Jack Gao

Using functor categories to generate intermediate code with Agda

Computer Science Tripos - Part II Dissertation

Homerton College

May 6, 2025

Declaration of Originality

I, the candidate for Part II of the Computer Science Tripos with Blind Grading Number 2330G, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Date: May 6, 2025

Proforma

Candidate Number:	2330G
Title of Project:	Using functor categories to generate intermediate code with Agda
Examination	Computer Science Tripos - Part II - 2025
Word-Count:	[wordcount]
Code line count:	[linecount]
Project Originator:	Yulong Huang
Project Supervisor:	Yulong Huang and Yufeng Li

Original Aims of the Project

Work Completed

Special Difficulties

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Language choice: Agda's advantages 2

1.3 Contributions 3

2 Preparation **4**

2.1 Starting Point 5

2.2 Category theory 5

2.2.1 Category 5

2.2.2 Isomorphism 7

2.2.3 Cartesian closed category 8

2.2.4 Functor 10

2.2.5 Natural transformation 10

2.2.6 Functor category 10

2.2.7 Presheaf category 11

2.2.8 Yoneda lemma 11

2.2.9 Cartesian closed structure in presheaf categories 11

2.3 Agda 12

2.3.1 Basic datatypes and pattern matching 12

2.3.2 Dependent Types 12

2.3.3 Curry-Howard-Lambek correspondence 13

2.3.4 Equality, congruence and substitution 13

2.3.5 Standard library 14

2.3.6 Interactive programming with holes 15

2.4 Requirement Analysis 16

3 Implementation **17**

3.1 Tools used 18

3.2 Repository Overview . . . 18
3.2.1 Directory structure . . . 18
3.2.2 File dependencies and descriptions . . . 18
3.3 Source Language . . . 19
3.3.1 Types . . . 19
3.3.2 Contexts, variables and the lookup judgement . . . 20
3.3.3 Terms and the typing judgement . . . 21
3.3.4 Operational Semantics . . . 22
3.4 Target Language . . . 22
3.4.1 Stack descriptor . . . 23
3.4.2 Grammar . . . 25
3.5 Customised Library . . . 26
3.5.1 Natural numbers, integers and addition . . . 27
3.5.2 Equality, congruence and substitution . . . 27
3.5.3 Order of natural numbers . . . 27
3.5.4 Properties of order and addition . . . 28
3.5.5 Subtraction . . . 29
3.5.6 More properties of natural numbers . . . 32
3.6 Compiler . . . 32

4 Evaluation **33**
4.1 A demonstrative example in Agda . . . 33
4.2 Success criteria . . . 33

5 Conclusion **34**
5.1 Results . . . 34
5.2 Overview . . . 34
5.3 Lessons learned . . . 34

Chapter 1

Introduction

Contents

1.1 Motivation 2

1.2 Language choice: Agda's advantages 2

1.3 Contributions 3

Programming languages act as a bridge between human thought and machine execution. They exist in two complementary realms: the human realm, where intent is expressed through abstractions like variables and functions, and the machine realm, where low-level instructions are executed on hardware.

Denotational semantics is related to the human realms. It provides a theoretical framework for defining the meaning of programming languages by interpreting them into mathematical objects. By formalising the denotational semantics of a programming language, we ensure that its abstraction align with human intuition.

Compilers are related to the machine realms. They are programs that translate high-level code into executable instructions, resolving abstractions into concrete operations like memory allocation and register management. Compilers are essential for turning human-written code into physical computation.

This project explores how the two process can be deeply connected by demonstrating how denotational semantics can directly generate a compiler. Simple Typed Lambda Calculus (STLC) is a well-studied programming language that serves as a foundation for many modern languages. It has denotational semantics in cartesian closed categories, shown by Lambek [1]. As a ccc, presheaf categories over store locations can be used to model STLC with stores, as shown by by Reynolds [2] and Oles [3] [4]. Instead, Reynolds [5] presented a denotational semantics of STLC with stores in the form of a presheaf category over compiler states. By interpresting the source language into the presheaf category over stack descriptors, where objects of the category represent instruction sequences parameterised by stack layouts, the semantic model directly yields a compiler.

I implement this compiler for STLC with stores in a dependently typed programming language, Agda [6]. The compiler is a functor (structure-preserving map) from the source language's semantic domain to the target language's instruction set. The implementation is verified with Agda's type system, which ensures that the generated code is well-typed and adheres to the semantics of both the source and target languages.

This work both validates and refines Reynolds' theory, offering a concrete example of how category-theoretic semantics can generate intermediate code. The project also serves as a practical demonstration of the power of dependently typed programming languages can mechanise the link between theory and practice.

1.1 Motivation

This project is motivated and guided by the following:

Motivation I. Implementation of the compiler

Reynolds concluded that he did not have a proper dependently typed programming language in hand, so his compiler remained a partial function. I aim to provide a computer implementation of this theoretical framework in a dependently typed programming language.

Motivation II. Formal verification of the compiler

Reynolds' work presented detailed definitions of denotational semantics of STLC with stores, which are complicated and error-prone. The rise of verified compilers including CompCert [7], CakeML [8] reflects a broader trend toward trustworthy systems, where correctness proofs replace testing for critical guarantees. I aim to provide a formalisation of the definition in a proof assistant to verify the correctness of the given denotational semantics.

1.2 Language choice: Agda's advantages

Agda [6] is a dependently typed programming language and proof assistant. Agda captures the source language's intrinsic sytax with indexed families, which contains only well-typed terms. Therefore, it focuses on the correct programs and rules out the ill-typed nonsensical inputs.

Dependently typed languages provide a natural framework for expressing functor categories is proven both theoretically and practically. There have been dependent-type-theoretic model of categories [9], and it has been shown that functor categories arise naturally as dependent function types [10]. A formalisation of Category Theory, including cartesian closed categories, functors and presheaves has been developed in Agda by Hu and Caratte [11]. Other proof assistants, such as Isabelle/Hol, does not have a dependently typed language structure, and thus cannot express the functor categories as naturally as Agda.

Compared to other dependently typed languages, Agda is more balanced in terms of programming and proving. Its with-abstraction and rewrite construction allow for a more flexible and powerful way to define and manipulate terms. The with-abstraction allows us to inspect intermediate values in a term, which gives a refined view of a function's argument. The rewrite construction allows us to define new terms by expanding existing terms, which avoids rewriting proofs of similar structures.

Agda also provides an interactive environment for writing and verifying programs, which will be further discussed in §2.3.6.

1.3 Contributions

Addressed the three motivations presented in §1.1 and contributed to the following:

> **Motivation I. Implementation of the compiler**
>
> I implemented the compiler from the source language to the target language in Agda.
>
> **Motivation II. Formal verification of the compiler**
>
> I formalised the terms in the source language and target language in Agda, and proved that the compiler is a functor from the source language to the target language.

Chapter 2

Preparation

Contents

2.1 **Starting Point** **5**

2.2 **Category theory** **5**

2.2.1 Category 5

2.2.2 Isomorphism 7

2.2.3 Cartesian closed category 8

2.2.4 Functor 10

2.2.5 Natural transformation 10

2.2.6 Functor category 10

2.2.7 Presheaf category 11

2.2.8 Yoneda lemma 11

2.2.9 Cartesian closed structure in presheaf categories 11

2.3 **Agda** **12**

2.3.1 Basic datatypes and pattern matching 12

2.3.2 Dependent Types 12

2.3.3 Curry-Howard-Lambek correspondence 13

2.3.4 Equality, congruence and substitution 13

2.3.5 Standard library 14

2.3.6 Interactive programming with holes 15

2.4 **Requirement Analysis** **16**

For implementing the compiler, I need to understand the theoretical background of presheaves and functor categories that are used as the denotational semantics of the source language, and I need to understand Agda's dependent types to correctly express the dependent function space of presheaf exponentials. This chapter begins with my starting point and an introduction to category theory, followed by a brief overview of the dependently typed programming language Agda, and requirements analysis of the compiler.

2.1 Starting Point

Prior to this project, I had no experience with Agda. Although I was aware of the open-source online tutorial *Programming Language Foundations in Agda* (PLFA) [12], my preparation was limited to setting up the Agda environment on my laptop by following the “Front Matter” section of the tutorial.

I did not have any other experience with compiler beyond Part IB Compiler Construction Course. I had no prior exposure to category theory and type theory before the Part II lectures.

2.2 Category theory

Category theory provides a high-level abstraction from which we can reason about the structure of mathematical objects and their relationships. It provides us a “bird’s eye view” of the mathematics which enable us to spot patterns that are difficult to see in the details [13]. More specifically, it provides a “purer” view of functions that is not derived from sets [14]. Compared to set theory which is “element-oriented”, category theory is “function-oriented” and “morphism-oriented”. We understand structures not via elements but by how they transform into each other.

Notation in category theory is similar to that in set theory and type theory. For example, $A, B, \ldots$ are used to denote objects, sets, or types, and arrows are used to denote morphisms or functions (e.g. $A \to B$). There is a deeper connection between category theory and type theory, which will be discussed in §2.3.3.

The following is a brief introduction to the basic concepts of category theory, which is based on the work of Leinster [13], Riehl [15], and the lecture notes of Part II Category Theory by Andrew Pitts and Marcelo Fiore [16].

2.2.1 Category

Definition 1 (Category). A *category* $\mathcal{C}$ is specified by the following:

- a collection of objects $\mathbf{obj}(\mathcal{C})$, whose elements are called $\mathcal{C}$-objects;
- for each $X, Y \in \mathbf{obj}(\mathcal{C})$, a collection of morphisms $\mathcal{C}(X, Y)$, whose elements are called $\mathcal{C}$-morphisms from X to Y (e.g. $f : X \to Y$);
- for each $X \in \mathbf{obj}(\mathcal{C})$, an element $\mathbf{id}_X \in \mathcal{C}(X, X)$ called the identity morphism on X;
- for each $X, Y, Z \in \mathbf{obj}(\mathcal{C})$, a function

  $$\begin{aligned} \mathcal{C}(X, Y) \times \mathcal{C}(Y, Z) &\to \mathcal{C}(X, Z) \\ (f, g) &\mapsto g \circ f \end{aligned}$$

  called the composition of morphisms;

satisfying the following properties:

- **(Unit Laws)** for all $X, Y \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$, we have:

$$\mathbf{id}_Y \circ f = f = f \circ \mathbf{id}_X \tag{2.1}$$

- **(Associativity Law)** for all $X, Y, Z, W \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$, $g \in \mathcal{C}(Y, Z)$, $h \in \mathcal{C}(Z, W)$, we have:

$$h \circ (g \circ f) = (h \circ g) \circ f \tag{2.2}$$

Example 1 ($\mathcal{S}et$). An example of category is the category of sets, denoted as $\mathcal{S}et$, specifed by:

- the objects of $\mathcal{S}et$ are a fixed universe of sets;
- for each $X, Y \in \mathbf{obj}(\mathcal{S}et)$, the morphisms from $X \to Y$ in $\mathcal{S}et$ are the functions $X \to Y$;
- the identity morphism on X in $\mathcal{S}et$ is the identity function on X;
- the composition of morphisms in $\mathcal{S}et$ is defined as the composition of functions.

Associativity law and unit laws are satisfied in $\mathcal{S}et$, since the composition of functions is associative and the identity function compose with any function is the function itself.

Example 2 ($\mathcal{P}oset$). Another example of category is the category of posets, denoted as $\mathcal{P}oset$, specified by:

- the objects of $\mathcal{P}oset$ are a fixed universe of posets, which is a set P with a binary relation $\subseteq$ that is
  - reflexive: $\forall x \in P, x \subseteq x$;
  - transitive: $\forall x, y, z \in P, x \subseteq y \land y \subseteq z \implies x \subseteq z$;
  - antisymmetric: $\forall x, y \in P, x \subseteq y \land y \subseteq x \implies x = y$.
- for each $X, Y \in \mathbf{obj}(\mathcal{P}oset)$, the morphisms from X to Y in $\mathcal{P}oset$ are the monotone (order-preserving) functions from X to Y;
- the identity morphism on X in $\mathcal{P}oset$ is the identity function on X;
- the composition of morphisms in $\mathcal{P}oset$ is defined as the composition of functions.

Associativity law and unit laws are satisfied in $\mathcal{P}oset$, since the set of monotone functions contains identity functions and is closed under composition.

$\mathcal{P}oset$ is used later for stack descriptors in the denotational semantics of the source language.

Opposite category

The idea of opposite category is that if we have a category $\mathcal{C}$, we can reverse the direction of all morphisms in $\mathcal{C}$ to obtain a new category $\mathcal{C}^{\mathbf{op}}$.

Definition 2 (Opposite category). Given a category $\mathcal{C}$, its *opposite category* $\mathcal{C}^{\mathbf{op}}$ is specified by:

- the objects of $\mathcal{C}^{\mathbf{op}}$ are the same as those of $\mathcal{C}$;

- for each $X, Y \in \mathbf{obj}(\mathcal{C})$, the morphisms from X to Y in $\mathcal{C}^{\mathbf{op}}$ are the morphisms from Y to X in $\mathcal{C}$;
- the identity morphism on X in $\mathcal{C}^{\mathbf{op}}$ is the identity morphism on X in $\mathcal{C}$;
- the composition of morphisms in $\mathcal{C}^{\mathbf{op}}$ is defined as the composition of morphisms in $\mathcal{C}$.

The notation of commutative diagram is widely used in category theory as a convenient visual representation of the relationships between objects and morphisms in a category.

Commutative diagrams

A *diagram* in a category $\mathcal{C}$ is a directed graph whose vertices are $\mathcal{C}$-objects and whose edges are $\mathcal{C}$-morphisms.

A diagram is *commutative* (or *commutes*) if any two finite paths in the graph between any two vertices X and Y in the diagram determine the equal morphism $f \in \mathcal{C}(X, Y)$ under the composition of morphisms.

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\mathbf{id}_X"] \arrow[dr, "f"] & Y \arrow[d, "\mathbf{id}_Y"] \\
X \arrow[r, "f"'] & Y
\end{tikzcd}$$

Figure 2.1: Commutative diagram for Unit Laws

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g \circ f"] & Y \arrow[d, "h \circ g"] \arrow[dl, "g"] \\
Z \arrow[r, "h"'] & W
\end{tikzcd}$$

Figure 2.2: Commutative diagram for Associativity Law

As examples of commutative diagrams, Figure 2.2 and Figure 2.1 are commutative diagrams for the unit laws and associativity law respectively.

2.2.2 Isomorphism

Definition 3 (Isomorphism). Given a category $\mathcal{C}$, a C-morphism $f : X \to Y$ is called an *isomorphism* if there exists a C-morphism $g : Y \to X$ such that the following diagram commutes:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[dr, "\mathbf{id}_X"'] & Y \arrow[d, "g"] \arrow[dr, "\mathbf{id}_Y"] & \\
& X \arrow[r, "f"'] & Y
\end{tikzcd} \tag{2.3}$$

In other words, f is an isomorphism if there exists a morphism g such that $g \circ f = \mathbf{id}_X$ and $f \circ g = \mathbf{id}_Y$.

The morphism g is uniquely determined by f and is called the *inverse* of f, denoted as f^{-1}.

Given two objects X and Y in a category $\mathcal{C}$, if there exists an isomorphism from X to Y, we say that X and Y are *isomorphic* in $\mathcal{C}$ and write $X \cong Y$.

2.2.3 Cartesian closed category

Terminal object

Definition 4 (Terminal object). Given a category $\mathcal{C}$, an object $T \in \mathbf{obj}(\mathcal{C})$ is called a *terminal object* if for all $X \in \mathbf{obj}(\mathcal{C})$, there exists a unique $\mathbf{C}$-morphism $f : X \to T$.

Terminal objects are unique up to isomorphism. In other words, we have

- if T and T' are both terminal objects in $\mathcal{C}$, then there exists a unique isomorphism $f : T \to T'$.
- if T is a terminal object in $\mathcal{C}$ and $T \cong T'$, then T' is also a terminal object in $\mathcal{C}$.

Example 3 (Terminal object in $\mathcal{S}et$). $\mathcal{S}et$ has a terminal object $\{*\}$, which is an arbitrary singleton set containing a single element $*$.

For any set X, there exists a unique function $f : X \to \{*\}$ that maps every element of X to the single element $*$ in $\{*\}$.

There is a unique isomorphism $f : \{*\} \to \{\cdot\}$ for any two singleton sets $\{*\}$ and $\{\cdot\}$, which is $f(*) = \cdot$.

Binary product

Definition 5 (Binary product). Given a category $\mathcal{C}$, the *binary product* of two objects X and Y in $\mathcal{C}$ is specified by

- a $\mathcal{C}$-object $X \times Y$;
- two $\mathcal{C}$-morphisms $\pi_1 : X \times Y \to X$ and $\pi_2 : X \times Y \to Y$ called the *projections* of $X \times Y$;

such that for all $Z \in \mathbf{obj}(\mathcal{C})$ and morphisms $f : Z \to X$ and $g : Z \to Y$, there exists a unique morphism $u : Z \to X \times Y$ such that the following diagram commutes in $\mathcal{C}$:

$$\begin{tikzcd}
 & Z \arrow[ld, "f"'] \arrow[d, dashed, "u" description] \arrow[rd, "g"] & \\
X & X \times Y \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & Y
\end{tikzcd} \tag{2.4}$$

The unique morphism u is written as

$$\{f, g\} : Z \to X \times Y$$

where $f = \pi_1 \circ u$ and $g = \pi_2 \circ u$.

It can be shown that the binary product is unique up to (unique) isomorphism.

Example 4 (Binary product in $\mathcal{S}et$). The binary product of two sets X and Y in $\mathcal{S}et$ is the Cartesian product $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$, where (x, y) are ordered pairs.

We have the following projections:

- $\pi_1 : X \times Y \to X$ is defined as $\pi_1(x, y) = x$ for all $(x, y) \in X \times Y$;
- $\pi_2 : X \times Y \to Y$ is defined as $\pi_2(x, y) = y$ for all $(x, y) \in X \times Y$.

For any set Z, for any functions $f : Z \to X$ and $g : Z \to Y$, the unique morphism $u : Z \to X \times Y$ is defined as:

$$u(z) = (f(z), g(z)) \text{ for all } z \in Z.$$

Exponential

Definition 6 (Exponential). Given a category $\mathcal{C}$ with binary products, the *exponential* of two objects X and Y in $\mathcal{C}$ is specified by

- a $\mathcal{C}$-object $X \Rightarrow Y$;
- a $\mathcal{C}$-morphism $\mathbf{app} : (X \Rightarrow Y) \times X \to Y$ called the *application* of $X \Rightarrow Y$;

such that for all $Z \in \mathbf{obj}(\mathcal{C})$ and morphisms $f : Z \times X \to Y$, there exists a unique morphism $u : Z \to X \Rightarrow Y$ such that the following diagram commutes in $\mathcal{C}$:

$$\begin{tikzcd}
(X \Rightarrow Y) \times X \arrow[r, "\mathbf{app}"] & Y \\
Z \times X \arrow[u, dashed, "u \times \mathbf{id}_X"] \arrow[ur, "f"'] &
\end{tikzcd} \tag{2.5}$$

We write $\mathbf{cur}f$ for the unique morphism u such that $f = \mathbf{app} \circ (\mathbf{cur}f \times \mathbf{id}_X)$, where $\mathbf{cur}f$ is called the *currying* of f.

It can be shown that the exponential is unique up to (unique) isomorphism.

Example 5 (Exponential in $\mathcal{S}et$). The exponential of two sets X and Y in $\mathcal{S}et$ is the set of all functions from X to Y.

Function application gives the morphism $\mathbf{app} : (X \Rightarrow Y) \times X \to Y$ as $\mathbf{app}(f, x) = f(x)$ for all $f \in X \Rightarrow Y$ and $x \in X$.

The currying operation transform a function $f : Z \times X \to Y$ into a function $\mathbf{cur}f : Z \to (X \Rightarrow Y)$, which is defined as $\mathbf{cur}f(z) = \lambda x.f(z, x)$ for all $z \in Z$ and $x \in X$.

Definition 7 (Cartesian closed category). A category $\mathcal{C}$ is called a *Cartesian closed category* (ccc) if it has a terminal object, binary products and exponentials of any two objects.

2.2.4 Functor

Definition 8 (Functor). Given two categories $\mathcal{C}$ and $\mathcal{D}$, a *functor* $F : \mathcal{C} \to \mathcal{D}$ is specified by:

- a function

$$\begin{aligned} \mathbf{obj}(\mathcal{C}) &\to \mathbf{obj}(\mathcal{D}) \\ X &\mapsto F(X) \end{aligned}$$

- for each $X, Y \in \mathbf{obj}(\mathcal{C})$, a function

$$\begin{aligned} \mathcal{C}(X,Y) &\to \mathcal{D}(F(X), F(Y)) \\ f &\mapsto F(f) \end{aligned}$$

satisfying the following properties:

- for all $X, Y \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X,Y)$, we have: $F(\mathbf{id}_X) = \mathbf{id}_{F(X)}$
- for all $X, Y, Z \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X,Y)$, $g \in \mathcal{C}(Y,Z)$, we have: $F(g \circ f) = F(g) \circ F(f)$

2.2.5 Natural transformation

Definition 9 (Natural transformation). Given two categories $\mathcal{C}$ and $\mathcal{D}$, and two functors $F, G : \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\theta : F \to G$ is a family of morphisms $\theta_X \in \mathcal{D}(F(X), G(X))$ for each $X \in \mathbf{obj}(\mathcal{C})$ such that for all $X, Y \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X,Y)$, the following diagram

$$\begin{tikzcd} F(X) \arrow[r, "\theta_X"] \arrow[d, "F(f)"'] & G(X) \arrow[d, "G(f)"] \\ F(Y) \arrow[r, "\theta_Y"'] & G(Y) \end{tikzcd} \tag{2.6}$$

commutes in $\mathcal{D}$, i.e. $G(f) \circ \theta_X = \theta_Y \circ F(f)$

2.2.6 Functor category

Definition 10 (Functor category). Given two categories $\mathcal{C}$ and $\mathcal{D}$, the *functor category* $\mathcal{D}^{\mathcal{C}}$ is the category satisfying the following:

- the objects of $\mathcal{D}^{\mathcal{C}}$ are all functors $\mathcal{C} \to \mathcal{D}$;
- given two functors $F, G : \mathcal{C} \to \mathcal{D}$, the morphisms from F to G in $\mathcal{D}^{\mathcal{C}}$ are all natural transformations $\theta : F \to G$;
- composition and identity morphisms in $\mathcal{D}^{\mathcal{C}}$ are defined as follows:
  - the identity morphism $\mathbf{id}_F$ on F is defined as $\theta_X = \mathbf{id}_{F(X)}$ for all $X \in \mathbf{obj}(\mathcal{C})$;
  - the composition of two natural transformations $\theta : F \to G$ and $\phi : G \to H$ is defined as $(\phi \circ \theta)_X = \phi_X \circ \theta_X$ for all $X \in \mathbf{obj}(\mathcal{C})$.

2.2.7 Presheaf category

Definition 11 (Presheaf). Given a category $\mathcal{C}$, a *presheaf* on $\mathcal{C}$ is a functor $F : \mathcal{C}^{\mathbf{op}} \to \mathbf{Set}$. A presheaf is a contravariant functor, which means that it reverses the direction of morphisms. In other words, a presheaf is a functor that takes objects in $\mathcal{C}$ and assigns them sets, and takes morphisms in $\mathcal{C}$ and assigns them functions between the corresponding sets. The presheaf F is defined as follows:

- for each $X \in \mathbf{obj}(\mathcal{C})$, $F(X)$ is a set;
- for each $X, Y \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$, $F(f)$ is a function $F(Y) \to F(X)$.

Definition 12 (Presheaf category). Given a category $\mathcal{C}$, the *presheaf category* $\hat{\mathcal{C}}$ is the is the functor category $\mathcal{S}et^{\mathcal{C}^{\mathrm{op}}}$, which explicitly contains the following:

- the objects of $\hat{\mathcal{C}}$ are all presheaves on $\mathcal{C}$;
- given two presheaves $F, G : \mathcal{C}^{\mathbf{op}} \to \mathcal{S}et$, the morphisms from F to G in $\hat{\mathcal{C}}$ are all natural transformations $\theta : F \to G$.

2.2.8 Yoneda lemma

Definition 13 (Yoneda functor). Given a category $\mathcal{C}$, the *Yoneda functor* $\text{よ} : \mathcal{C} \to \hat{\mathcal{C}}$ is defined as follows:

- for each $X \in \mathbf{obj}(\mathcal{C})$, $\text{よ}(X)$ is the functor $\mathcal{C}^{\mathbf{op}} \to \mathcal{S}et$ defined as:

  $$\text{よ}(X)(Y) = \mathcal{C}(Y, X) \tag{2.7}$$

  for all $Y \in \mathbf{obj}(\mathcal{C})$;

- for each $X, Y \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$, $\text{よ}(f)$ is the morphism $\text{よ}(X) \to \text{よ}(Y)$ defined a natural transformation whose component at any given $Z \in \mathcal{C}^{\mathbf{op}}$ is given by:

  $$\begin{aligned} (\text{よ}(f))_Z : \mathcal{C}(Z, Y) &\to \mathcal{C}(Z, X) \\ g &\mapsto g \circ f \end{aligned} \tag{2.8}$$

  for all $Z \in \mathbf{obj}(\mathcal{C})$.

Theorem 1 (Yoneda lemma). Given a category $\mathcal{C}$, the *Yoneda lemma* states that for each object $X \in \mathbf{obj}(\mathcal{C})$ and any functor $F : \mathcal{C} \to \mathbf{Set}$, there is a natural isomorphism:

$$\hat{\mathcal{C}}(\text{よ}(X), F) \cong F(X) \tag{2.9}$$

2.2.9 Cartesian closed structure in presheaf categories

Proof 1 (Cartesian closed structure in presheaf categories). Given a small category $\mathcal{C}$, the presheaf category $\hat{\mathcal{C}}$ is a Cartesian closed category.

- The terminal object in $\hat{\mathcal{C}}$ is the constant functor $\mathbf{1} : \mathcal{C}^{\mathbf{op}} \to \mathcal{S}et$, given by

$$\begin{cases} \mathbf{1}(X) = \{*\} & \text{for all } X \in \mathbf{obj}(\mathcal{C}) \\ \mathbf{1}(f) = \mathbf{id}_{\{*\}} & \text{for all } f \in \mathcal{C}(X, Y) \end{cases} \tag{2.10}$$

- The binary product in $\hat{\mathcal{C}}$ is given by the product of functors, which is defined as follows:

$$\begin{aligned} (F \times G)(X) &= F(X) \times G(X) \\ (F \times G)(f) &= F(f) \times G(f) \end{aligned} \tag{2.11}$$

- The exponential in $\hat{\mathcal{C}}$ is derived from the Yoneda lemma,

$$\begin{aligned} G^F(X) &= \hat{\mathcal{C}}(\text{よ}(X) \times F, G) \\ G^F(f)(\theta) &= \theta \circ (\text{よ}(f) \times \mathbf{id}_F) \quad \text{for all } \theta \in \hat{\mathcal{C}}(\text{よ}(Y) \times F, G) \end{aligned} \tag{2.12}$$

2.3 Agda

2.3.1 Basic datatypes and pattern matching

We will go through a simple example in PLFA [12] to illustrate the basic datatypes and pattern matching in Agda.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero + n = n
(suc m) + n = suc (m + n)
```

Here we define a datatype ℕ for natural numebers. ℕ itself has a type Set, which is the type of all small types. The natural number is defined as a recursive datatype with two constructors: zero and suc, where zero is the base case and suc is the inductive case. The zero constructor has type ℕ, and the suc constructor has type ℕ → ℕ.

In order to define the plus function _ + _, we use pattern matching to match the input argument with the constructors of the datatype. The first case is the base case, where we match the input argument with zero. In this case, we return the second argument. The second case is the inductive case, where we match the input argument with suc. In this case, we return suc applied to the result of adding one to the second argument.

2.3.2 Dependent Types

A dependent type is a type that depends on a value. In terms of type judgement, simple types are in form of

$$x_1 : T_1, x_2 : T_2, \dots, x_n : T_n \vdash t(x_1, \dots, x_n) : T$$

In contrast, dependent types are in form of

$$x_1 : T_1, x_2 : T_2, \dots, x_n : T_n \vdash t(x_1, \dots, x_n) : T(x_1, \dots, x_n)$$

or more generally

$$x_1 : T_1, x_2 : T_2(x_1), \dots, x_n : T_n(x_1, \dots, x_{n-1}) \vdash t(x_1, \dots, x_n) : T(x_1, \dots, x_n)$$

A classical example of dependent types in Agda is the type of vectors, which are lists with a length.

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀{n : ℕ} (x : A) (xs : Vec A n) → Vec A (suc n)
```

Here we define a datatype Vec for vectors. The type of vectors is dependent on the length of the vector. With dependent types, we can encode properties directly into types and ensure that they are satisfied at compile time.

2.3.3 Curry-Howard-Lambek correspondence

Curry-Howard correspondence [17] establishes an isomorphism between logic and type theory, where propositions correspond to types and proofs correspond to terms. This correspondence forms the foundation of functional programming languages that can be used to implement proofs. Building upon this, Joachim Lambek showed that cartesian closed categories provide a natural semantic setting for the simply typed lambda calculus (STLC) [1]. The Curry-Howard-Lambek correspondence can be summarised as follows:

Logic	**Type theory**	**Category theory**
Proposition	Type	Object
Proof	Term	Morphism
Falsity	Empty type	Initial object
Truth	Unit type	Terminal object
Implication	Function type	Exponential
Conjunction	Product type	Product
Disjunction	Sum type	Coproduct

Table 2.1: Curry-Howard-Lambek correspondence

2.3.4 Equality, congruence and substitution

Equality here refers to the propositional equality. In Agda it is defined as

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

With equality as a type, whenever we want to prove that two terms are equal, we need to provide a witness of the equality. For example if we want to prove $x \equiv y$, we write out a term of type x = y, and then give its definition, which constructs a proof of the equality. A simple proof that directly uses the definition of equality is as follows:

```
sym : ∀ {A : Set} {x y : A} → x ≡ y → y ≡ x
sym refl = refl

trans : ∀ {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

Here we are able to prove the symmetry and transitivity of equality by using the definition of equality. Those two properties are very useful for later proofs.

We can also have more complex proof with congruence and substitution, which can also be directly derived from the definition of equality as follows:

```
cong : ∀ {A B : Set} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl

subst : ∀ {A : Set} {x y : A} (P : A → Set) → x ≡ y → P x → P y
subst P refl px = px
```

Congruence is a property of equality that states that if two terms are equal, then they can be substituted for each other in any context. Substitution is a property where we can get a new proof by replacing a term in a proof with another term that is equal to it.

Here is a simple example of how congruence can be used in a proof:

```
+-identity : ∀ (n : ℕ) → n + zero ≡ n
+-identity zero = refl
+-identity (suc n) = cong suc (+-identity n)
```

In this example, we want to prove that zero is the right identity of the plus function. We do an inductive proof by pattern matching on the first argument of the plus function.

In the base case, we have zero + zero $\equiv$ zero, which is trivially true by the definition of the plus function (zero is defined to be the left identity of the plus function).

In the inductive case, we need to show suc n + zero $\equiv$ suc n. We can use the definition of the plus function to rewrite the left-hand side as suc (n + zero). By the inductive hypothesis, we know that n + zero $\equiv n$, so we can substitute n for n + zero in the right-hand side by congruence, and we are done.

2.3.5 Standard library

The Agda standard library [18] is a collection of modules that provide a wide range of useful functions and types. It includes modules for basic data types, such as natural numbers, lists, and vectors. In my implementation, however, I defined most of the basic data types and functions from scratch for the following two reasons:

- I wanted to have a better understanding of the basic data types and functions, so I decided to implement them from scratch.
- I need a particular representation of minus operation that ensures the input is always valid. This is further explained in [to add]

Builtin pragma

With standard library, we can use actual numbers $1, 2, 3, \ldots$ instead of calling the constructor suc multiple times. This is a convenient feature for testing. Agda provides a way to link self-defined functions to the standard library with a BUILTIN pragma. This allows me to use a convenient syntax for numbers, while still using the self-defined functions in the implementation.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
{-# BUILTIN NATURAL ℕ #-}

_+_ : ℕ → ℕ → ℕ
zero + n = n
(suc m) + n = suc (m + n)
{-# BUILTIN NATPLUS _+_ #-}

1+1≡2 : 1 + 1 ≡ 2
1+1≡2 = refl
```

Note that in agda we can use unicode characters for terms, which makes the code more readable. For example, here we simply name the term 1+1≡2.

2.3.6 Interactive programming with holes

A feature of Agda is that it allows us to write programs with holes interatively. A hole is a placeholder for a term that we have not yet defined. By leaving holes in place of undefined terms, we can write programs that are incomplete but still type-check, and Agda's type checker will guide completion of the program: the context window displays inferred types of the holes, available variables and candidate terms with their types. Holes also supports case split and refinement, which means we can fill in a hole partially and split it into smaller holes.

In my implementation, I used holes to write the terms in the compiler. The complex terms are incrementally filled and verified by interatively refining partial implementations, reducing post-hoc debugging and ensuring robustness.

2.4 Requirement Analysis

To complete a compiler in Agda we need to implement the following components:

- A file source.agda that record the syntax of the source language.
- A file target.agda that record the syntax of the target language.
- A file compiler.agda that uses source.agda and target.agda as modules, and write functions whose input is a term in the source language and output is a term in the target language.

The success criteria of the project is that the compiler can compile a term in the source language to a term in the target language, and the output term is well-typed in the target language.

Chapter 3

Implementation

Contents

3.1 **Tools used** **18**

3.2 **Repository Overview** **18**

3.2.1 Directory structure 18

3.2.2 File dependencies and descriptions 18

3.3 **Source Language** **19**

3.3.1 Types 19

3.3.2 Contexts, variables and the lookup judgement 20

3.3.3 Terms and the typing judgement 21

3.3.4 Operational Semantics 22

3.4 **Target Language** **22**

3.4.1 Stack descriptor 23

3.4.2 Grammar 25

3.5 **Customised Library** **26**

3.5.1 Natural numbers, integers and addition 27

3.5.2 Equality, congruence and substitution 27

3.5.3 Order of natural numbers 27

3.5.4 Properties of order and addition 28

3.5.5 Subtraction 29

3.5.6 More properties of natural numbers 32

3.6 **Compiler** **32**

This chapter details the implementation of the compiler in Agda. Starting with a brief overview of the tools used, it then describes the directory structure and file dependencies. Then it discusses the specification of the source language, and the target language, which lead us to the problem of implementing a rigorous natural number subtraction and two different approaches. We will further see the difference between the two approaches and how they affect the implementation of the customised library and the implementation of the compiler.

3.1 Tools used

The project is implemented in Agda 2.7.0.1, which is the latest stable version at the time of writing.

Completing the project is an iterative process. I used Git [19] for version control, and work had been synchronised with a GitHub [20] repository for backup.

For the development environment, I tried both Emacs [21] and Visual Studio Code [22] with an agda-mode extension [23] on Windows Subsystem for Linux with Ubuntu [24] 22.04 LTS. I am more familiar with the snippet and syntax highlighting features of Visual Studio Code, so I used it for most of the development.

Code from the PLFA tutorial and Agda standard library [18] were used as references for implementation of the source language and the customised library. Apart from that, the code is written from scratch.

3.2 Repository Overview

3.2.1 Directory structure

The repository contains two independent implementations of the compiler, organised as follows:

```
.
├── fin
│   ├── lib.agda
│   ├── source.agda
│   ├── target.agda
│   └── compiler.agda
└── proof
    ├── lib.agda
    ├── source.agda
    ├── target.agda
    └── compiler.agda
```

3.2.2 File dependencies and descriptions

The dependency graph for each implementation is identical as follows:

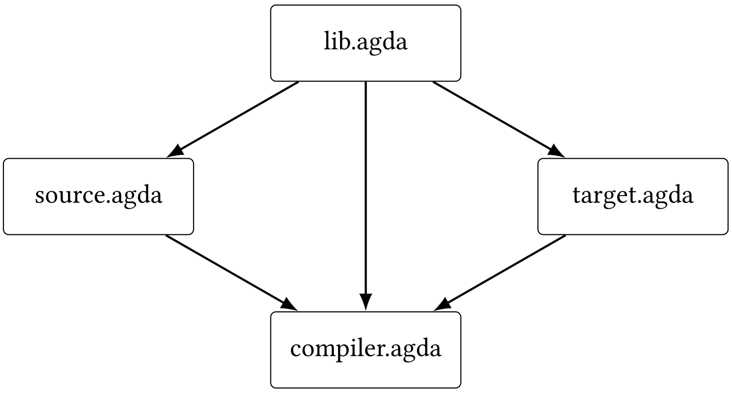

Figure 3.1: Dependency graph for the compiler

File	Description
lib.agda	Shared utilities (e.g. natural numbers and operations, equality, etc.)
source.agda	Source language syntax
target.agda	Target language (stack machine) instructions
compiler.agda	Compiler functor

Table 3.1: File descriptions

3.3 Source Language

A simply typed lambda calculus (STLC) is a typed lambda calculus with only one type constructor (→) that builds function types.

3.3.1 Types

In this dissertation, the source language is an STLC with the following primitive types:

- comm: the commands
- intexp: the integer expressions
- intacc: the integer acceptors
- intvar: the integer variables

and the set Θ of types is defined as follows:

$$\Theta := \mathsf{comm} \mid \mathsf{intexp} \mid \mathsf{intacc} \mid \mathsf{intvar} \mid \Theta \to \Theta$$

which corresponds with the following agda implementation:

```
data Type : Set where
  comm : Type
  intexp : Type
  intacc : Type
  intvar : Type
  _⇒_ : Type → Type → Type
```

Subtypes

We use the preorder $A \leq: B$ to denote the subtype relation A is a subtype of B, as it has the following properties:

- reflexivity: $A \leq: A$;
- transitivity: if $A \leq: B$ and $B \leq: C$, then $A \leq: C$.

The source language has the following subtype relations:

$$\mathsf{intvar} \leq: \mathsf{intexp} \qquad \mathsf{intvar} \leq: \mathsf{intacc}$$

and for function types we have the contravariant subtyping:

$$A' \leq: A \wedge B \leq: B' \Rightarrow A \to B \leq: A' \to B'$$

it is defined in Agda as follows:

```
data _≤:_ : Type → Type → Set where
  ≤:-refl : ∀{A} → A ≤: A
  ≤:-trans : ∀{A A’ A”} → A ≤: A’ → A’ ≤: A” → A ≤: A”
  ≤:-fn : ∀{A A’ B B’} → A’ ≤: A → B ≤: B’ → A ⇒ B ≤: A’ ⇒ B’

  var-≤:-exp : intvar ≤: intexp
  var-≤:-acc : intvar ≤: intacc
```

3.3.2 Contexts, variables and the lookup judgement

We define the context as a finite list of types. When we are looking up a variable, the type of the variable is either on the top of the context or in the tail of the context. The context and the lookup judgement are defined as follows:

```
-- Contexts
data Context : Set where
  · : Context
 _,_ : Context → Type → Context

-- Variables and the lookup judgement
data _∈_ : Type → Context → Set where
  Zero : ∀{Γ A} → A ∈ Γ , A
  Suc : ∀{Γ A B} → B ∈ Γ → B ∈ Γ , A
```

3.3.3 Terms and the typing judgement

The terms and the typing judgement are described in the following rules:

- For any variable we have

$$\frac{a : A \in \Gamma}{\Gamma \vdash a : A}\ \textsc{Var}$$

- For subtyping we have

$$\frac{\Gamma \vdash a : A \qquad A \leq: B}{\Gamma \vdash a : B}\ \textsc{Sub}$$

- For lambda abstraction we have

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.e : A \Rightarrow B}\ \textsc{Lambda} \qquad \frac{\Gamma \vdash e : A \Rightarrow B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\, e' : B}\ \textsc{App}$$

- For command we have

$$\frac{}{\Gamma \vdash \texttt{skip} : \mathsf{comm}}\ \text{Skip} \qquad \frac{\Gamma \vdash c_1 : \mathsf{comm} \qquad \Gamma \vdash c_2 : \mathsf{comm}}{\Gamma \vdash c_1 ; c_2 : \mathsf{comm}}\ \textsc{Seq} \qquad \frac{\Gamma, x : \mathsf{intvar} \vdash c : \mathsf{comm}}{\Gamma \vdash \texttt{new}\ x\ \texttt{in}\ c : \mathsf{comm}}\ \textsc{NewVar}$$

- For integer expression we have

$$\frac{z : \mathbb{Z}}{\Gamma \vdash \texttt{lit}\ \ z : \mathsf{intexp}}\ \textsc{Lit} \qquad \frac{\Gamma \vdash e : \mathsf{intexp}}{\Gamma \vdash -e : \mathsf{intexp}}\ \textsc{Neg} \qquad \frac{\Gamma \vdash e_1 : \mathsf{intexp} \qquad \Gamma \vdash e_2 : \mathsf{intexp}}{\Gamma \vdash e_1 + e_2 : \mathsf{intexp}}\ \textsc{Plus}$$

The corresponding Agda implementation is as follows, note that the Agda implementation does not include the name of the terms, but the names are included in the latex implementation for clarity:

```
data _⊢_ : Context → Type → Set where
  Var : ∀{Γ A} → A ∈ Γ → Γ ⊢ A

  -- subtyping
  Sub : ∀{Γ A B} → Γ ⊢ A → A ≤: B → Γ ⊢ B

  -- lambda function and application
  Lambda : ∀{Γ A B} → Γ , A ⊢ B → Γ ⊢ A ⇒ B
  App : ∀{Γ A B} → Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B

  -- command
  Skip : ∀{Γ} → Γ ⊢ comm
  Seq : ∀{Γ} → Γ ⊢ comm → Γ ⊢ comm → Γ ⊢ comm
  NewVar : ∀{Γ} → Γ , intvar ⊢ comm → Γ ⊢ comm
  Assign : ∀{Γ} → Γ ⊢ intacc → Γ ⊢ intexp → Γ ⊢ comm

  -- intexp
  Lit : ∀{Γ} → ℤ → Γ ⊢ intexp
  Neg : ∀{Γ} → Γ ⊢ intexp → Γ ⊢ intexp
  Plus : ∀{Γ} → Γ ⊢ intexp → Γ ⊢ intexp → Γ ⊢ intexp
```

Here the definition of the integers is defined in the customised library, which is defined in [section ref].

3.3.4 Operational Semantics

The operational semantics of the source language is defined with renaming, substitution and reduction rules. However, since the compiler itself does not require the operational semantics, the implementation is included in the appendix [link to appendix]. Operational semantics can be used to verify the correctness of the compiler in the future, but it is beyond the scope of this dissertation.

3.4 Target Language

The target language is an assembly-style intermediate language for a stack machine. It is defined with four stack-descriptor-indexed families of non-terminals:

- $\{\mathsf{L}_{sd}\}$: lefthand sides
- $\{\mathsf{S}_{sd}\}$: simple righthand sides
- $\{\mathsf{R}_{sd}\}$: righthand sides
- $\{\mathsf{I}_{sd}\}$: instruction sequences

3.4.1 Stack descriptor

The stack descriptor sd is defined as a pair of natural numbers $\langle f, d\rangle$, where f is the frame number and d is the displacement.

It is defined in Agda as follows:

```
record SD : Set where
  constructor ⟨_,_⟩
  field
    f : ℕ
    d : ℕ
```

This requires the definition of natural numbers, which is defined in §3.5.1.

For a stack descriptor sd, we can use SD.f sd to access the frame number and SD.d sd to access the displacement.

Order

The stack descriptor is ordered lexicographically with $\leq_s$ as follows:

$$\langle f, d\rangle \leq_s \langle f', d'\rangle \Leftrightarrow f < f' \vee (f = f' \wedge d \leq d')$$

It is defined in Agda as follows:

```
data _≤ₛ_ : SD → SD → Set where
  <-f : ∀ {f f' d d'} → f < f' → ⟨ f , d ⟩ ≤ₛ ⟨ f' , d' ⟩
  ≤-d : ∀ {f d d'} → d ≤ d' → ⟨ f , d ⟩ ≤ₛ ⟨ f , d' ⟩
```

This requires the definition of the order of natural numbers, which is defined in §3.5.3.

Addition and subtraction of stack descriptors

We define addition and subtraction of stack descriptors as follows:

$$\langle f, d\rangle \pm_s n = \langle f, d \pm n\rangle \text{ for } n \in \mathbb{N}$$

Addition of stack displacement is defined in Agda as follows:

```
_+ₛ_ : SD → ℕ → SD
⟨ f , d ⟩ +ₛ n = ⟨ f , d + n ⟩
```

For subtraction, we need to make sure that the subtraction is valid, i.e. the subtracted displacement is less than or equal to the displacement of the current stack descriptor. Here we need a rigorous definition of the subtraction of natural numbers, which is defined in §3.5.5

Properties of order and addition

We can use reflexivity and transitivity of the order $<$ and $\leq$ to prove that the order of stack descriptors $\leq_s$ is a preorder. The proof is as follows:

- Reflexivity: $sd \leq_s sd$ is trivially true since $f = f$ and $d \leq d$.
- Transitivity: if $\langle f, d \rangle \leq_s \langle f', d' \rangle$ and $\langle f', d' \rangle \leq_s \langle f'', d'' \rangle$, we do the following case split:
    - if $f < f'$, $f' = f''$ and $d \leq d'$, then $f < f''$;
    - if $f < f'$ and $f' < f''$, then $f < f''$ by transitivity of $<$;
    - if $f = f'$, $d \leq d'$ and $f' < f''$, then $f < f''$;
    - if $f = f'$, $d \leq d'$, $f' = f''$ and $d' \leq d''$, then $f = f''$ and $d \leq d''$ by transitivity of $\leq$. [1]

The proof is defined in Agda as follows:

```
≤ₛ-refl : ∀{sd : SD} → sd ≤ₛ sd
≤ₛ-refl {⟨ f , d ⟩} = ≤-d ≤-refl

≤ₛ-trans : ∀{sd sd’ sd” : SD} → sd ≤ₛ sd’ → sd’ ≤ₛ sd” → sd ≤ₛ sd”
≤ₛ-trans (<-f f<f’) (≤-d _) = <-f f<f’
≤ₛ-trans (<-f f<f’) (<-f f’<f”) = <-f (<-trans f<f’ f’<f”)
≤ₛ-trans (≤-d _) (<-f f’<f”) = <-f f’<f”
≤ₛ-trans (≤-d d≤d’) (≤-d d’≤d”) = ≤-d (≤-trans d≤d’ d’≤d”)
```

When a natural number is added to a stack descriptor, it is guaranteed that the new stack descriptor is not less than the old stack descriptor. This can be proved directly using the same properties of integer addition. The proof is defined in Agda as follows:

```
+ₛ→≤ₛ : ∀{sd : SD} → ∀{n : ℕ} → sd ≤ₛ sd +ₛ n
+ₛ→≤ₛ = ≤-d +→≤
```

Proof above requires the proof of the properties of natural numbers, which is defined in [section ref].

[1] Note that the properties of $=$ is not required in as in the Agda definition of $\leq_s$, we directly define f' to be f instead of defining a equivalence relation, thus equivalence can be checked directly via type checking.

3.4.2 Grammar

The grammar of the target language is defined as follows, given the current stack descriptor $sd = \langle f, d \rangle$, and new stack descriptor $sd' = \langle f', d' \rangle$:

$$
\begin{aligned}
\{\mathsf{L}_{sd}\} &::= sd^v \quad \text{when } sd^v \leq_s sd \\
&\quad \mid \texttt{sbrs} \\
\{\mathsf{S}_{sd}\} &::= \{\mathsf{L}_{sd}\} \\
&\quad \mid \texttt{lit}\,\{\text{integer}\} \\
\{\mathsf{R}_{sd}\} &::= \{\mathsf{S}_{sd}\} \\
&\quad \mid \{\text{unary operator}\}\,\{\mathsf{S}_{sd}\} \\
&\quad \mid \{\mathsf{S}_{sd}\}\,\{\text{binary operator}\}\,\{\mathsf{S}_{sd}\} \\
\{\mathsf{I}_{sd}\} &::= \texttt{stop} \\
&\quad \left.\begin{array}{l}
\mid \{\mathsf{L}_{sd+\delta}\} \leftarrow \{\mathsf{R}_{sd}\}\,[\delta]\ ;\ \{\mathsf{I}_{sd+\delta}\} \\
\mid \texttt{if}\ \ \{\mathsf{S}_{sd}\}\,\{\text{relational operator}\}\,\{\mathsf{S}_{sd}\}\,[\delta] \\
\quad \texttt{then}\ \ \{\mathsf{I}_{sd+\delta}\}\ \ \texttt{else}\ \ \{\mathsf{I}_{sd+\delta}\} \\
\mid \texttt{adjustdisp}[\delta]\ ;\ \{\mathsf{I}_{sd+\delta}\}
\end{array}\right\}\ \text{if } d + \delta \geq 0 \\
&\quad \mid \texttt{popto}\ \ sd'\ ;\ \{\mathsf{I}_{sd'}\} \quad \text{when } sd' \leq_s sd
\end{aligned}
$$

Operators

The operators are defined as follows:

$$
\begin{aligned}
\text{unary operator} &\in \{\texttt{UNeg}\} \\
\text{binary operator} &\in \{\texttt{BPlus}, \texttt{BMinus}, \texttt{BTimes}\} \\
\text{relational operator} &\in \{\texttt{RLeq}, \texttt{RLt}\}
\end{aligned}
$$

The operators are defined in Agda as follows:

```
data UnaryOp : Set where
  UNeg : UnaryOp

data BinaryOp : Set where
  BPlus : BinaryOp
  BMinus : BinaryOp
  BTimes : BinaryOp

data RelOp : Set where
  RLeq : RelOp
  RLt : RelOp
```

Lefthand sides, simple righthand sides and righthand sides

The lefthand sides, simple righthand sides and righthand sides are straightforward, and they are defined as follows in Agda:

```
-- Lefthand sides
data L (sd : SD) : Set where
  l-var : (sdᵛ : SD) → sdᵛ ≤ₛ sd → L sd
  l-sbrs : L sd

-- Simple righthand sides
data S (sd : SD) : Set where
  s-l : L sd → S sd
  s-lit : ℤ → S sd

-- Righthand sides
data R (sd : SD) : Set where
  r-s : S sd → R sd
  r-unary : UnaryOp → S sd → R sd
  r-binary : S sd → BinaryOp → S sd → R sd
```

Instruction sequences

In the instruction sequences, δ as a displacement can be either positive or negative. To make a rigorous definition, we define δ as a natural number, and treat the positive and negative displacements as two different instructions, using addition and subtraction of stack descriptors respectively. Since the definition of negative displacement involves the rigorous definition of subtraction, we will discuss the subtraction of natural numbers and correpsonding implementation in §3.5.5. The definition (except for the negative displacement) in Agda is as follows:

```
data I (sd : SD) : Set where
  stop : I sd
  assign-inc : (δ : ℕ) → L (sd +ₛ δ) → R sd → I (sd +ₛ δ) → I sd
  if-then-else-inc : (δ : ℕ) → S sd → RelOp → S sd
                          → I (sd +ₛ δ) → I (sd +ₛ δ) → I sd
  adjustdisp-inc : (δ : ℕ) → I (sd +ₛ δ) → I sd
  popto : (sd' : SD) → sd' ≤ₛ sd → I sd' → I sd
```

3.5 Customised Library

The customised library is a collection of types and functions centered around natural numbers, their operations and properties.

3.5.1 Natural numbers, integers and addition

The definition of natural numbers, integers and addition is standard, and it is defined as follows:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
{-# BUILTIN NATURAL ℕ #-}

data ℤ : Set where
  pos : ℕ → ℤ
  negsuc : ℕ → ℤ
{-# BUILTIN INTEGER          ℤ      #-}
{-# BUILTIN INTEGERPOS       pos    #-}
{-# BUILTIN INTEGERNEGSUC negsuc #-}

_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
{-# BUILTIN NATPLUS _+_ #-}
```

The use of BUILTIN is specifed in §2.3.5.

3.5.2 Equality, congruence and substitution

The definition of equality, congruence and substitution is similar to the one in §2.3.4.[2]

3.5.3 Order of natural numbers

The order of natural numbers we defined include $\leq$ and $<$, which are defined standardly as follows:

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n : ℕ} → zero ≤ n
  s≤s : ∀ {m n : ℕ} → m ≤ n → suc m ≤ suc n

data _<_ : ℕ → ℕ → Set where
  z<s : ∀ {n : ℕ} → zero < suc n
  s<s : ∀ {m n : ℕ} → m < n → suc m < suc n
```

[2] The definition of equality and related properties in the custom library differ slightly by using an indexed Set, which provides a more general formulation. See Appendex [corresponding link].

3.5.4 Properties of order and addition

Transitivity and reflexivity

The transitivity and reflexivity of the order $<$ and $\leq$ are defined as follows:

```
≤-refl : ∀ {n : ℕ} → n ≤ n
≤-refl {zero} = z≤n
≤-refl {suc n} = s≤s ≤-refl

≤-trans : ∀ {m n p : ℕ} → m ≤ n → n ≤ p → m ≤ p
≤-trans z≤n _ = z≤n
≤-trans (s≤s m≤n) (s≤s n≤p) = s≤s (≤-trans m≤n n≤p)

<-trans : ∀ {m n p : ℕ} → m < n → n < p → m < p
<-trans z<s (s<s _) = z<s
<-trans (s<s m<n) (s<s n<p) = s<s (<-trans m<n n<p)
```

Implication of order

We know that $\forall m, n \in \mathbb{N}$, if $m < n$, then we have $m + 1 \leq n$ and $m \leq n$. This is defined in Agda as follows:

```
<→s≤ : ∀ {m n : ℕ} → m < n → suc m ≤ n
<→s≤ (z<s) = s≤s z≤n
<→s≤ (s<s m<n) = s≤s (<→s≤ m<n)

<→≤ : ∀ {m n : ℕ} → m < n → m ≤ n
<→≤ m<n = ≤-trans n≤suc-n (<→s≤ m<n)
```

Totality of order

It is useful to case split the order of natural numbers, i.e. for any $m, n \in \mathbb{N}$, we have $m \leq n$ or $n \leq m$. This is defined in Agda by defining either case as an instance of the Total type, and then showing that for any $m, n \in \mathbb{N}$, we can construct a Total type by case-split and with-construct. The proof is as follows:

```
data Order : ℕ → ℕ → Set where
  leq : ∀ {m n : ℕ} → m ≤ n → Order m n
  geq : ∀ {m n : ℕ} → n ≤ m → Order m n

≤-compare : ∀ {m n : ℕ} → Order m n
≤-compare {zero} {n} = leq z≤n
≤-compare {suc m} {zero} = geq z≤n
≤-compare {suc m} {suc n} with ≤-compare {m} {n}
```

Other properties

We can show that when a number n is added to a natural number m, the result is greater than or equal to m. This is defined in Agda as follows:

```
+→≤ : ∀ {m n : ℕ} → m ≤ m + n
+→≤ {zero} {n} = z≤n
+→≤ {suc m} {n} = s≤s +→≤
```

3.5.5 Subtraction

In the standard library [18], the subtraction of natural numbers is defined as follows:[3]

```
_∸_ : ℕ → ℕ → ℕ
n ∸ zero = n
zero ∸ suc m = zero
suc n ∸ suc m = n ∸ m
{-# BUILTIN NATMINUS _∸_ #-}

0∸1≡0 : 0 ∸ 1 ≡ 0
0∸1≡0 = refl
```

Mathematically we have $n \dot{-} m = \max(0, n - m)$. This definition is valid and ensures that the result is a natural number. However, there are two problems with this definition for our implementation:

- The definition of subtraction is not rigorous. Input n should be guaranteed not to be less than m, and at any point in the program if there is a subtraction where $n < m$, the program should fail to type check instead of making 0 type check with it.
- We lose many numerical properties of natural numbers. For example, $n \dot{-} m + m = n$ or $n \dot{-} (n \dot{-} m) = m$ are not guaranteed to hold if we allow $n < m$. Those properties are very useful for the implementation of the compiler.

We need a more rigorous definition of subtraction that ensures the result is a natural number and preserves the numerical properties. The key is to have a proof that the first argument is greater than or equal to the second argument. There are two approaches to achieve this:

Explicit proof-passing approach

A straightforward approach is directly include a third argument in the subtraction function, which is a proof that the first argument is greater than or equal to the second argument. The definition in Agda is as follows:

```
_−_ : (n : ℕ) → (m : ℕ) → (p : m ≤ n) → ℕ
(n − zero) (z≤n) = n
(suc n − suc m) (s≤s m≤n) = (n − m) m≤n
```

[3] The definition in the standard library directly uses symbol −, but here for clarity we use $\dot{-}$ instead.

The corresponding implementation for subtraction of stack descriptors carries along the proof as follows:

```
_−s_ : (sd : SD) → (n : ℕ) → (n≤d : n ≤ SD.d sd) → SD
(⟨ f , d ⟩ −s n) n≤d = ⟨ f , (d − n) n≤d ⟩
```

In the implementation of instruction sequences, we need to carry along the proof as well. The implementation in Agda is as follows:

```
data I (sd : SD) : Set where
  assign-dec : (δ : ℕ) → (δ≤d : δ ≤ SD.d sd) → L ((sd −s δ) δ≤d)
                 → R sd → I ((sd −s δ) δ≤d) → I sd
  if-then-else-dec : (δ : ℕ) → (δ≤d : δ ≤ SD.d sd)
                       → S sd → RelOp → S sd
                       → I ((sd −s δ) δ≤d)
                       → I ((sd −s δ) δ≤d) → I sd
  adjustdisp-dec : (δ : ℕ) → (δ≤d : δ ≤ SD.d sd)
                     → I ((sd −s δ) δ≤d) → I sd
```

Fin-based approach

Instead of include the proof directly, we can define the type of the subtrahend as a type that depends on the minuend, where the type encodes the proof. There is a standard library in Agda on finite sets, where a type Fin is defined as a type that depends on a natural number n, where Fin n can be seen as the set of natural numbers less than n. The definition is as follows:

```
data Fin : ℕ → Set where
  fzero : ∀ {n} → Fin (suc n)
  fsuc : ∀ {n} → Fin n → Fin (suc n)
```

An intuitive explanation of the definition of Fin is as follows:

- Base case: when $n = 0$, there is no element less than 0, so we do not have any constructor that gives us an element of type Fin 0.
- Inductive case:

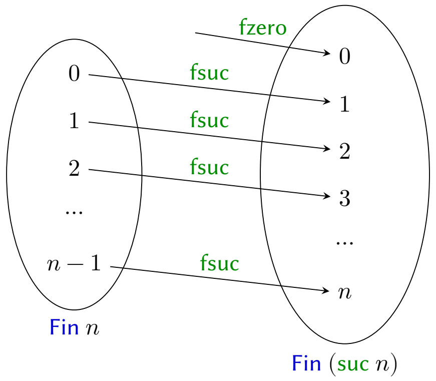

Figure 3.2: Inductive case of Fin

For subtraction, we only need to change the type of the subtrahend to Fin (suc n) compared to the implementation of addition. The definition in Agda is as follows:

```
_−_ : (m : ℕ) → Fin (suc m) → ℕ
m − fzero = m
suc m − fsuc n = m − n
```

The corresponding implementation for subtraction of stack descriptors and instruction sequences only requires a change of the type of the subtrahend as follows:

```
_−ₛ_ : (sd : SD) → Fin (suc (SD.d sd)) → SD
⟨ f , d ⟩ −ₛ n = ⟨ f , d − n ⟩

-- Instruction sequences
data I (sd : SD) : Set where
assign-dec : (δ : Fin (suc (SD.d sd))) → L (sd −ₛ δ) → R sd
             → I (sd −ₛ δ) → I sd
if-then-else-dec : (δ : Fin (suc (SD.d sd))) → S sd → RelOp → S sd
                   → I (sd −ₛ δ) → I (sd −ₛ δ) → I sd
adjustdisp-dec : (δ : Fin (suc (SD.d sd))) → I (sd −ₛ δ) → I sd
```

Comparison of the two approaches

The straightforward approach explicitly carries proofs in the type system, which is more transparent but verbose. The fin approach intially appears to be more elegant, as it encapsulates bounds check within a refined type. However, this elegance is superficial: constructing a term of type Fin n still requires the same underlying proof of boundedness, shifting complexity to auxiliary conversions.

To prove a property with the fin approach, we need the following auxiliary function:

```
≤→Fin : ∀ {m n} → m ≤ n → Fin (suc n)
≤→Fin z≤n = fzero
≤→Fin (s≤s p) = fsuc (≤→Fin p)
```

Here is a comparison of the two approaches representing the property suc $(n - m) \equiv$ suc $n - m$:

```
-- Explicit approach
−-suc : ∀ {n m} → {m≤n : m ≤ n}
          → suc ((n − m) m≤n) ≡ (suc n − m) (≤-trans m≤n n≤suc-n)

-- Fin-based approach
−-suc : ∀{n m} → {m≤n : m ≤ n}
      → suc (n − ≤→Fin m≤n) ≡ suc n − ≤→Fin (≤-trans m≤n n≤suc-n)
```

Figure 3.3: Comparison of explicit vs. Fin-based approaches

It is clear that the explicit proof-passing approach is more straightforward and easier to understand. The fin-based approach requires additional auxiliary functions and conversions, which makes the code more complex and harder to read.

Although the project completed both approaches, the following implementation of the compiler uses the explicit proof-passing approach due to its simplicity and clarity.

3.5.6 More properties of natural numbers

3.6 Compiler

Chapter 4

Evaluation

Contents

4.1 A demonstrative example in Agda 33

4.2 Success criteria 33

Chapter 3 describes the implementation of the source language, the target language and the compiler in Agda. This chapter evaluates the implementation with a demonstrative example in Agda, and discusses the success criteria of the project.

4.1 A demonstrative example in Agda

4.2 Success criteria

Chapter 5

Conclusion

Contents

5.1 Results 34
5.2 Overview 34
5.3 Lessons learned 34

5.1 Results

5.2 Overview

5.3 Lessons learned

Bibliography

[1] J. Lambek, "Cartesian closed categories and typed lambda- calculi," in *Proceedings of the Thirteenth Spring School of the LITP on Combinators and Functional Programming Languages*, Berlin, Heidelberg: Springer-Verlag, 1985, pp. 136–175, ISBN: 3540171843.

[2] J. C. Reynolds, "The essence of algol," in *ALGOL-like Languages, Volume 1.* USA: Birkhauser Boston Inc., 1997, pp. 67–88, ISBN: 0817638806.

[3] F. J. Oles, "A category-theoretic approach to the semantics of programming languages," AAI8301650, Ph.D. dissertation, USA, 1982.

[4] F. J. Oles, "Type algebras, functor categories, and block structure," *DAIMI Report Series*, vol. 12, no. 156, Jan. 1983. DOI: 10.7146/dpb.v12i156.7430. [Online]. Available: https://tidsskrift.dk/daimipb/article/view/7430.

[5] J. C. Reynolds, "Using functor categories to generate intermediate code," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 25–36, ISBN: 0897916921. DOI: 10.1145/199448.199452. [Online]. Available: https://doi.org/10.1145/199448.199452.

[6] U. Norell, "Dependently typed programming in agda," in *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, ser. TLDI '09, Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 1–2, ISBN: 9781605584201. DOI: 10.1145/1481861.1481862. [Online]. Available: https://doi.org/10.1145/1481861.1481862.

[7] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. [Online]. Available: https://doi.org/10.1145/1538788.1538814.

[8] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "Cakeml: A verified implementation of ml," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14, San Diego, California, USA: Association for Computing Machinery, 2014, pp. 179–191, ISBN: 9781450325448. DOI: 10.1145/2535838.2535841. [Online]. Available: https://doi.org/10.1145/2535838.2535841.

[9] S. Castellan, P. Clairambault, and P. Dybjer, *Categories with families: Unityped, simply typed, and dependently typed*, 2020. arXiv: 1904.00827 [cs.LO]. [Online]. Available: https://arxiv.org/abs/1904.00827.

[10] "Chapter 10 first order dependent type theory," in *Categorical logic and type theory*, ser. Studies in Logic and the Foundations of Mathematics, B. Jacobs, Ed., vol. 141, Elsevier, 1998, pp. 581–644. DOI: https://doi.org/10.1016/S0049-237X(98)80040-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0049237X98800402.

[11] J. Z. S. Hu and J. Carette, “Formalizing category theory in agda,” in *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2021, Virtual, Denmark: Association for Computing Machinery, 2021, pp. 327–342, ISBN: 9781450382991. DOI: 10.1145/3437992.3439922. [Online]. Available: https://doi.org/10.1145/3437992.3439922.

[12] P. Wadler, W. Kokke, and J. G. Siek, *Programming Language Foundations in Agda*. Aug. 2022. [Online]. Available: https://plfa.inf.ed.ac.uk/20.08/.

[13] T. Leinster, *Basic category theory*, 2016. arXiv: 1612.09375 [math.CT]. [Online]. Available: https://arxiv.org/abs/1612.09375.

[14] D. S. Scott, “Relating theories of the λ-calculus,” in *Relating Theories of the Lambda-Calculus: Dedicated to Professor H. B. Curry on the Occasion of His 80th Birthday*, Oxford: Springer, 1974, p. 406.

[15] E. Riehl, *Category Theory in Context*. Mineola, NY: Dover Publications, 2016. [Online]. Available: https://math.jhu.edu/~eriehl/context.pdf.

[16] A. Pitts and M. Fiore, *Category theory lecture notes*, Lecture notes, University of Cambridge, 2025. [Online]. Available: https://www.cl.cam.ac.uk/teaching/2425/CAT/CATLectureNotes.pdf.

[17] T. G. Griffin, “A formulae-as-type notion of control,” in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’90, San Francisco, California, USA: Association for Computing Machinery, 1989, pp. 47–58, ISBN: 0897913434. DOI: 10.1145/96709.96714. [Online]. Available: https://doi.org/10.1145/96709.96714.

[18] The Agda Community, *Agda standard library*, version 2.1.1, Release date: 2024-09-07, 2024. [Online]. Available: https://github.com/agda/agda-stdlib.

[19] *Git*, Accessed: 2025-04-15. [Online]. Available: https://git-scm.com/.

[20] *Github*, Accessed: 2025-04-15. [Online]. Available: https://github.com/.

[21] *Gnu emacs*, Accessed: 2025-04-15. [Online]. Available: https://www.gnu.org/software/emacs/.

[22] *Visual studio code*, Accessed: 2025-04-15. [Online]. Available: https://code.visualstudio.com/.

[23] T.-G. LUA, *Agda-mode*, Accessed: 2025-04-15. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=banacorn.agda-mode.

[24] *Windows subsystem for linux (wsl)*, Accessed: 2025-04-15. [Online]. Available: https://ubuntu.com/desktop/wsl.

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/