Using functor categories to generate intermediate code with Agda

Jack Gao

April 21, 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.

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.

(Example of Agda)

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.

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

1.3 Contributions

Addressed the two motivations presented in 1.1 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

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;
- 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:

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

[Considering adding different notations for morphisms, composition, identity morphisms, etc.]

[Considering adding a diagram for the definition of category]

[Considering adding examples of categories]

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.

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

Figure 2.1: Commutative diagram for Unit Laws

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

Figure 2.2: Commutative diagram for Associativity Law

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

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:

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

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

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

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

[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 $\mathcal{C}$:

$$\begin{tikzcd}
 & Z \arrow[dl, "x"'] \arrow[d, "u"] \arrow[dr, "y"] & \\
X & X \times Y \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & Y
\end{tikzcd} \tag{2.4}$$

The unique morphism u is written as

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

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 $\mathcal{C}$:

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

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

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

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

$$\begin{tikzcd} F(X) \arrow[r, "\theta_X"] \arrow[d, "F(f)"'] & G(X) \arrow[d, "G(f)"] \\ F(Y) \arrow[r, "\theta_Y"'] & G(Y) \end{tikzcd} \tag{2.8}$$

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 Yoneda lemma

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

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

  $$y(X)(Y) = \mathcal{C}(Y, X)$$

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

- for each $X, Y \in \textbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$, $y(f)$ is the natural transformation $y(f) : y(X) \to y(Y)$ defined as:

  $$y(f)_Y = f \circ - : y(X)(Y) \to y(Y)(Y)$$

  for all $Y \in \textbf{obj}(\mathcal{C})$. In other words, $y(f)_Y$ is the function that takes an element $g \in y(X)(Y) = \mathcal{C}(Y, X)$ and returns the element $f \circ g \in y(Y)(Y) = \mathcal{C}(Y, Y)$. The function $y(f)_Y$ is a morphism in $\mathcal{C}^{\mathcal{C}^{op}}$ from $y(X)(Y)$ to $y(Y)(Y)$. In other words, $y(f)_Y$ is a morphism in $\mathcal{C}^{\mathcal{C}^{op}}$ from the functor $y(X)$ to the functor $y(Y)$. The morphism $y(f)_Y$ is defined for all $Y \in \textbf{obj}(\mathcal{C})$. In other words, $y(f)$ is a natural transformation from the functor $y(X)$ to the functor $y(Y)$.

Definition (Yoneda lemma). Given a category $\mathcal{C}$, the *Yoneda lemma* states that for any functor $F : \mathcal{C} \to \textbf{Set}$, there is a natural isomorphism:

$$\begin{aligned} \textbf{Nat}(y(X), F) &\cong F(X) \\ \theta &\mapsto \theta_X \end{aligned}$$

for all $X \in \textbf{obj}(\mathcal{C})$, where $\textbf{Nat}(y(X), F)$ is the set of natural transformations from the functor $y(X)$ to the functor F.

2.2.11 Presheaf

Definition (Presheaf). Given a category $\mathcal{C}$, a *presheaf* on $\mathcal{C}$ is a functor $F : \mathcal{C}^{op} \to \textbf{Set}$. 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 $X \in \textbf{obj}(\mathcal{C})$, $F(X)$ is a set;

- for each $X, Y \in \textbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$, $F(f)$ is a function $F(Y) \to F(X)$. In other words, $F(f)$ is a function that takes an element $g \in F(Y)$ and returns the element $f \circ g \in F(X)$. The function $F(f)$ is a morphism in $\mathcal{C}^{op}$ from $F(Y)$ to $F(X)$. In other words, $F(f)$ is a morphism in $\mathcal{C}^{op}$ from the functor $F(Y)$ to the functor $F(X)$. The morphism $F(f)$ is defined for all $X, Y \in \textbf{obj}(\mathcal{C})$ and $f \in \mathcal{C}(X, Y)$. In other words, $F(f)$ is a natural transformation from the functor $F(Y)$ to the functor $F(X)$.

2.2.12 Cartesian closed structure in presheaf categories

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.

2.3.2 Dependent Types

2.3.3 Curry-Howard-Lambek correspondence

2.3.4 Equality, congruence and substitution

2.3.5 Standard library

2.3.6 Interactive programming with holes

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

[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

2.5 Tools Used

Completing the project is an iterative process. I used Git [13] for version control, and work had been synchronised with a GitHub [14] repository for backup.

For the development environment, I tried both Emacs [15] and Visual Studio Code [16] with an agda-mode extension [17] on Windows Subsystem for Linux with Ubuntu [18] 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.

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

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

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

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

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

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

[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.