Jack Gao

Using functor categories to generate intermediate code with Agda

Computer Science Tripos - Part II Dissertation

Homerton College

May 18, 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 18, 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] [1]
Code line count:	[linecount] [2]
Project Originator:	Yulong Huang
Project Supervisor:	Yulong Huang and Yufeng Li

Original Aims of the Project

Work Completed

Special Difficulties

[1] This word-count was computed by `texcount -1 -sum -merge -q dissertation.tex`.
[2] This code line count was computed by `find . -name "*.agda" | xargs wc -l`.

Contents

Chapter 1

Introduction

Contents

Programming languages act as a bridge between human thoughts 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 realm. 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 our intentions.

Compilers are related to the machine realm. 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. Simply typed lambda calculus (STLC) [1] is a well-studied programming language that serves as a foundation for many modern languages. It has denotational semantics in cartesian closed categories (CCC) [2]. As a CCC, presheaf categories over store locations can be used to model STLC with stores, as shown by by Reynolds [3] and Oles [4, 5]. Later, Reynolds presented a denotational semantics of STLC with stores in the form of a presheaf category over compiler states [6]. By interpreting the source language into the presheaf category over *stack descriptors*, where objects of the category represent instruction sequences parametrised by stack layouts, the semantic model directly yields a compiler.

In this dissertation, I implement Reynolds' presheaf-based compiler for STLC with stores in a dependently typed programming language, Agda [7]. The compiler is a *functor* (structure-preserving map) from the source language to the target language. The implementation is verified with Agda, which ensures that the input source program, the compilation process, and the output target program are all well-typed.

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. 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 [8] and CakeML [9] 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.

Motivation II. 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 [6, Ch.6]. I aim to provide a computer implementation of this theoretical framework in a dependently typed programming language.

1.2 Language choice: Agda's advantages

Agda [7], as a dependently typed proof assistant, captures source language's intrinsic syntax with indexed families. This automatically rules out the ill-typed nonsensical inputs, and only well-typed programs can be compiled.

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 [10], and it has been shown that functor categories arise naturally as dependent function types [11]. A formalisation of Category Theory, including cartesian closed categories, functors and presheaves has been developed in Agda by Hu and Caratte [12]. 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 §2.3.5.

1.3 Contributions

The success criteria of the project are:

- Formalise the definitions of the source language in Agda
- Formalise the definitions of the target language in Agda
- Implement a compiler from the source language to the target language in Agda
- The compiler successfully turns well-typed closed programs in the source language to valid target language expressions

The project successfully met all of the above success criteria, addressed the two motivations presented in §1.1 and contributed to the following:

> **Motivation I. Formal verification of the compiler**
>
> I formalised the terms in the source and target language in Agda. I also refined some type definitions in the denotational semantics of the source language.
>
> **Motivation II. Implementation of the compiler**
>
> I implemented a compiler from the source language to the target language in Agda.

Chapter 2

Preparation

Contents

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 dependent types to correctly express the dependent function space of presheaf exponentials.

This chapter begins with my starting point and introducing core concepts of category theory. It then provides a brief overview of Agda as a dependently typed programming language, concluding with a requirements analysis for the compiler design.

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) [13], 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 [14]. More specifically, it provides a "purer" view of functions that is not derived from sets [15]. 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 §2.2.10.

The following is a brief introduction to the basic concepts of category theory, which is based on the work of Leinster [14], Riehl [16], and the lecture notes of Part II Category Theory by Andrew Pitts and Marcelo Fiore [17].

2.2.1 Category

Definition 2.1 (Category). A *category* $\mathcal{C}$ is specified by

- 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. $f : X \xrightarrow{\mathcal{C}} Y$);
- 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)** 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)** 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 2.2 ($\mathcal{S}et$). An example of category is the category of sets, denoted as $\mathcal{S}et$, specified by the following:

- The objects of $\mathcal{S}et$ are small[1] 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.

Definition 2.3 (Preorder). A *preorder* $\underline{P} = (P, \sqsubseteq)$ is a set P with a binary relation $\sqsubseteq$ that is

- reflexive: $\forall x \in P, x \sqsubseteq x$;
- transitive: $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$.

Example 2.4 (Category determined by preorder). Another example of category is a category $\mathcal{C}_{\underline{P}}$ determined by any preorder $\underline{P}$, which is specified by the following:

- The objects of $\mathcal{C}_{\underline{P}}$ are the elements of P;
- $\mathcal{C}_{\underline{P}}(x,y) = \begin{cases} \{(x,y)\} & \text{if } x \sqsubseteq y \\ \emptyset & \text{otherwise} \end{cases}$
- The identity morphism on X in $\mathcal{C}_{\underline{P}}$ is the identity function on X;
- The composition of morphisms in $\mathcal{C}_{\underline{P}}$ is defined as the composition of functions.

[1] Here the smallness condition avoids foundational issues analogous to Russell's paradox. While we omit the formal definition of size due to the space limit here, all constructions in this work preserve smallness.

Associativity law and unit laws are satisfied in $\mathcal{C}_{\underline{P}}$.

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

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

- 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.

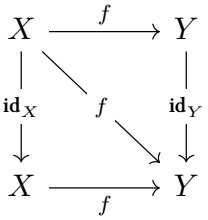


Figure 2.1: Commutative diagram for Unit Laws

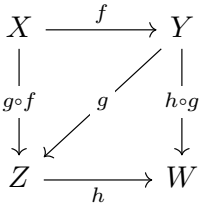
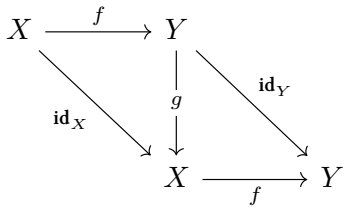


Figure 2.2: Commutative diagram for Associativity Law

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



(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

Definition 2.7 (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 **C**-morphism $f : X \to T$.

Terminal objects are unique up to isomorphism. In other words, we have the following properties:

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

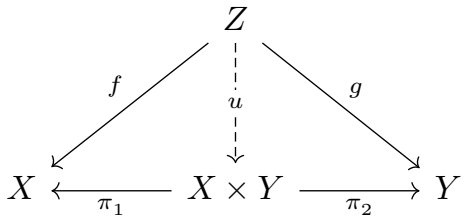
For any set X, there exists a unique function $f : X \to \{*\}$ that maps every element of X to the single element $*$ in $\{*\}$.

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

Definition 2.9 (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 $\mathcal{C}$:



(2.4)

The unique morphism u is written as $\langle f, g \rangle : Z \to X \times Y$ 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 2.10 (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 $(x, y) \in X \times Y$.

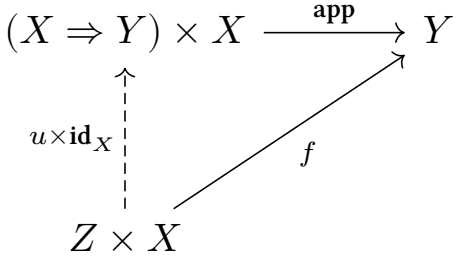
For any set Z, for any functions $f : Z \to X$ and $g : Z \to Y$, the unique morphism $u : Z \to X \times Y$ is defined as:

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

Definition 2.11 (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 $\mathcal{C}$:



(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 2.12 (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 2.13 (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 2.14 (Functor). Given two categories $\mathcal{C}$ and $\mathcal{D}$, a *functor* $F : \mathcal{C} \to \mathcal{D}$ is specified by

- a function referred to as object mapping

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

- for each $X, Y \in \mathbf{obj}(\mathcal{C})$, a function referred to as functorial mapping

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

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 2.15 (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

$$\begin{array}{ccc} F(X) & \xrightarrow{\theta_X} & G(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow[\theta_Y]{} & G(Y) \end{array} \tag{2.6}$$

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

2.2.6 Functor category

Definition 2.16 (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

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 $\mathcal{C}^{\mathbf{op}}$ and assigns them functions between the corresponding sets.

Definition 2.17 (Presheaf). Given a category $\mathcal{C}$, a *presheaf* on $\mathcal{C}$ is a functor $F : \mathcal{C}^{\mathbf{op}} \to \mathcal{S}et$. 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 2.18 (Presheaf category). Given a category $\mathcal{C}$, the *presheaf category* $\widehat{\mathcal{C}}$ is the functor category $\mathcal{S}et^{\mathcal{C}^{\mathbf{op}}}$, which explicitly contains the following:

- The objects of $\widehat{\mathcal{C}}$ 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 $\widehat{\mathcal{C}}$ are all natural transformations $\theta : F \to G$.

2.2.8 Yoneda lemma

Definition 2.19 (Yoneda functor). Given a category $\mathcal{C}$, the *Yoneda functor* $よ : \mathcal{C} \to \widehat{\mathcal{C}}$ is defined as follows:

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

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

  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 2.20 (Yoneda lemma). For each small[2] category $\mathcal{C}$, the *Yoneda lemma* states that for each object $X \in \mathbf{obj}(\mathcal{C})$ and each presheaf $F \in \hat{\mathcal{C}}$, there exists 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 Sketch 2.21 (Cartesian closed structure in presheaf categories). Given a small[3] 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,

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

2.2.10 Curry-Howard-Lambek correspondence

Joachim Lambek showed that cartesian closed categories provide a natural semantic setting for the simply typed lambda calculus (STLC) [2]. His work builds upon Curry-Howard correspondence [18], the isomorphism between logic and type theory, where propositions correspond to types and proofs correspond to terms. The Curry-Howard-Lambek correspondence can be summarised as follows:

[2] See footnote 1.
[3] See footnote 1.

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

This correspondence forms the foundation of functional programming languages that can be used to implement proofs.

2.3 Agda

2.3.1 Basic data types and pattern matching

Here is a simple example selected from PLFA [13] to illustrate the basic data types and pattern matching in Agda. The example is a definition of natural numbers and a plus function.

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

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

In Agda we use the data keyword to define a type. Every type has a type, and we use Set to denote the type of all small types. To define a type, we specify its type and its constructors. To define a function, we specify its type and pattern match the input according to the constructors of the type.

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 a data type Vec is defined for vectors. The type of vectors is dependent on the length of the vector. With dependent types, we can encode properties directly into types.

2.3.3 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 $x \equiv y$, and then give its definition, which constructs a proof of the equality. All of the examples here are type-checked by Agda, so they are all valid proofs.

Two simple proofs that directly use 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 y and z are reduced to x by the definition of the constructor refl. The properties are called *symmetry* and *transitivity* of equality, which 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
```

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

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

In this example, we want to prove that zero is the right identity of the plus function. This property cannot be directly shown by using refl because refl only knows about definitional equality. In the definition of the plus function, only the first argument is pattern matched, so we can only show that zero is the left identity of the plus function by directly sing the refl. To show that zero is the right identity of the plus function, we need to do an inductive proof by pattern matching on the first argument of the plus function.

In the base case, we have zero + zero ≡ zero, which is true by the definition of the plus function.

In the inductive case, we need to show suc n + zero ≡ suc n. The left hand side is definitionally equal to suc (n + zero). By the inductive hypothesis, we know that n + zero ≡ n, so we can substitute n for n + zero in the right-hand side by congruence, and we are done.

2.3.4 Standard library

The Agda standard library [19] 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, standard library is used as a reference for defining a customised library, which is specified in §3.5.

Builtin pragma

With standard library, we can use literal natural numbers 1, 2, ... instead of calling the constructor suc multiple times. This feature can be extended to self-defined functions with a BUILTIN pragma. We simply add the following two lines after the definition of ℕ and _+_:

```
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN NATPLUS _+_ #-}

1+1≡2 : 1 + 1 ≡ 2
1+1≡2 = refl
```

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

2.3.5 Interactive programming with holes

A feature of Agda is interactive programming with holes. When users leave holes in place of undefined terms, Agda's type checker will infer the type of the hole, show 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. This feature is used in my implementation. The complex terms are incrementally filled and verified by refining partial implementations, reducing post-hoc debugging and ensuring robustness.

2.4 Requirement Analysis

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

- A file `source.agda` that record the syntax of the source language, which include basic instructions in STLC with store.
- A file `target.agda` that record the syntax of the target language.
- A file `compiler.agda` that uses `source.agda` and `target.agda`, and write functions whose input is a closed term in the source language and output is a term in the target language.
- A file `test.agda` that contains test cases for the compiler. The test cases are written in the source language, and the expected output is written in the target language.

The implementation and type-checking of all files specified above constitutes a necessary and sufficient condition for meeting the project's success criteria mentioned in §1.3.

Chapter 3

Implementation

Contents

This chapter details the implementation of the compiler in Agda. Following a brief overview of the tools used (§3.1), it provides a repository overview (§3.2). Then it discusses the intrinsic syntax of the source language (§3.3) and the grammar of target language (§3.4), leading to the problem of implementing a proof-relevant natural number subtraction.

To address this, it presents the customised library (§3.5) with two different approaches for implementing subtraction: one using additional proof object and the other using Fin, a dependent type. It compares the two approaches and discusses the trade-offs between them. The chapter concludes with a detailed implementation of the compiler (§3.6).

3.1 Tools used

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

Completing the project is an iterative process. I used Git [20] for version control, and work had been synchronised with a GitHub [21] repository for backup.

For the development environment, I tried both Emacs [22] and Visual Studio Code [23] with an agda-mode extension [24] on Windows Subsystem for Linux with Ubuntu [25] 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 [19] 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 in two directories: `fin` and `proof` as follows:

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

3.2.2 File dependencies and descriptions

The dependency graph for each implementation is identical as follows:

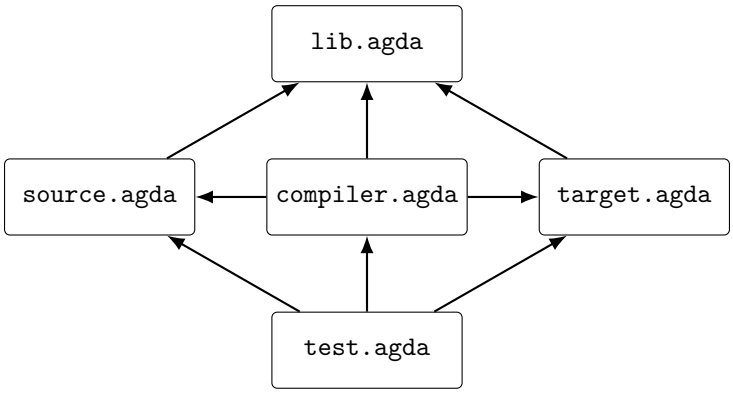


Figure 3.1: Dependency graph for the compiler

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 derived from denotational semantics
`test.agda`	test cases for the compiler

Table 3.1: File descriptions

3.3 Source Language

The source language is a simply typed lambda calculus (STLC) with store, following the syntax in Reynolds' paper [6].

3.3.1 Types

The source language has four primitive types:

- comm: commands, which is the type of programs
- intexp: integer expressions, corresponding to `int` in most programming languages
- intacc: integer acceptors, representing reference to integer values
- intvar: integer variables, which are integer acceptors that can also be used as expressions with automatic dereferencing

and general types are defined as follows:

$$A, B := \mathsf{comm} \mid \mathsf{intexp} \mid \mathsf{intacc} \mid \mathsf{intvar} \mid A \Rightarrow B$$

which corresponds with the following agda implementation:

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

Subtypes

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

$$\frac{}{A \leq: A}\ \textsc{Reflexivity} \qquad \frac{A \leq: A' \qquad A' \leq: A''}{A \leq: A''}\ \textsc{Transitivity}$$

Figure 3.2: Preorder of subtypes

The source language has the following subtype relations:

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

since a variable can be used as an expression (e.g. $\mathsf{x} + 1$) or an acceptor (e.g. $\mathsf{x} := 1$).

For function types we have the contravariant subtyping:

$$\frac{A' \leq: A \qquad B \leq: B'}{A \Rightarrow B \leq: A' \Rightarrow B'}\ \textsc{Function}$$

Figure 3.3: Contravariant subtyping for function types

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 and variables

We define the context as a finite list of types as follows:

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

data _∈_ : Type → Context → Set where
  Zero : ∀ {Γ A} → A ∈ Γ , A
  Suc : ∀ {Γ A B} → B ∈ Γ → B ∈ Γ , A
```

When we are looking for a variable, Zero case corresponds to the situation where the variable is the head of the context, and the Suc case corresponds to the situation where the variable is in the tail of the context. The number formed by Zero and Suc is the index of the variable in the context, called the *de Bruijn index* [26].

3.3.3 Terms and the typing judgement

The terms are defined as follows:

$$c_1, c_2 = \texttt{skip} \mid c_1; c_2 \mid a := e \mid \texttt{new}\ \ x\ \ \texttt{in}\ \ c_1$$
$$e_1, e_2 = \texttt{lit}\ \ z \mid -e_1 \mid e_1 + e_2 \mid \lambda e_1.\, e_2 \mid e_1\ e_2 \mid x$$

and the typing judgement are described in the following rules:

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

Figure 3.4: Variable typing rule

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

Figure 3.5: Subtyping rule

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

Figure 3.6: Lambda abstraction typing rules

$$\frac{}{\Gamma \vdash \texttt{skip} : \mathsf{comm}}\ \textsc{Skip} \qquad \frac{\Gamma \vdash c_1 : \mathsf{comm} \qquad \Gamma \vdash c_2 : \mathsf{comm}}{\Gamma \vdash c_1; c_2 : \mathsf{comm}}\ \textsc{Seq}$$

Figure 3.7: Command typing rules (part 1)

$$\frac{\Gamma \vdash a : \mathsf{intacc} \qquad \Gamma \vdash e : \mathsf{intexp}}{\Gamma \vdash a := e : \mathsf{comm}}\ \textsc{Assign} \qquad \frac{\Gamma, x : \mathsf{intvar} \vdash c : \mathsf{comm}}{\Gamma \vdash \texttt{new}\ \ x\ \ \texttt{in}\ \ c : \mathsf{comm}}\ \textsc{NewVar}$$

Figure 3.8: Command typing rules (part 2)

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

Figure 3.9: Integer expression typing rules

The corresponding Agda implementation is as follows, note that the Agda implementation only does not include the name of the terms, as a term can be specified by how it is constructed (i.e. the name of the typing rules and the corresponding types)

```
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
```

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 Appendix A. Operational semantics can be used to verify the correctness of the compiler in the future, but it is beyond the scope of this project.

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:

- $<\mathsf{L}_{sd}>$: left-hand sides
- $<\mathsf{S}_{sd}>$: simple right-hand sides
- $<\mathsf{R}_{sd}>$: right-hand sides

- $<\mathsf{I}_{sd}>$: instruction sequences

The intent of the indexing here is any instructions with the form $<\mathsf{I}_{sd}>$ is an instruction sequence that can be executed when the current stack descriptor is *sd*.

The following subsections start with the definition of stack descriptors, then define the grammar of the target language, and finally discuss the problem of subtraction of natural numbers.

3.4.1 Stack descriptor

Now, I define the stack descriptor in Agda, as well as their pre-order, operations (addition and subtraction), and properties about ordering.

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 d : ℕ
```

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' ⟩
```

Addition and subtraction of stack descriptors

The addition and subtraction of stack descriptors is defined 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. $\langle f, d \rangle - sn$ is only defined for $n \leq d$. This leads us to the problem of defining the subtraction of natural numbers, which is explained in detail in §3.5.2.

Properties

The properties of stack descriptors can be summarised as follows:

Agda term	Mathematical meaning
$\leq_s$-refl	$\forall sd.\ sd \leq_s sd$
$\leq_s$-trans	$\forall sd, sd', sd''.\ sd \leq_s sd' \wedge sd' \leq_s sd'' \Rightarrow sd \leq_s sd''$
$+_s \longrightarrow \leq_s$	$\forall sd, n.\ sd \leq_s sd +_s n$
sub-sd$\leq_s$	$\forall sd, sd', sd''.sd' \equiv sd'' \wedge sd \leq_s sd' \Rightarrow sd \leq_s sd''$
$-_s\equiv$	$\forall f, d, d', n.\ d' - n \equiv d \Rightarrow \langle f, d\rangle \equiv \langle f, d'\rangle -_s n$

Table 3.2: Properties of order and addition of stack descriptors

3.4.2 Grammar

The grammar of the target language[1] is defined as follows, given the current stack descriptor $sd = \langle f, d\rangle$, and new stack descriptor sd' and sd^v:

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

Figure 3.10: Target language grammar

Some explanations of the grammar are as follows:

- `lit`: a literal integer.
- `stop`: stops the execution of the program.
- δ: an integer representing the displacement adjustment.
- `adjustdisp`: adjusts the displacement of specified stack descriptor.

[1] This grammar can be extended with subroutines as an L command, and conditionals as an I command

- `popto`: pops to the specified stack descriptor, which is used to reduce the number of frames.

Note that the design of right-hand sides ensures that the right operand of the assignment contains at most one operator, which means that any instruction that involves multiple operators must be divided into multiple instructions. This feature of the target language significantly complicates the compilation of integer expressions, which we will analysis in detail in §3.6.5.

A fundamental property of the target language is its order-preserving extension with respect to stack descriptors: any term operating on a smaller stack descriptor can be systematically lifted to operate on a larger one. As later formalised in §3.6.4, this property exhibits a functorial nature, and is essential for the construction of the compiler.

The operators are defined as follows:

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

The operators are defined in Agda as follows:

```
data UnaryOp : Set where
  UNeg : UnaryOp

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

The left-hand sides, simple right-hand sides and right-hand sides are straightforward, and they are defined as follows in Agda:

```
data L (sd : SD) : Set where
  l-var : (sd^v : SD) → sd^v ≤ₛ sd → L sd

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

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
```

In the instruction sequences, δ as a displacement adjustment can be positive, zero or negative. To make a rigorous definition, I 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.

The assignment with negative displacement is left to discuss in §3.5.2, since it involves proof objects to validate the subtraction. The remaining instruction sequences are defined in Agda

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
  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 centred around natural numbers, their operations and properties. Instead of using the standard library [19], I implemented a customised library for the following reasons:

- **Specialised Requirements**: The standard library's definitions (e.g. subtraction as monus, '$\dot{-}$') do not align with my project's need for a rigorously defined subtraction operation. This is specified in §3.5.2.
- **Verification Clarity**: By defining the basic data types and functions from scratch, I can tailor properties (e.g. the property '$n - [n - m] \equiv m$') to my specific requirements, and avoid unnecessary complexity (e.g. '$\leq$' being also defined as a boolean predicate in the standard library).
- **Minimal Dependencies**: Avoiding the standard library reduces external assumptions, making the project self-contained.

All definitions and properties can be found in Appendix [TBD].

3.5.1 Definitions from standard library

The following is a list of definitions (or their variants) from the standard library and PLFA [13] that are redefined in the customised library:

- Natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$) and addition ($+$).
- Equality ($\equiv$), congruence (cong), substitution (sub) and transitivity of equality (trans).
- Order of natural numbers, including $\leq$ and $<$.
- Properties of preorder and addition (summarised in Table 3.3).

Agda term	Mathematical meaning
+-identity^r	$\forall n \in \mathbb{N}.\ n + 0 = n$
+-suc^r	$\forall m, n \in \mathbb{N}.\ m + \mathsf{suc}\ n = \mathsf{suc}\ (m + n)$
+-comm^r	$\forall m, n \in \mathbb{N}.\ m + n \equiv n + m$
≤-irrelevant	For all $m, n \in \mathbb{N}$, we consider all proofs of $m \leq n$ to be equal.
≤-refl	$\forall n \in \mathbb{N}.\ n \leq n$
≤-trans	$\forall m, n, p \in \mathbb{N}.\ m \leq n \wedge n \leq p \Rightarrow m \leq p$
n≤suc-n	$\forall n \in \mathbb{N}.\ n \leq \mathsf{suc}\ n$
m≡n,p≤n→p≤m	$\forall p, m, n \in \mathbb{N}.\ m \equiv n \wedge p \leq n \Rightarrow p \leq m$
+→≤	$\forall m, n \in \mathbb{N}.\ m \leq m + n$
+→≤^{r}	$\forall m, n \in \mathbb{N}.\ m \leq n + m$
<-trans	$\forall m, n, p \in \mathbb{N}.\ m < n \wedge n < p \Rightarrow m < p$
<→s≤	$\forall m, n \in \mathbb{N}.\ m < n \Rightarrow \mathsf{suc}\ m \leq n$
<→≤	$\forall m, n \in \mathbb{N}.\ m < n \Rightarrow m \leq n$
≤-compare	$\forall m, n \in \mathbb{N}.\ m \leq n \vee n \leq m$

Table 3.3: Properties of order and addition

3.5.2 Subtraction

In the standard library [19], the subtraction of natural numbers is defined as follows:[2]

```
_∸_ : 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 $\dot{-}$ is not suitable for this project. 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.

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

- We do not have many numerical properties of natural numbers that are required for the implementation of the compiler. For example, $n \dot{-} m + m = n$ or $n \dot{-} (n \dot{-} m) = m$ are not guaranteed to hold if we allow $n < m$.

A definition of subtraction that ensures the result is a natural number and preserves the numerical properties is required for the implementation of the compiler. 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: passing proofs explicitly or using Fin, a dependent type that encodes the proof of boundedness.

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
```

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
  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 to be dependent on the minuend, where the dependence 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 $n = 0$, we do not have any constructor that gives us an element of type Fin 0, corresponding to the empty set.

- Inductive case:

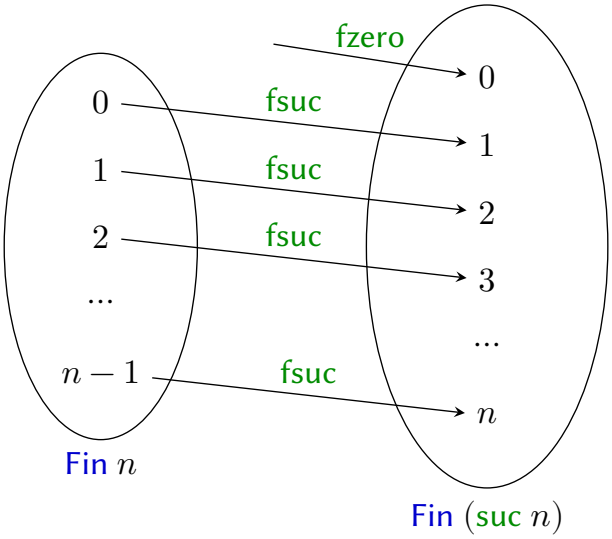


Figure 3.11: Inductive case of Fin

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
adjustdisp-dec : (δ : Fin (suc (SD.d sd))) → I (sd −ₛ δ) → I sd
```

Comparison of the two approaches

The explicit approach carries proofs directly in the type system, which is more transparent but verbose. The Fin-based approach initially 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-based 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 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 3.12: Comparison of explicit vs. Fin-based approaches

The explicit approach's verbosity pays off in readability when representing properties. the type of the term directly mirrors the mathematical statement with an explicit proof. In contrast, the Fin-based approach fractures this relationship, and reconstructing the original subtrahend requires tracing the type of the proof. This becomes cumbersome when the proofs are chained inequalities with transitivity, making the code harder to understand.

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

3.5.3 Properties with subtractions

Given the definition of subtraction, the additional properties of natural numbers are summarised in the table below:

Agda term	Mathematical meaning
--irrelevant	For all $m, n \in \mathbb{N}$, we consider all terms of $n - m$ with different proofs of $m \leq n$ to be equal.
−→≤	$\forall m, n \in \mathbb{N}.\ m \leq n \Rightarrow m - n \leq m$
n−n≡0	$\forall n \in \mathbb{N}.\ n - n \equiv 0$
--suc	$\forall m, n \in \mathbb{N}.\ \text{suc}\ (n - m) \equiv \text{suc}\ n - m$
n−[n−m]≡m	$\forall m, n \in \mathbb{N}.\ m \leq n \Rightarrow n - (n - m) \equiv m$
n+m−m≡n	$\forall m, n \in \mathbb{N}.\ n + m - m \equiv n$
suc-d≤d′→d≤d′−[d′−[suc-d]]	$\forall d, d' \in \mathbb{N}.\ \text{suc}\ d \leq d' \Rightarrow d \leq d' - (d' - \text{suc}\ d)$

Table 3.4: Properties of natural numbers related to subtraction

3.6 Compiler

This section develops the semantic foundation of the compiler, starting with the presheaf semantics of the target language. Building on this framework, I introduce the continuation-passing constructs to model control flow, enabling a simplified denotational semantics of types

and contexts. I then derive a functorial mapping of semantics terms and the target terms. This section concludes with the implementation of the compiler from the denotational semantics of terms in the source language.

3.6.1 Presheaf semantics

The denotation semantics interprets types as presheaves over a base category Σ.

Base category

The base category Σ is specified as follows:

- Objects are stack descriptors ($sd = \langle f, d \rangle$);
- Morphisms are stack expansions ($\iota : sd \to sd'$ for $sd \leq_{\mathrm{s}} sd'$).

We define SD as the set of stack descriptors. Then the base category is the category determined by preorder $\underline{\mathrm{SD}} = \{\mathrm{SD}, \leq_{\mathrm{s}}\}$ in Example 2.4.

Semantic category

Semantic category $\mathcal{K}$ is the functor category $\mathrm{PDOM}^{\Sigma^{\mathbf{op}}}$, where PDOM is the category of predomains and continuous functions.

- Objects are presheaves: $P = \Sigma^{\mathbf{op}} \to \mathrm{PDOM}$;
- Morphisms are natural transformations: $\eta : P \to Q$ for $P, Q \in \mathrm{PDOM}^{\Sigma^{\mathbf{op}}}$.

The proof for semantic category is similar to the one in §2.2.9, and the full proof is given by Oles in [4, 5].

- Products are given by pointwise product of presheaves: $P \times Q = \lambda sd.\ P\ sd \times Q\ sd$.
- Exponentials $P \Rightarrow_s Q$ are given by the following equation:

$$
\begin{aligned}
P \Rightarrow_s Q &= \lambda sd.\ \mathcal{K}(\text{ɏ}(sd) \times P, Q) \\
&\cong \lambda sd.\{F \mid sd' \in \Sigma^{\mathbf{op}}, F : (\text{ɏ}(sd) \times P)\ sd' \to Q\ sd'\} \\
&\qquad\qquad \text{by definition of morphisms in Definition 2.1} \\
&\cong \lambda sd.\{F \mid sd' \in \Sigma^{\mathbf{op}}, F : (\Sigma^{\mathbf{op}}(sd', sd) \times P)\ sd' \to Q\ sd'\} \\
&\qquad\qquad \text{by definition of ɏ in Definition 2.19} \\
&\cong \lambda sd.\{F \mid sd' \in \Sigma, F : (\Sigma(sd, sd') \times P)\ sd' \to Q\ sd'\} \\
&\qquad\qquad \text{by definition of opposite category in Definition 2.5} \\
&\cong \lambda sd.\{F \mid sd' \in \Sigma, F : sd \leq_{\mathrm{s}} sd' \to P\ sd' \to Q\ sd'\} \\
&\qquad\qquad \text{by definition of } \Sigma \text{ above}
\end{aligned}
$$

The definition of products and exponentials in Agda is as follows:

```
_×ₛ_ : (SD → Set) → (SD → Set) → SD → Set
(P ×ₛ Q) sd = P sd × Q sd

_⇒ₛ_ : (SD → Set) → (SD → Set) → SD → Set
(P ⇒ₛ Q) sd = ∀{sd′} → (sd ≤ₛ sd′) → P sd′ → Q sd′
```

The definition of exponentials captures the essence of stack-respecting computations in the compiler.

- $sd \leq_s sd'$ means the new stack descriptor sd' is larger than the old stack descriptor sd, representing the dynamic expansion of the stack during execution.
- P and Q are denotational semantics of types.

The exponential $P \Rightarrow_s Q$ is a function that given a proof of stack growth, transforming a P-value at sd' into a Q-value at sd', and the $\forall\, sd'$ quantifier guarantees it works no matter how the stack grows. It is designed in this way so that a functor $[\![-]\!] : \Theta \to \mathcal{K}$ exists that interprets the type constructor $\Rightarrow$ as the exponential functor $\Rightarrow_s$.

3.6.2 Continuations

In Reynolds' model, we construct Compl and Intcompl[3] as tools in the denotational semantics to represent command continuations and integer continuations, respectively. We can think of integer continuations as a function that takes an integer and returns a command continuation. Mathematically we have

$$\mathsf{Compl}\ sd : \mathsf{SD} \to O \qquad \mathsf{Intcompl}\ sd : \mathbb{Z} \to (\mathsf{SD} \to O)$$

where O is an unspecified domain of outputs.

For code generation I define the continuation to be the instruction sequence in the target language:

$$\mathsf{Compl}\ sd = \mathsf{I}_{sd}$$

The definition of integer continuation is an exponential object in the presheaf category, which is defined as follows:

$$\mathsf{Intcompl} = R \Rightarrow_s \mathsf{Compl}$$

where R is the functor corresponding to the right-hand sides in the target language, $R\ sd = \mathsf{R}_{sd}$. This works because the right-hand sides are all integer expressions we can use according to the grammar of the target language. The functorial mapping is defined later in §3.6.4.

[3] In the work of Reynolds, compl and intcompl are introduced as new types in the source language, and the definitions here are the denotational semantics of those types, respectively. Since these two introduced types are only used for the denotational semantics, I think directly defining them as denotational semantics constructs is natural and sufficient.

Thus we can directly use the definition of R in the target language to define the integer continuation as follows:

```
Compl : SD → Set
Compl sd = I sd

Intcompl : SD → Set
Intcompl = R ⇒ₛ Compl
```

3.6.3 Denotational semantics of types and contexts

With the idea of continuation, the denotational semantics of types can be defined as follows:

- Command passes on command continuations;
- Integer expressions takes an integer continuation and returns a command continuation;
- Integer acceptors take a command continuation and returns an integer continuation;
- Integer variables can be used both as an integer expression and an integer acceptor, so it has the denotational semantics of both, defined as a product of the two.

The denotational semantics of types in Agda as follows:

```
⟦_⟧ty : Type → SD → Set
⟦ comm ⟧ty = Compl ⇒ₛ Compl
⟦ intexp ⟧ty = Intcompl ⇒ₛ Compl
⟦ intacc ⟧ty = Compl ⇒ₛ Intcompl
⟦ intvar ⟧ty = ⟦ intexp ⟧ty ×ₛ ⟦ intacc ⟧ty
⟦ A ⇒ B ⟧ty = ⟦ A ⟧ty ⇒ₛ ⟦ B ⟧ty
```

The denotational semantics of contexts is essentially a list expressed by the product of the denotational semantics of the types in the context. The denotational semantics of contexts is defined as follows, where we use $\emptyset$ to denote the denotational semantics of the empty context:

```
data ∅ : Set where
  unit : ∅

⟦_⟧ctx : Context → SD → Set
⟦ · ⟧ctx _ = ∅
⟦ Γ , A ⟧ctx sd = ⟦ Γ ⟧ctx sd × ⟦ A ⟧ty sd
```

3.6.4 Functorial mapping

The denotational semantics of terms is defined as a functor. Before we define its object mapping, we need to define the functorial mapping, which acts on the morphisms of the base category Σ.

Since the morphisms of the base category Σ are stack expansions, the functorial mapping essentially lifts a semantic object indexed by a stack descriptor sd to a semantic object indexed

by a larger stack descriptor sd'. This ensures the denotational semantics of terms remains valid when the stack grows.

Functorial mapping for exponentials is simple, as the exponential $P \Rightarrow_S Q\ sd$ can transforms a P-value at any sd' greater than sd into a Q-value at sd'. To construct a new exponential of type $P \Rightarrow_S Q\ sd'$, we simply take a new P-value at sd'' which is greater than sd', and use the original exponential to transform it into a Q-value at sd', as sd'' is guaranteed to be greater than sd due to the transitivity of order of the stack descriptors.

Functorial mapping for types can be defined based on functorial mapping of exponentials and products (defined point-wise), as denotational semantics of types is defined as either product or exponential. As a list of types, the functorial mapping of contexts is simply a map of the functorial mapping of types in the list. The definition of functorial mapping in Agda is as follows:

```
fmap-⇒ : ∀ {P Q sd sd'} → (P ⇒ₛ Q) sd → sd ≤ₛ sd' → (P ⇒ₛ Q) sd'
fmap-⇒ P⇒Q sd≤ₛsd' sd'≤ₛsd'' p = P⇒Q (≤ₛ-trans sd≤ₛsd' sd'≤ₛsd'') p

fmap-ty : ∀ {A sd sd'} → ⟦ A ⟧ty sd → sd ≤ₛ sd' → ⟦ A ⟧ty sd'
fmap-ty {comm} = fmap-⇒ {Compl} {Compl}
fmap-ty {intexp} = fmap-⇒ {Intcompl} {Compl}
fmap-ty {intacc} = fmap-⇒ {Compl} {Intcompl}
fmap-ty {intvar} ( exp , acc ) sd≤ₛsd' =
  ( fmap-ty {intexp} exp sd≤ₛsd' , fmap-ty {intacc} acc sd≤ₛsd' )
fmap-ty {A ⇒ B} = fmap-⇒ {⟦ A ⟧ty} {⟦ B ⟧ty}

fmap-ctx : ∀ {Γ sd sd'} → ⟦ Γ ⟧ctx sd → sd ≤ₛ sd' → ⟦ Γ ⟧ctx sd'
fmap-ctx {·} unit _ = unit
fmap-ctx {Γ , A} (γ , a) p = fmap-ctx γ p , fmap-ty {A} a p
```

Similarly, we can use transitivity of order to define the functorial mapping of the denotational semantics of left-hand sides, as the definition of left-hand sides requires a proof. Since simple right-hand sides are defined from left-hand sides and literals, we can use the functorial mapping of left-hand sides to define the functorial mapping of simple right-hand sides. The functorial mapping of the denotational semantics of left-hand sides and simple right-hand sides is defined as follows:

```
fmap-L : ∀ {sd sd'} → L sd → sd ≤ₛ sd' → L sd'
fmap-L (l-var sdᵛ sdᵛ≤ₛsd) sd≤ₛsd' = l-var sdᵛ (≤ₛ-trans sdᵛ≤ₛsd sd≤ₛsd')

fmap-S : ∀ {sd sd'} → S sd → sd ≤ₛ sd' → S sd'
fmap-S (s-l l) sd≤ₛsd' = s-l (fmap-L l sd≤ₛsd')
fmap-S (s-lit lit) _ = s-lit lit
```

Similarly, the functorial mapping of right-hand sides can be defined based on the functorial mapping of simple right-hand sides. Since it is not used in the implementation of the compiler, the definition is omitted here.

Functorial mapping for instruction sequences

The functorial mapping of the denotational semantics of instruction sequences is defined as follows, assuming the stack descriptor $sd' = \langle f', d' \rangle$ is greater than $sd = \langle f, d \rangle$:

- When $f < f'$, we use the popto instruction that directly pops the stack to sd;
- When $f' = f$ and $d < d'$, we use adjustdisp to decrease the displacement by $d' - d$.

In the Agda implementation, we also need to show that after the the adjustment, the stack descriptor is now sd. This requires the term n−[n−m]≡m proved in §3.5.3. The functorial mapping of the denotational semantics of instruction sequences is defined as follows:

```
fmap-I : ∀ {sd sd′} → I sd → sd ≤ₛ sd′ → I sd′
fmap-I {sd} c (<-f f≤f′) = popto sd (<-f f≤f′) c
fmap-I {⟨ f , d ⟩} {⟨ f , d′ ⟩} c (≤-d d≤d′) =
  adjustdisp-dec ((d′ − d) d≤d′) (−→≤ d≤d′)
    (I-sub {n = (d′ − d) d≤d′} (n−[n−m]≡m d≤d′) c)
```

3.6.5 Compilation

The denotational semantics of terms is defined as a morphism from the denotational semantics of contexts to the denotational semantics of types. The denotational semantics of terms is a functor from the source language to the target language, which directly yields a compiler. Similarly, it is a family of continuous functions indexed by the stack descriptor. The denotational semantics of terms has the following type in Agda:

```
⟦_⟧ : ∀ {Γ A} → Γ ⊢ A → (sd : SD) → ⟦ Γ ⟧ctx sd → ⟦ A ⟧ty sd
```

The denotational semantics of the terms corresponding to all rules in the typing judgement mentioned in §3.3.3 is defined as follows:

Variables

We define an auxiliary function that simply gets the variable from the list of variables (the context), and the denotational semantics of variables is given by applying this function:

```
⟦_⟧var : ∀ {Γ A sd} → A ∈ Γ → ⟦ Γ ⟧ctx sd → ⟦ A ⟧ty sd
⟦ Zero ⟧var (_ , a) = a
⟦ Suc b ⟧var (γ , _) = ⟦ b ⟧var γ

⟦ Var a ⟧ sd γ = ⟦ a ⟧var γ
```

Subtyping

The idea of subtyping is to use a term of type A as a term of type A', where A is a subtype of A'. We define an auxiliary function that takes in the subtype relation $A \leq: A'$ and the denotational semantics of A, and returns the denotational semantics of A'.

- For var-≤:-exp and ⟦var-≤:-acc, sincethe denotational semantics of integer variable is defined as product of the denotational semantics of integer expression and integer acceptor in §3.6.3, we can simply use the projection to get the denotational semantics of either side.

- For ≤:-refl, we can simply use the identity function.
- For ≤:-trans, we can use the composition of the two functions.
- For ≤:-fn, we can use a contravariant function.

The denotational semantics of subtyping is given by applying the auxiliary function:

```
⟦_⟧sub : ∀ {A A′ sd} → A ≤: A′ → ⟦ A ⟧ty sd → ⟦ A′ ⟧ty sd
⟦ ≤:-refl ⟧sub a = a
⟦ ≤:-trans A≤:A′ A′≤:A″ ⟧sub a = ⟦ A′≤:A″ ⟧sub (⟦ A≤:A′ ⟧sub a)
⟦ ≤:-fn A≤:A′ B′≤:B ⟧sub a =
    λ sd≤ₛsd′ a′ → ⟦ B′≤:B ⟧sub (a sd≤ₛsd′ (⟦ A≤:A′ ⟧sub a′))
⟦ var-≤:-exp ⟧sub (exp , acc) = exp
⟦ var-≤:-acc ⟧sub (exp , acc) = acc

⟦ Sub a A≤:B ⟧ sd γ = ⟦ A≤:B ⟧sub (⟦ a ⟧ sd γ)
```

Lambda abstraction

- For lambda, we need to extend the context γ with the extra variable in the lambda term. This is done by using the functorial mapping of the denotational semantics of contexts.
- For application, we simply apply the denotational semantics of the lambda term to the denotational semantics of the argument, where the proof is simply the reflexivity of the order of stack descriptors.

The denotational semantics of lambda abstraction is defined as follows:

```
⟦ Lambda f ⟧ sd γ {sd′ = sd′} sd≤ₛsd′ a = ⟦ f ⟧ sd′ (fmap-ctx γ sd≤ₛsd′ , a)
⟦ App f e ⟧ sd γ = ⟦ f ⟧ sd γ (≤-d ≤-refl) (⟦ e ⟧ sd γ)
```

Command

- For skip, we simply return the continuation κ without changing it.
- For sequence, we need to prefix the continuation with the denotational semantics of the second command, and then prefix it with the denotational semantics of the first command, as the continuation κ is to be performed after both commands.
- For assignment, the definition is similar to sequence, as we need to prefix the continuation with the denotational semantics of the integer accepter and then prefix it with the denotational semantics of the integer expression.

The denotational semantics of commands is defined as follows:

```
⟦ Skip ⟧ sd γ sd≤ₛsd′ κ = κ
⟦ Seq c₁ c₂ ⟧ sd γ sd≤ₛsd′ κ = ⟦ c₁ ⟧ sd γ sd≤ₛsd′ (⟦ c₂ ⟧ sd γ sd≤ₛsd′ κ)
⟦ Assign a e ⟧ sd γ sd≤ₛsd′ κ = ⟦ e ⟧ sd γ sd≤ₛsd′ (⟦ a ⟧ sd γ sd≤ₛsd′ κ)
```

New variable The denotational semantics of NewVar command is more complicated. We prefix the continuation with the following in order:

1. an assignment with displacement adjustment 1, representing the new variable;
2. the denotational semantics of the command where this variable is used;
3. an adjustment of the stack descriptor by -1, which deallocates the new variable.

In step 2, the denotational semantics of the command in NewVar needs to be applied to

- a stack descriptor increased by 1 to include the new variable;
- an extended denotational semantics of the context, which includes the new variable.

The denotational semantics of the new variable is a product of the denotational semantics of an integer expression and the denotational semantics of an integer acceptor by definition in §3.6.3.

- The expression component fills the a given integer continuation β with the stack descriptor for the new variable;
- The acceptor component prefixes the continuation κ with an assignment of the new variable.

Please see Appendix [TBD] for the full implementation of the function NewVar.

Integer expression

The denotational semantics of type intexp is an exponential from an integer continuation to a command continuation. We need to fill the integer continuation with appropriate R terms to get the command continuation.

- For integer literals, we can simply fill in the integer literal (as an R term) to the integer continuation β.
- For addition and negation, we wish to fill the negation or the sum of the given expression(s) to the integer continuation β. However, this is only possible when the given expression(s) are simple left-hand sides, as the grammar of the target language in Figure 3.10 restricts right-hand sides to contain at most one operator. An auxiliary function use-temp that stores the intermediate result of given expressions is introduced to solve this problem.

With use-temp, the denotational semantics of addition and negation is simply wrapping β with a lambda function using use-temp to get temporary variable(s) of the given expression(s), and directly fill the negation or the sum of the variable(s) to the integer continuation β. Note that in the denotational semantics of addition, we need to use the functorial mapping of simple

right-hand sides for the first argument, as the stack may grow during the evaluation of the second argument. The denotational semantics of integer expressions is defined as follows:

```
⟦ Lit i ⟧ sd γ sd≤ₛsd′ β = β ≤ₛ-refl (r-s (s-lit i))

⟦ Neg e ⟧ sd γ sd≤ₛsd′ β =
  ⟦ e ⟧ sd γ sd≤ₛsd′ (use-temp λ sd≤ₛsd′ s → β sd≤ₛsd′ (r-unary UNeg s))

⟦ Plus e₁ e₂ ⟧ sd γ sd≤ₛsd′ β =
  ⟦ e₁ ⟧ sd γ sd≤ₛsd′
    (use-temp (λ sd′≤ₛsd″ s₁ → ⟦ e₂ ⟧ sd γ (≤ₛ-trans sd≤ₛsd′ sd′≤ₛsd″)
      (use-temp (λ sd″≤ₛsd‴ s₂ → β (≤ₛ-trans sd′≤ₛsd″ sd″≤ₛsd‴)
        (r-binary (fmap-S s₁ sd″≤ₛsd‴) BPlus s₂)))))
```

The auxiliary function use-temp checks if the given expression is a simple left-hand side,

- If it is, we simply fill the expression to the integer continuation β.[4]
- If it is not, we need to use a temporary variable to store the result of the expression, and then fill the temporary variable to the integer continuation β.

However, regarding the stack descriptor, according to Reynolds' model, we assume

- sd is the stack descriptor before the evaluation of plus or negation, which is the position of the temporary variable in the stack.
- sd' is the stack descriptor after the evaluation on the given expression and just before the assignment of the temporary variable. $sd \leq_s sd'$ is guaranteed.
- $sd'' = sd +_s 1$ is the stack descriptor after the assignment of the temporary variable, as it is just large enough to hold the temporary variable.

In Reynolds' model, the stack descriptor can be directly adjusted from sd' to sd''. However, in the Agda implementation, we need to know whether the adjustment is positive or negative and use assign-inc or assign-dec respectively. This is done by using ≤-compare in Table 3.3. We also need to show that the adjustment is valid for each case. Please refer to Appendix [TBD] for the full implementation of the function use-temp.

Compilation for closed program

To define a compilation function for closed programs in the source language, we use the denotational semantics of terms and fill in the initial stack descriptor ($\langle 0, 0 \rangle$), the empty context (unit), a trivial proof for stack descriptor being not less than itself ($\leq_s$-refl) and the continuation representing the last instruction (stop):

```
compile-closed : · ⊢ comm → I ⟨ 0 , 0 ⟩
compile-closed t = ⟦ t ⟧ ⟨ 0 , 0 ⟩ unit ≤ₛ-refl stop
```

[4] The type of β here is actually a bit different to integer continuation I defined, as it only accepts simple right-hand sides instead of right-hand sides. Reynolds gave a wrong definition of the type of this term. I have corrected it in my Agda implementation.

Chapter 4

Evaluation

Contents

This chapter evaluates the implementation with integration tests and a feature checklist and then discusses the success criteria of the project.

4.1 Tests

The best way to evaluate a formalised compiler project is to formalise compiler correctness, which requires the full definition of the operational semantics of the source language and the target language, and then prove that for any program e in the source language and the program e' it compiles to in the target language, the value v' that e' evaluates to is also the compilation of the value v that e evaluates to. Due to time limitations, this is beyond the scope of this project. Instead, I used a set of integration tests that covers all features of the source language and the target language to evaluate the correctness of the compiler. All of the test cases have been passed successfully.

4.1.1 Test cases

Each test case is an Agda program defined with the follow syntax, where Skip is replaced with any closed program in the source language and stop is replaced with the expected compiled result in the target language:

```
term-0 : · ⊢ comm
term-0 = Skip -- source term
result-0 = compile-closed term-0

test-0 : result-0 ≡ stop -- target term
test-0 = refl
```

Figure 4.1: Test 0 as an example of test cases

where definition of the compile-closed function is specified in §3.6.5. Each test case type checks only if both the source term and target term are well-typed, and the compiled result of the source term is definitionally equal to the target term.

There are special tests to check that some expressions in the source language compiles to the same expression in the target language. Such test is defined as follows:

```
test-3′ : result-3′ ≡ result-3
test-3′ = refl
```

Figure 4.2: Test 3′ as an example for checking equivalent compiled results

The full test cases are available in the appendix [TBD]. Here is a summary of the test cases:

Test case	Source term
Test 0	skip
Test 1	$x := 2$
Test 2	$x := (\lambda a.\ a)\ 4$
Test 3	$x := (\lambda a.\ (\lambda b.\ a + b)\ 2)\ 3$
Test 3′	$x := 3 + 2$
Test 4	$x := -3;\ y := (\lambda a.x)\ 2$
Test 5	$x := 2;\ x := x + 1$
Test 5′	$x := 2;$ skip$;\ x := x + 1$
Test 6	$x := 2;\ x := -x + 1$
Test 7	$x := (\lambda a.\ -a + 1)\ 2$

Table 4.1: Test cases

where **Test 3** and **Test 3′**, **Test 5** and **Test 5′** compile to the same target term.

Test 3 and **Test 3′** are equivalent in terms of compilation because the compilation directly substitutes the value in corresponding lambda terms.

Test 5 and **Test 5′** are equivalent in terms of compilation because skip is a no-op in the source language.

4.1.2 Feature checklist

Traditional unit tests are not suitable for this project, as many typing judgement rules cannot be tested in isolation. For example, testing subtyping rule (Figure 3.5) or any of the integer expression typing rules (Figure 3.9) inherently requires embedding them in a well-typed command (NewVar in Figure 3.8), as the compiler only accepts complete commands.

Instead, I employed integration testing with a feature checklist, ensuring comprehensive coverage by validating all typing rules and targeting edge cases.

Feature	Typing judgement	Specification	Test cases
Variable	Var	-	1, 2, 3, 3′, 4, 5, 5′, 6, 7
Subtyping	Sub	intvar $\leq:$ intacc	1, 2, 3, 3′, 4, 5, 5′, 6, 7
		intvar $\leq:$ intexp	4, 5, 5′, 6
Lambda abstraction	Lambda	-	2, 3, 4, 7
		Multiple variables	3
		Free variables	4
	App	-	2, 3, 4, 7
Command	Skip	-	0, 5′
	Seq	-	4, 5, 5′, 6
		Multiple sequencing	5′
	NewVar	-	1, 2, 3, 3′, 4, 5, 5′, 6, 7
		Multiple variables	4
	Assign	-	1, 2, 3, 3′, 4, 5, 5′, 6, 7
		Self-referential assignments	5, 5′, 6
Integer expression	Lit	Natural number	1, 3, 3′, 4, 5, 6, 7
		Negative number	2, 4
	Neg	-	5, 5′, 6, 7
	Plus	-	3, 3′, 6, 7
	-	Multiple expressions	6, 7

Table 4.2: Feature checklist

The specifications that include multiple variables, free variables and self-referential assignments ensure that the translation of the Debrujin index of the context works correctly under edge cases.

The specification of multiple expressions in integer expression is necessary because the target language does not support doing multiple integer operations in a single instruction, as specified in Figure 3.10. As a result, special tests are required to check that the compiler correctly translates integer expressions with multiple operations into a continuation of single operations defined in the target language.

The feature checklist covers all the typing rules in the source language, including edge cases including multiple variables, free variables and self-referential assignments. Additionally, I account for target-language-specific constraints, ensuring multiple operations are translated into simple instructions with continuation. The tests cover all the cases in the feature checklist, ensuring that the compiler generates expected target code under all circumstances.

4.2 Extensibility

This project is inherently extensible, as all of the components are defined as terms and types in Agda. The compiler can be extended to support more features by adding new terms in the form of type judgement rules in the source language, adding any new instructions in the target language if needed, and write corresponding compilation rules.

4.3 Success criteria

The project has met the success criteria as follows:

- A formalisation of the source language has been given in Agda, which includes the syntax, typing rules and operational semantics.
- A formalisation of the target language has been given in Agda, which includes all instructions from the instruction set. Rigorous definitions of the operation of the stack descriptors are given, ensuring correctness and preserving the mathematical properties of the stack descriptors.
- A formalisation of the compiler has been given in Agda, which includes the denotational semantics of the source language and a term for compilation.
- Integration tests have been given in Agda, which cover all the case in the feature checklist. The tests ensure that the compiler generates expected target code under all circumstances.

The following are the extension criteria that are met:

- A customised library have been developed, which includes rigorous definitions of subtraction of natural numbers and corresponding properties. Two approaches to rigorous definition of subtraction has been implemented and compared.
- Operational semantics of the source language has been defined with renaming, substitution and reduction rules.
- Complicated functions defined for translating expressions with multiple operations in the source language to a continuation of simple instructions in the target language.

Chapter 5

Conclusion

Contents

5.1 Results

This project has successfully implemented a compiler from a source language (STLC with store) to a target language (stack machine) in Agda. The compiler is defined as a functor from the source language to the target language, which is directly generated from the denotational semantics of the source language. The compiler validates and refines Reynolds' theory [6], offers a concrete example of how category-theoretic semantics can generate intermediate code.

5.2 Lessons learned

The project has been quite challenging as I have to learn both category theory and Agda from scratch and apply them to implementation. The implementation process revealed unexpected gaps between Reynolds' theoretical framework and the computational realisation: while his denotational semantics provided core definitions, actualising them in Agda required formalising numbers, their operations, and properties and diagnosing and resolving type mismatches and ambiguities in the original paper [6]. These hurdles highlighted how even meticulously defined theories can be complicated when translated to working systems.

In terms of planning the project, the project has deviated slightly from the original timetable in the project proposal due to differences between the anticipated structure of the project and its actual development process. When I was drafting the proposal, I expected the workflow to be similar to a standard compiler, so I divided the work into components including basic instructions, variable declarations and conditionals, allocating two weeks for each. In hindsight, these components are similar term definitions, while the most difficult task was to spot problems with the original paper and to prove properties about customised subtraction of natural numbers.

Overall, the project has been a valuable learning experience, and I have gained a deeper understanding of both category theory and Agda and an exciting glimpse of how the correspondences between logic, type theory and category theory can be used to implement a compiler.

5.3 Future work

The implementation of the compiler can be extended in the following ways:

- The compiler can be extended to support more features from Reynolds' paper [6], such as subroutines and iterations.
- The operational semantics of the target language can be defined, which can be used, along with the operational semantics of the source language, to prove the correctness of the compiler.
- Reynolds pointed out that the requirement of instructions being natural transformation must be relaxed to allow for instruction sequences that differ operationally but are equivalent in terms of denotational semantics [6, Ch. 6]. Extra research can be done to develop 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).

Bibliography

[1] R. Loader, *Notes on Simply Typed Lambda Calculus*, 1996. [Online]. Available: https://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/.

[2] J. Lambek, "Cartesian closed categories and typed λ-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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] "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.

[12] 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.

[13] 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/.

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

[15] 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.

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

[17] 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.

[18] 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.

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

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

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

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

[23] *Visual Studio Code*, Accessed: 2025-04-15. [Online]. Available: https://code.visualstudio.com/.

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

[25] *Windows Subsystem for Linux (WSL)*, Accessed: 2025-04-15. [Online]. Available: https://ubuntu.com/desktop/wsl.

[26] N. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem**reprinted from: Indagationes math, 34, 5, p. 381-392, by courtesy of the koninklijke nederlandse akademie van wetenschappen, amsterdam.," in *Selected Papers on Automath*, ser. Studies in Logic and the Foundations of Mathematics, R. Nederpelt, J. Geuvers, and R. de Vrijer, Eds., vol. 133, Elsevier, 1994, pp. 375–388. DOI: https://doi.org/10.1016/S0049-237X(08)70216-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0049237X08702167.

Appendix A

Operational Semantics of the source language

The operational semantics of the source language is defined with renaming, substitution and reduction rules. The implementation is as follows:

```
data Value : ∀{Γ A} → Γ ⊢ A → Set where
  V-Lambda : ∀{Γ A B} {F : Γ , A ⊢ B} → Value (Lambda {Γ} F)
  V-Lit : ∀{Γ} {i : ℤ} → Value (Lit {Γ} i)
  V-Skip : ∀{Γ} → Value (Skip {Γ})

-- Renaming
ext : ∀{Γ Δ} → (∀{A} → A ∈ Γ → A ∈ Δ)
             → (∀{A B} → B ∈ Γ , A → B ∈ Δ , A)
ext ρ Zero = Zero
ext ρ (Suc x) = Suc (ρ x)

rename : ∀{Γ Δ} → (∀{A} → A ∈ Γ → A ∈ Δ)
                → (∀{A} → Γ ⊢ A → Δ ⊢ A)
rename ρ (Var A∈Γ) = Var (ρ A∈Γ)
rename ρ (Lambda Γ,A⊢B) = Lambda (rename (ext ρ) Γ,A⊢B)
rename ρ (Sub Γ⊢A A≤:B) = Sub (rename ρ Γ⊢A) A≤:B
rename ρ (App Γ⊢A Γ⊢B) = App (rename ρ Γ⊢A) (rename ρ Γ⊢B)
rename ρ Skip = Skip
rename ρ (Seq Γ⊢c₁ Γ⊢c₂) = Seq (rename ρ Γ⊢c₁) (rename ρ Γ⊢c₂)
rename ρ (NewVar Γ⊢c) = NewVar (rename (ext ρ) Γ⊢c)
rename ρ (Assign Γ⊢i Γ⊢e) = Assign (rename ρ Γ⊢i) (rename ρ Γ⊢e)
rename ρ (Lit Γ⊢i) = Lit Γ⊢i
rename ρ (Neg Γ⊢i) = Neg (rename ρ Γ⊢i)
rename ρ (Plus Γ⊢i₁ Γ⊢i₂) = Plus (rename ρ Γ⊢i₁) (rename ρ Γ⊢i₂)
```

Figure A.1: Operational semantics of the source language (Part 1)

```
-- Simultaneous substitution
exts : ∀{Γ Δ} → (∀{A} → A ∈ Γ → Δ ⊢ A)
               → (∀{A B} → B ∈ Γ , A → Δ , A ⊢ B)
exts σ Zero = Var Zero
exts σ (Suc x) = rename Suc (σ x)

subst : ∀{Γ Δ} → (∀{A} → A ∈ Γ → Δ ⊢ A)
               → (∀{A} → Γ ⊢ A → Δ ⊢ A)
subst σ (Var A∈Γ) = σ A∈Γ
subst σ (Sub Γ⊢A A≤:B) = Sub (subst σ Γ⊢A) A≤:B
subst σ (Lambda Γ,A⊢B) = Lambda (subst (exts σ) Γ,A⊢B)
subst σ (App Γ⊢A Γ⊢B) = App (subst σ Γ⊢A) (subst σ Γ⊢B)
subst σ Skip = Skip
subst σ (Seq Γ⊢c₁ Γ⊢c₂) = Seq (subst σ Γ⊢c₁) (subst σ Γ⊢c₂)
subst σ (NewVar Γ⊢c) = NewVar (subst (exts σ) Γ⊢c)
subst σ (Assign Γ⊢i Γ⊢e) = Assign (subst σ Γ⊢i) (subst σ Γ⊢e)
subst σ (Lit Γ⊢i) = Lit Γ⊢i
subst σ (Neg Γ⊢i) = Neg (subst σ Γ⊢i)
subst σ (Plus Γ⊢i₁ Γ⊢i₂) = Plus (subst σ Γ⊢i₁) (subst σ Γ⊢i₂)

-- Single substitution
_[_] : ∀{Γ A B} → Γ , B ⊢ A → Γ ⊢ B → Γ ⊢ A
_[_] {Γ} {A} {B} N M = subst {Γ , B} {Γ} σ {A} N
  where
  σ : ∀ {A} → A ∈ Γ , B → Γ ⊢ A
  σ Zero = M
  σ (Suc x) = Var x

-- Reduction
data _⟶_ : ∀{Γ A} → (Γ ⊢ A) → (Γ ⊢ A) → Set where
  App-cong₁ : ∀{Γ A B} {F F' : Γ ⊢ A ⇒ B} {E : Γ ⊢ A}
                     → F ⟶ F' → App F E ⟶ App F' E
  App-cong₂ : ∀{Γ A B} {V : Γ ⊢ A ⇒ B} {E E' : Γ ⊢ A}
                     → Value V → E ⟶ E' → App V E ⟶ App V E'
  Lambda-β : ∀{Γ A B} {F : Γ , A ⊢ B} {V : Γ ⊢ A}
                     → Value V → App (Lambda F) V ⟶ F [ V ]
```

Figure A.2: Operational semantics of the source language (Part 2)

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/