Using functor categories to generate intermediate code with Agda

Jack Gao

April 23, 2025

Contents

Chapter 1

Introduction

Contents

An Algol-like language is a typed lambda calculus with store. In this dissertation, the source language is an Algol-like language with the following primitive types:

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

and the set Θ of types is defined as follows:

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

[Considering adding an example of Algol-like language here]

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:

- $\langle \mathrm{L}_{sd} \rangle$: lefthand sides
- $\langle \mathrm{S}_{sd} \rangle$: Simple righthand sides
- $\langle \mathrm{R}_{sd} \rangle$: righthand sides
- $\langle \mathrm{I}_{sd} \rangle$: instruction sequences

The grammer of the target language is specified in Chapter 3. [Add exact section/subsection here]

This dissertation presents an implementation of a compiler from the source language to the target language with Agda [1]. This implementation is based on the work of Reynolds [2], who presented a denotational semantics of Algol-like languages in the form of a presheaf category over stack descriptors. The compiler is implemented as a functor from the source language to the target language. The implementation is verified with Agda's type system, which ensures that the generated code is well-typed and adheres to the semantics of both the source and target languages. This implementation proves and refines Reynolds' work, providing a practical example of how to use functor categories to generate intermediate code.

1.1 Motivation and related work

The denotational semantics of Algol-like languages can be structured as a presheaf category over stack descriptors, which has been shown by Reynolds [3] and Oles [4] [5]. By interpreting the source language into this category, where objects of the category represent instruction sequences parameterised by stack layouts, the semantic model directly yields a compiler. The mathematical structure of the compiler has been specified by Reynolds in his paper "Using Functor Categories to Generate Intermediate Code" [2].

This project is motivated and guided by the following:

Motivation I. Implementation of the compiler

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

Motivation II. Formal verification of the compiler

The terms in Reynolds' work are also written by hand. Terms are complicated and error-prone, and it is difficult to verify the correctness of the terms. We aim to provide a formalisation of the terms in a proof assistant to verify the correctness of the terms.

Motivation III. Trend of verified compilers

The rise of verified compilers including CompCert [6], CakeML [7] reflects a broader trend toward trustworthy systems, where correctness proofs replace testing for critical guarantees. Like Lean 4 [8], we leverage dependent types to internalise the verification of correctness of terms.

1.2 Language choice: Agda's advantages

Agda [1] is a dependently typed programming language and proof assistant. Agda captures the source language's intrinsic sytax with indexed families, which contains only well-typed terms. Therefore, it focuses on the correct programs and rules out the ill-typed nonsensical inputs.

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

[Do I need a table for comparing other proof assistants and Agda?]

1.3 Contributions

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

Motivation I. Implementation of the compiler

We implemented the compiler from the source language to the target language in Agda.

Motivation II. Formal verification of the compiler

We formalised the terms in the source language and target language in Agda, and proved that the compiler is a functor from the source language to the target language.

Motivation III. Trend of verified compilers

This implementation follows the trend of verified compilers, and provides a practical example of how to use functor categories to generate intermediate code.

Chapter 2

Preparation

Contents

2.1 Starting Point

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

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

2.2 Category theory background

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

Notation in category theory is similar to that in set theory and type theory. For example, $A, B, \ldots$ are used to denote objects, sets, or types, and arrows are used to denote morphisms or functions (e.g. $A \to B$). There is a deeper connection between category theory and type theory. Curry-Howard correspondence [15] states that there is a correspondence between logic and type theory, where propositions correspond to types and proofs correspond to terms. This correspondence was later extended by Joachim Lambek, who showed that cartesian closed categories provide a natural semantic setting for the simply typed lambda calculus [16]. Under this perspective, types become objects in a category, and programs correspond to morphisms. We will further explore this connection in §**??**.

The following is a brief introduction to the basic concepts of category theory, which is based on the work of Leinster [13], Riehl [17], and the lecture notes of Part II Category Theory by Andrew Pitts and Marcelo Fiore [18].

2.2.1 Category

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

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

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

  called the composition of morphisms;

satisfying the following properties:

- **(Unit Laws)** for all $X, Y \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$, we have:
$$\mathbf{id}_Y \circ f = f = f \circ \mathbf{id}_X \tag{2.1}$$
- **(Associativity Law)** for all $X, Y, Z, W \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$, $g \in \mathcal{C}(Y, Z)$, $h \in \mathcal{C}(Z, W)$, we have:
$$h \circ (g \circ f) = (h \circ g) \circ f \tag{2.2}$$

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

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

Opposite category

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

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

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

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

Commutative diagrams

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

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

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

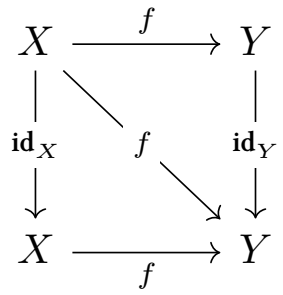


Figure 2.1: Commutative diagram for Unit Laws

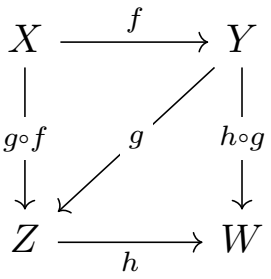
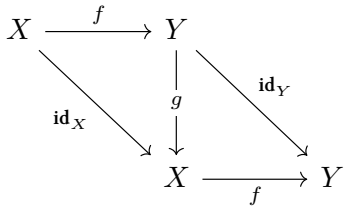


Figure 2.2: Commutative diagram for Associativity Law

2.2.2 Isomorphism

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



(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 $X \cong Y$.

[Considering adding examples of isomorphism]

2.2.3 Terminal object

Definition (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 $f : X \to T$.

[Considering adding examples of terminal object]

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

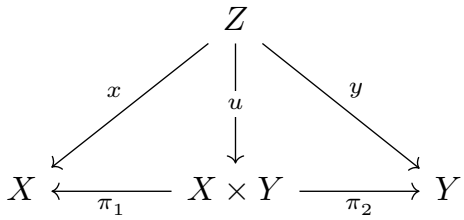
- if T and T' are both terminal objects in $\mathcal{C}$, then there exists a unique isomorphism $f : T \to T'$.
- if T is a terminal object in $\mathcal{C}$ and $T \cong T'$, then T' is also a terminal object in $\mathcal{C}$.

2.2.4 Binary product

Definition (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

$$\langle x, y \rangle : Z \to X \times Y$$

where $x = \pi_1 \circ u$ and $y = \pi_2 \circ u$.

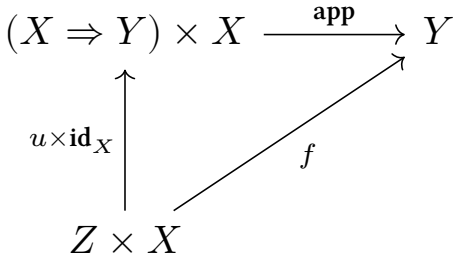
It can be shown that the binary product is unique up to (unique) isomorphism.

2.2.5 Exponential

Definition (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.

2.2.6 Cartesian closed category

Definition (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.

2.2.7 Functor

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

- a function
$$\begin{aligned} \mathbf{obj}(\mathcal{C}) &\to \mathbf{obj}(\mathcal{D}) \\ X &\mapsto F(X) \end{aligned}$$
- for each $X, Y \in \mathbf{obj}(\mathcal{C})$, a function
$$\begin{aligned} \mathcal{C}(X, Y) &\to \mathcal{D}(F(X), F(Y)) \\ f &\mapsto F(f) \end{aligned}$$

satisfying the following properties:

- for all $X, Y \in \mathbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$, we have:
$$F(\mathbf{id}_X) = \mathbf{id}_{F(X)} \tag{2.6}$$
- 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:
$$F(g \circ f) = F(g) \circ F(f) \tag{2.7}$$

[Considering adding examples of functors: e.g. free functor, forgetful functor, etc.]

2.2.8 Natural transformation

Definition (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

$$\begin{array}{ccc} F(X) & \xrightarrow{\theta_X} & G(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow[\theta_Y]{} & G(Y) \end{array} \tag{2.8}$$

commutes in $\mathcal{D}$, i.e. the following equation holds:

$$G(f) \circ \theta_X = \theta_Y \circ F(f) \tag{2.9}$$

[Considering adding examples of natural transformations]

2.2.9 Functor category

Definition (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 $X \in \mathbf{obj}(\mathcal{C})$.

2.2.10 Presheaf category

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

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

Definition (Presheaf category). Given a category $\mathcal{C}$, the *presheaf category* $\mathcal{P}(\mathcal{C})$ is the functor category $\mathcal{S}et^{\mathcal{C}^{\mathbf{op}}}$.

- the objects of $\mathcal{P}(\mathcal{C})$ 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 $\mathcal{P}(\mathcal{C})$ are all natural transformations $\theta : F \to G$.

2.2.11 Yoneda lemma

Definition (Yoneda functor). Given a category $\mathcal{C}$, the *Yoneda functor* $よ : \mathcal{C} \to \mathcal{S}et^{\mathcal{C}^{\mathbf{op}}}$ is defined as follows:

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

$$よ(X)(Y) = \mathcal{C}(Y, X) \tag{2.10}$$

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

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

$$\begin{aligned}(よ(f))_Z : \mathcal{C}(Z,Y) &\to \mathcal{C}(Z,X) \\ g &\mapsto g \circ f\end{aligned} \tag{2.11}$$

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

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

$$\mathcal{S}et^{\mathcal{C}^{\mathrm{op}}}(よ(X), F) \cong F(X) \tag{2.12}$$

2.2.12 Cartesian closed structure in presheaf categories

Theorem (Cartesian closed structure in presheaf categories). Given a small category $\mathcal{C}$, the presheaf category $\mathcal{S}et^{\mathcal{C}^{\mathrm{op}}}$ is a Cartesian closed category.

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

$$\begin{cases} \mathbf{1}(X) = \{*\} & \text{for all } X \in \mathbf{obj}(\mathcal{C}) \\ \mathbf{1}(f) = \mathbf{id}_{\{*\}} & \text{for all } f \in \mathcal{C}(X,Y) \end{cases} \tag{2.13}$$

- The binary product in $\mathcal{S}et^{\mathcal{C}^{\mathrm{op}}}$ is given by the product of functors, which is defined as follows:

$$\begin{aligned}(F \times G)(X) &= F(X) \times G(X) \\ (F \times G)(f) &= F(f) \times G(f)\end{aligned} \tag{2.14}$$

- The exponential in $\mathcal{S}et^{\mathcal{C}^{\mathrm{op}}}$ is given by the Yoneda lemma,

$$\begin{aligned}G^F(X) &= \mathcal{S}et^{\mathcal{C}^{\mathrm{op}}}(よ(X) \times F, G) \\ G^F(f)(\theta) &= \theta \circ (よ(f) \times \mathbf{id}_F) \quad \text{for all } \theta \in \mathcal{S}et^{\mathcal{C}^{\mathrm{op}}}(よ(Y) \times F, G)\end{aligned} \tag{2.15}$$

2.3 Agda

2.3.1 Basic datatypes and pattern matching

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

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

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

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

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

2.3.2 Dependent Types

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

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

In contrast, dependent types are in form of

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

or more generally

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

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

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

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

2.3.3 Curry-Howard-Lambek correspondence

As introduced in §??, Curry-Howard correspondence states that there is a correspondence between logic and type theory, and Lambek extended this correspondence to show that cartesian closed categories provide a natural semantic setting for the simply typed lambda calculus. They can be summarised as follows:

Logic	Type theory	Category theory
Proposition	Type	Object
Proof	Term	Morphism
Falsity	Empty type	Initial object
Truth	Unit type	Terminal object
Implication	Function type	Exponential object
Conjunction	Product type	Product
Disjunction	Sum type	Coproduct
Universal quantification	Dependent product type	
Existential quantification	Dependent sum type	

Table 2.1: Curry-Howard-Lambek correspondence

2.3.4 Equality, congruence and substitution

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.5 Proof Example

[decide exact example here]

2.3.6 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, however, I defined most of the basic data types and functions from scratch for the following two reasons:

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

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

tation.

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

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

1+1≡2 : 1 + 1 ≡ 2
1+1≡2 = refl
```

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

2.3.7 Interactive programming with holes

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

[Considering adding examples of holes and case split, see https://plfa.github.io/Naturals/]

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

2.4 Requirement Analysis

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

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

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

2.5 Tools Used

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.

2.6 Summary

- requirement Analysis
- tools and engineering approach

Chapter 3

Implementation

Chapter 4

Evaluation

Chapter 5

Conclusion

Bibliography

[1] 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.

[2] 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.

[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] 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.

[7] 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.

[8] L. d. Moura and S. Ullrich, "The lean 4 theorem prover and programming language," in *Automated Deduction – CADE 28*, A. Platzer and G. Sutcliffe, Eds., Cham: Springer International Publishing, 2021, pp. 625–635, ISBN: 978-3-030-79876-5.

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

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

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

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

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

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

[15] 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.

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

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

[18] 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.

[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.