality holds).

Substance and Structure

Theoretical foundation: functor category semantics based on the denotational semantics model

My implementation:
- Compiler: written in Agda, a dependent typed proof-based language
- Source language: an Algol-like language (a declarative programming language with stores)
- Target language: an assembly-style intermediate language for a stack machine

The specification of the source language and the target language will be given in the project.

Starting Point

I have not learned Agda prior to the project. I am aware that there is an online open-source tutorial for Agda[2]. In preparation for the project, I set up the environment for Agda on my laptop according to "Front Matter" in the book, but did not read anything from Part 1.

I do not have any other experience with compiler beyond Part IB Compiler Construction Course. I have not learned category theory and type theory prior to the lectures in Part II.

Evaluation

The current approach involves only type-checking the Agda program without executing it, as the performance is expected to be highly inefficient. At present, no effective solution to this issue has been proposed in academia. Consequently, evaluating the compiler's performance would not provide relevant insights.

Rigorously proving the correctness of a compiler involves intricate proofs rooted in equational theory. I propose implementing a series of unit tests on the generated intermediate code. The correctness can be demonstrated by comparing the output of the target code with the expected results.

Success Criteria

The project will be considered successful if the following conditions are met:
- A specification of an Algol-like language is given
- The Agda code for compiler successfully compiles given the specified Algol-like language
- A specification of an assembly-style intermediate language written
- The compiler output an language that satisfies the specification
- Basic instructions, variable declarations and conditionals can be compiled

Work Plan

Dates	Deliverable
24 Oct–8 Jan	Completion of Core Objectives
9 Jan–22 Jan	Progress Report and Presentation
23 Jan–5 Mar	Extensions, Proofs and Tests
6 Mar–14 May	Dissertation

Work packages breakdown:

1. **Michaelmas weeks 3–4**: 24 Oct–6 Nov

   - Learn Agda and be familiar with its different usage.

   Milestone: completing the exercises on Programming Language Foundations in Agda

2. **Michaelmas weeks 5–6**: 7 Nov–20 Nov

   - Implement basic instructions of the compiler.

   Milestone: code for basic instructions completed.

3. **Michaelmas weeks 7–8**: 21 Nov–4 Dec

   - Implement variable declarations of the compiler.

   Milestone: code for variable declarations completed.

4. **Christmas weeks 1–2**: 5 Dec–18 Dec

   Slack period:
   - Finish implementing basic instructions and variable declarations of the compiler.
   - Write proofs for basic instructions and variable declarations of the compiler if possible.

   Milestone: code for basic instructions and variable declarations completed

   Possible Milestone: proofs for basic instructions and variable declarations written.

5. **Christmas weeks 3–4**: 19 Dec–1 Jan

   - Implement conditionals of the compiler.

   Milestone: code for conditionals completed.

6. **Christmas weeks 5**: 2 Jan–8 Jan

   Slack period:
   - Check all core part functions.
   - Write unit tests for the generated intermediate code.
   - Write proofs for conditionals if possible.
   - Implement iterations (extension) and write proofs if possible.

Milestone: code for basic instructions, variable declarations and conditionals completed; all corresponding proofs written; unit tests written.

Possible Milestone: code for iterations completed; proofs for conditionals and iterations written.

7. **Christmas week 6–7**: 9 Jan–22 Jan

   - Write a progress report.
   - Prepare presentation slides.

   Milestone: progress report written; presentation slides prepared.

8. **Lent weeks 1–2**: 23 Jan–5 Feb

   - Rehearse the presentation.
   - Write unit tests for the generated intermediate code.
   - Check the feasibility of extensions.
   - Work on extensions.

   *Milestone: progress report submitted (**Due 7 Feb**); presentation prepared and rehearsed with supervisor; partial result of extensions; unit tests written.*

9. **Lent weeks 3–4**: 6 Feb–19 Feb

   - Give presentation.
   - Work on extensions.
   - Show correctness of translation by proving it satisfies the necessary properties.

   Milestone: presentation given; partial result of extensions.

   Possible Milestone: Proofs for extensions written.

10. **Lent weeks 5–6**: 20 Feb–5 Mar

    Slack period:
    - Complete the extensions.
    - Finish all proofs of the compiler.
    - Finish all unit tests of the generated intermediate code.

    Milestone: extensions completed; all unit tests written.

    Possible Milestone: Proofs of all components of the compiler written.

11. **Lent weeks 7–8**: 6 Mar–19 Mar

    - Draft introduction and preparation chapter of the dissertation and get feedback from supervisors.

    Milestone: introduction and preparation chapter written and checked.

12. **Easter vacation weeks 1–2**: 20 Mar–2 Apr

    - Draft implementation chapter and get feedback from supervisors.

    Milestone: implementation chapter written and checked.

13. **Easter vacation weeks 3–4**: 3 Apr–16 Apr

    - Draft evaluation and conclusion chapter and get feedback from supervisors.

    Milestone: evaluation and conclusion chapter written and checked.

14. **Easter vacation weeks 5–6**: 17 Apr–30 Apr

    Slack period:
    - adjust dissertation based on feedback from supervisors.

    Milestone: dissertation completed.

15. **Easter weeks 1–2**: 1 May–14 May

    - Final proof-reading and submit dissertation.

    *Milestone: dissertation and source code submitted. (**Due 16 May**)*

Resource Declaration

I will use my personal laptop for this project. My laptop specifications are:
- Lenovo Legion Y7000P IRH8
- CPU: 13th Gen Intel(R) Core(TM) i7-13700H 2.40GHz
- RAM: 16 GB
- SSD: 1TB + 2TB
- OS: Windows 11 Home

Windows Subsystem for Linux is installed on the machine. The version is Ubuntu 22.04 LTS.

Should this machine fail, I will continue my work on my other laptop. I will use Git for version control and daily work will push to a GitHub repository.

The project requires an Agda compiler, which is open-source and freely available online. I have already installed and tested it on my laptop.

References

[1] J. C. Reynolds, "Using Functor Categories to Generate Intermediate Code.," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Association for Computing Machinery, 1995. doi: 10.1145/199448.199452.

[2] Philip Wadler, Wen Kokke, and Jeremy G. Siek, *Programming Language Foundations in Agda.* 2022. [Online]. Available: https://plfa.inf.ed.ac.uk/22.08/