Automated Derivation of Correct Compilers

Sebastian Bue Bjørner
sebj@itu.dk

Advisor: Patrick Bahr
Submitted: September 2024
Stads Code: KISPECI1SE

IT UNIVERSITY OF COPENHAGEN

Abstract

Ensuring the correctness of a compiler is crucial for reliable software development, as even minor bugs can lead to significant issues in the software. One approach to achieving compiler correctness is by addressing the problem of calculating compilers, where the compiler, target language, and virtual machine are systematically derived from the source language's syntax and semantics. This thesis contributes to this line of approach by presenting a specialized tool, *CoCa* (*Co*mpiler *Ca*lculator). CoCa is designed to verify compiler calculations for source languages that include arithmetic expressions, exceptions and state. The tool integrates a lexical analyzer, parser, proof system, and term rewriting system to ensure the correctness of these calculations. Currently, CoCa focuses on verifying compiler calculations. However, the thesis describes opportunities for further development of CoCa, including facilitating the automation of compiler calculations and deriving definitions directly from these calculations.

Preface

With this thesis on CoCa, I am (almost) completing my 2-year Master's program in Software Design at the IT University of Copenhagen. This Master's in Software Design builds upon my Bachelor's degree in Sustainable Design from Aalborg University. Although both disciplines share the word "design", they have very few similarities.

Writing a thesis on compilers has been a fascinating endeavor, though it came with a steep learning curve. I would not have been able to complete this thesis without the exceptional guidance of my supervisor, Patrick Bahr. Thank you for your supervision and patience!

Contents

Contents iv

1 Introduction 1

2 Background 4
2.1 Compiler Correctness 5
2.1.1 Reasoning about Compiler Correctness 8
2.1.2 Calculating Compilers 19
2.2 Rewriting Systems 32
2.2.1 Abstract Reduction Systems 33
2.2.2 First-Order Term Rewriting Systems 35

3 Implementation 42
3.1 Conceptual Model of CoCa 43
3.1.1 Defining an Object Language with the Meta Language 44
3.1.2 Expanding the Meta Language for Compiler Calculations 45
3.1.3 Expressing Compiler Correctness with the Proof Language 46
3.1.4 Proof Steps and Their Relation to the Model 46
3.1.5 Finalizing the Proof 55
3.1.6 Overview of Commands in the Proof Language 55
3.2 Technical Model of CoCa 57
3.2.1 Technical Overview 57
3.2.2 Source Language 58
3.2.3 Proof System 68

4 Reflections and Limitations 75

5 Conclusion and Outlook on Future Work 77

References 78

Appendices 79
Appendix A - Exceptions 79
Appendix B - State 86
Appendix C - Lexical Program Structure 94
Appendix D - Context-Free Grammar 97

Chapter 1

Introduction

Compilers are fundamental tools in software development. Compilers translate high-level source code into lower-level target code, making the target code ready for execution on a specific platform. The correctness of a compiler is crucial, as even minor bugs can lead to significant issues in the software being developed. While testing is an essential part of the development of compilers, it is often insufficient to guarantee the absence of bugs, especially as the complexity of source languages increases.

Given the critical importance of compiler correctness, there has been considerable interest in formal techniques to ensure that compilers behave as intended. One approach involves calculating compilers directly from a formal specification, which inherently guarantees their correctness through the calculation process. This approach eliminates many of the uncertainties associated with manually defined compilers.

However, the process of calculating compilers is not without its challenges. The steps involved can be complex, and even with the assistance of advanced proof assistants like Coq or Agda, the task can be overwhelming. These tools, while powerful, are designed to be general-purpose, handling a wide range of proof scenarios. As a result, they often introduce unnecessary complexity when applied to specialized tasks, such as compiler calculations.

Recognizing this gap, the objective of this thesis is to create a tool that is specifically designed to address the needs of compiler calculations

involving stack-based virtual machines. This tool aims to:

- Check and verify compiler calculations.
- Facilitate the automation of compiler calculations.

This thesis presents the design and implementation of *CoCa* (*Co*mpiler *Ca*lculator), a specialized tool for compiler calculations. CoCa is capable of verifying compiler calculations with source languages that include arithmetic expressions, exceptions and state. While currently limited to verification, CoCa's underlying architecture lays the groundwork for future extensions, such as facilitating the automating of compiler calculations and deriving the implementation details of both the compiler and virtual machine from these calculations.

CoCa is developed in Haskell and comprises several components: A lexical analyzer that generates a stream of tokens from a source file, a parser that converts these tokens into an abstract syntax tree based on a defined grammar, a proof system that traverses the abstract syntax tree to verify the calculations and derive a set of predefined rules used as rewrite rules, and a term rewriting system that performs the underlying transformations necessary for the verification process based on these rules.

This thesis is structured to guide the reader through the foundational concepts and the practical implementation of CoCa. Chapter 2 offers the necessary background by introducing key concepts required for understanding compiler correctness, the approach of calculating compilers and rewriting systems.

Following this theoretical foundation, Chapter 3 outlines the design and implementation of CoCa from both conceptual and technical perspectives. It begins by presenting the conceptual model, which offers a high-level understanding of CoCa. The chapter then describes the technical details of CoCa and how its components – the lexical analyzer, parser, term rewriting system and proof system – work together to achieve the tool's objectives.

Chapter 4 reflects on the results, limitations, and potential improvements of CoCa, providing a critical assessment of its effectiveness and

identifying areas for future enhancement.

Finally, Chapter 5 concludes the thesis by summarizing the key contributions of the work and proposing directions for future developments of CoCa.

Chapter 2

Background

This chapter delves into the foundational concepts crucial for understanding compiler correctness and rewriting systems. The first half is dedicated to compiler correctness, with a specific focus on how compilers can be derived through a calculation process. Following this, the chapter provides a detailed introduction to rewriting systems, with an emphasis on First-Order Term Rewriting Systems. These topics provide the foundation to the theoretical concepts that will be used to understand the implementation of CoCa discussed in the subsequent chapter.

2.1 Compiler Correctness

The problem of compiler correctness was first formally addressed in 1967 by John McCarthy and James Painter. They proved the correctness of a compiler that translates arithmetic expressions into target code for a register-based machine. They provided the syntax and semantics of both the source and target languages, a compiler and a specification describing the correctness of the compiler. Using this specification, they proceeded to verify the correctness of the compiler through induction on the expressions being compiled (McCarthy & Painter, 1967).

In 1973, F. Lockwood Morris further refined the compiler correctness problem by emphasizing that a compiler must be *semantics-preserving*. He proposed that a proof of compiler correctness should demonstrate that the semantics of the compiled program preserve the meaning of the source program, which he illustrated using a commutative diagram, as shown in Figure 2.1 (Lockwood, 1973).

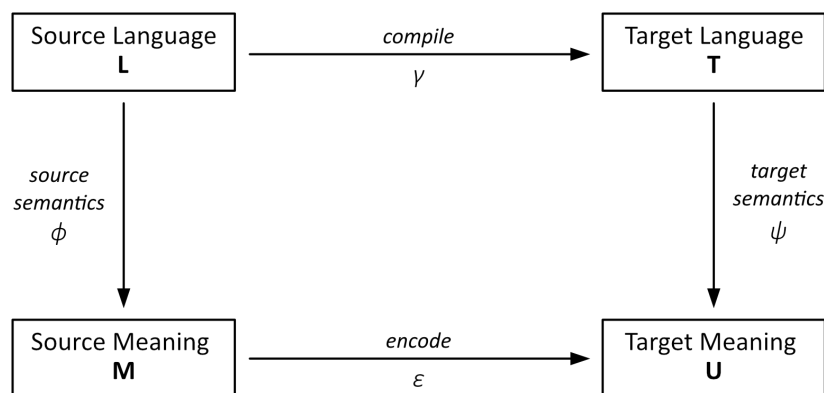

Figure 2.1: Fundamental commutative diagram illustrating compiler correctness.

The diagram captures the relationships between the source language (L), the target language (T), the source meaning (M), and the target meaning (U). The arrows represents mappings such that:

- $\phi : L \rightarrow M$ is a function that maps programs in the source language to their meanings.
- $\gamma : L \rightarrow T$ is a function that translates programs written in the source language to the target language.

- $\psi : T \rightarrow U$ is a function that maps programs in the target language to their meanings.
- $\epsilon : M \rightarrow U$ is a function that ensures that the target program's meaning corresponds to the source program's meaning.

Morris emphasized that these semantics should be formalized as functions that assign definite mathematical objects, or "meanings", to programs (Lockwood, 1973). This formalization can be achieved using denotational semantics, a framework described by Scott and Strachey (1971), where "The values of expressions are determined in such a way that the values of a whole expression depends functionally on the values of its parts."

For the diagram to commute, the composition of applying γ to a source program, $(\cdot)$, followed by ψ to the resulting target program must be equivalent to applying ϕ to the source program and then ϵ to the result. This results in the following equation:

$$\psi \circ \gamma(\cdot) = \epsilon \circ \phi(\cdot)$$

This equation states that translating a source program and then interpreting its meaning should yield the same result as interpreting the source program directly and then encode its meaning. The equation serves as a fundamental criterion for ensuring that a compiler correctly preserves the semantics of the source program.

The Problem of Verifying Compiler Correctness

Verifying the correctness of a compiler involves addressing the problem of ensuring that the compiler is semantics-preserving. Given a source language L, a target language T, their semantics ϕ and ψ, and a compiler γ, one constructs an encoder ϵ and shows that the diagram in Figure 2.1 commutes.

The Problem of Calculating Compilers

Another approach to ensure compiler correctness is through *calculating compiler*. The problem of calculating compilers differs from that of

verifying compiler correctness. When calculating compilers, one begins with the source language L and its semantics ϕ, and then systematically derive the compiler γ, the target language T, and the target semantics ψ. In this approach, the encode function ϵ – which plays a critical role in traditional verification by ensuring that the source program's meaning corresponds to the target program's meaning – is inherently incorporated within the construction process. This integration eliminates the need for a separate verification step, as development and proof proceed hand in hand, making the compiler *correct by construction* (Backhouse, 2003).

Bahr and Hutton (2015) presents an approach to the problem of calculating compilers targeting stack-based virtual machines. Their technique uses standard equational reasoning, avoiding the need for more advanced techniques such as continuations and defunctionalisation, which have been previously explored and used to solve the problem of calculating compilers. According to them, their method is versatile and can be applied to calculating compilers for various language features, including arithmetic expressions, exceptions, state, loops, lambda calculi, non-determinism, and interrupts.

The approach begins by defining the source language syntax and semantics, along with equations capturing compiler correctness. The compiler correctness equations are then used directly as specifications to derive the implementation of the compiler, target language and virtual machine by *constructive induction*. Bahr and Hutton (2015) summarizes the approach in a three-step process:

1. Define an evaluation function in a compositional manner;
2. Define equations that specify the correctness of the compiler;
3. Calculate definitions that satisfy these specifications.

In the next section, I will demonstrate how to reason about specifying equations that captures the correctness of a compiler for various instantiations of the diagram from Figure 2.1. Following this, some of the correctness equations will serve as specifications to calculate compilers for source languages featuring extended arithmetic expressions, exceptions, and state. The calculations will be based on the approach presented by Bahr and Hutton (2015) and closely resemble those in Bahr

and Hutton (2015), but a solid understanding of this approach will be essential for the rest of the thesis.

2.1.1 Reasoning about Compiler Correctness

Let's first consider a compiler, which translates source language, $Expr$, to target language, $Code$. We can encapsulate the compilation through a function $compile$:

$$compile :: Expr \to Code$$

The semantics for the source language is given through an evaluation function $eval$, that maps the source language to their meaning M:

$$eval :: Expr \to M$$

Likewise, the semantics for the target language is given through a function $exec$, which maps the target language to their meaning U:

$$exec :: Code \to U$$

As a starting point, let's instantiate the semantic domains M and U to the set of integers, such that:

$$eval :: Expr \to Int$$
$$exec :: Code \to Int$$

The relationship between the source language $Expr$, the target language $Code$, the source meaning Int, and the target meaning Int is illustrated on the diagram in Figure 2.2.

The correctness of the compiler is captured in an equation that ensures the commutativity of the diagram. In its most general form, the equation becomes:

$$\lambda e.\ exec\ (compile\ e) = \lambda e.\ encode\ (eval\ e)$$

Where:

- e is an arbitrary expression in the source language.

- *encode* is a function that maps the result of the evaluation in the source language to the corresponding result in the target language domain.

Since the source and target languages share the same semantic domain, *encode* would be the identity function, defined as:

$$encode \equiv \lambda a.\ a$$

The right-hand side of the correctness equation undergoes a transformation through function application and beta reduction:

$$\begin{aligned} & \lambda e.\ encode\ (eval\ e) \\ = & \quad \{ \text{ apply definition of } encode \ \} \\ & \lambda e.\ (\lambda a.a)\ (eval\ e) \\ = & \quad \{ \text{ beta reduction } \} \\ & \lambda e.\ eval\ e \end{aligned}$$

Thus, the correctness equation for the compiler is reduced to:

$$\lambda e.\ exec\ (compile\ e) = \lambda e.\ eval\ e$$

Finally, by extensionality, the correctness equation can be expressed as:

$$exec\ (compile\ e) = eval\ e$$

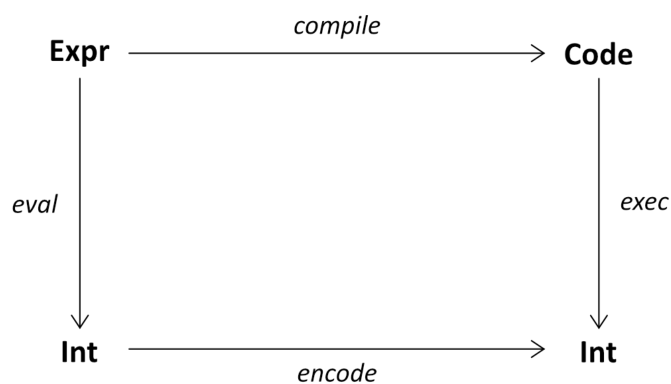

Figure 2.2: Commutative diagram illustrating compiler correctness with both semantic domains instantiated to integers.

This equation states that executing the compiled expression e produces the same result as directly evaluating e.

In this simplified model, *exec* acts as an evaluator for the target language, directly evaluating the compiled code, much like the function *eval* does for the source language. However, this representation does not reflect a typical virtual machine, which often relies on an execution context – provided by components such as a stack or registers – to tasks like handling intermediate results and control flow during execution.

Extending 'exec' to a Stack-Based Virtual Machine

To transform *exec* into a more traditional virtual machine, we can introduce a stack. The stack is defined as a stack of integers following the Last In, First Out (LIFO) principle:

$$\textbf{type}\ \mathit{Stack} = [\mathit{Int}]$$

The execution function *exec* is then instantiated to take an initial (empty) stack as an additional input, giving the following signature:

$$\mathit{exec} :: \mathit{Code} \rightarrow \mathit{Stack} \rightarrow \mathit{Stack}$$

This gives us the diagram in Figure 2.3.

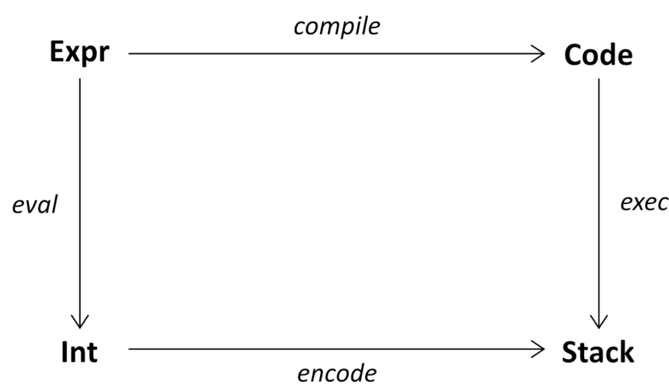

Figure 2.3: Commutative diagram illustrating compiler correctness with the semantic domain of the target language instantiated to a Stack type.

With this instantiation, the correctness equation for the compiler becomes:

$$\lambda e.\ exec\ (compile\ e)\ [\,] = \lambda e.\ encode\ (eval\ e) \tag{2.1}$$

The function *encode* should now map the semantic domain of the source language, Int, to the semantic domain of the target language, $Stack$. This results in the following definition of *encode*:

$$encode \equiv \lambda n.\ [n]$$

Again, this definition can be used to simplify the right-hand side of equation 2.1:

$$\begin{aligned} & \lambda e.\ encode\ (eval\ e) \\ = \quad & \{ \text{ apply definition of } encode \ \} \\ & \lambda e.\ (\lambda n.\ [n])\ (eval\ e) \\ = \quad & \{ \text{ beta reduction } \} \\ & \lambda e.\ [eval\ e] \end{aligned}$$

For an empty initial stack, the correctness equation for the compiler simplifies to:

$$\lambda e.\ exec\ (compile\ e)\ [\,] = \lambda e.\ [eval\ e]$$

Then, by applying function extensionality, we obtain the following correctness equation:

$$exec\ (compile\ e)\ [\,] = [eval\ e]$$

This equation states that compiling an expression e and executing the resulting code together with an empty initial stack results in a stack containing the evaluation of expression e.

Generalizing from the Empty Initial Stack to an Arbitrary Initial Stack

In the basic stack-based model, the virtual machine starts with an empty initial stack when executing the compiled code. However, this approach can be generalized to handle any arbitrary initial stack.

To achieve this, we can instantiate the semantic domain of the target language to a function that takes a stack and returns a final stack. The function *exec* now has the following type signature:

$exec :: Code \to (Stack \to Stack)$

In this version, the *exec* function operates on a stack by first taking the compiled code and then returning a function that, when provided with a stack, will execute the code on that stack to produce a final stack.

This instantiation of the semantic domain of the target language is illustrated on the diagram in Figure 2.4.

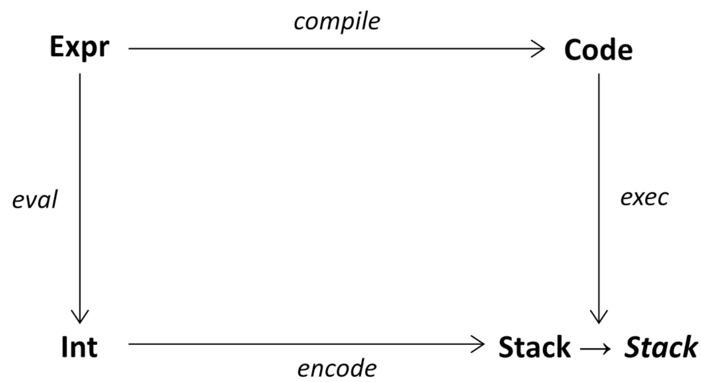

Figure 2.4: Commutative diagram illustrating compiler correctness with the semantic domain of the target language instantiated to a function from Stack to Stack.

As a result, the correctness equation for the compiler becomes:

$$\lambda e.\ exec\ (compile\ e) = \lambda e.\ encode\ (eval\ e) \tag{2.2}$$

The function *encode* should now take an integer as an argument and produce a function that takes a stack and returns a new stack:

$encode \ \equiv \ \lambda n.\lambda s.\ n : s$

Once again, the definition of *encode* can be used to simplify the right-hand side of equation 2.2, which reduces to:

$\lambda e.\lambda s.\ (eval\ e) : s$

Thus, the correctness equation for the compiler becomes:

$$\lambda e.\ exec\ (compile\ e) = \lambda e.\lambda s.\ (eval\ e) : s$$

Again, the correctness equation can be given more concisely by applying function extensionality and then applying the argument s to both sides, followed by beta reduction:

$$exec\ (compile\ e)\ s = eval\ e : s$$

This equation states that compiling an expression e and executing the compiled code on an arbitrary initial stack s results in the evaluation of e being pushed on top of the stack s.

Introducing a Code Continuation

The correctness equation can be further generalized by the introduction of a code continuation into the compilation process. The continuation represents remaining code that should be executed after the compiled code has been executed. By including a continuation, the compiler can ensure that not only the current expression is correctly compiled and executed, but also that the subsequent code is properly handled.

To include the continuation into the compilation process, let's first specify the target language *Code*:

```
type Code = [Op]
data Op
```

The target language is represented with the intermediate representation *Code* that comprises a sequence of target language operations, Op. This way of representing target code is illustrated by Hutton (2016, Section 16.7).

Since the compilation now requires compiling an expression and combining it with the code continuation, we introduce a function f that takes a pair of code sequences, resulting in target code that correctly incorporates both the compiled expression and the continuation:

$$f :: (Code, Code) \rightarrow Code$$

The diagram in Figure 2.5 illustrates the inclusion of the additional code continuation and the generalized compilation process.

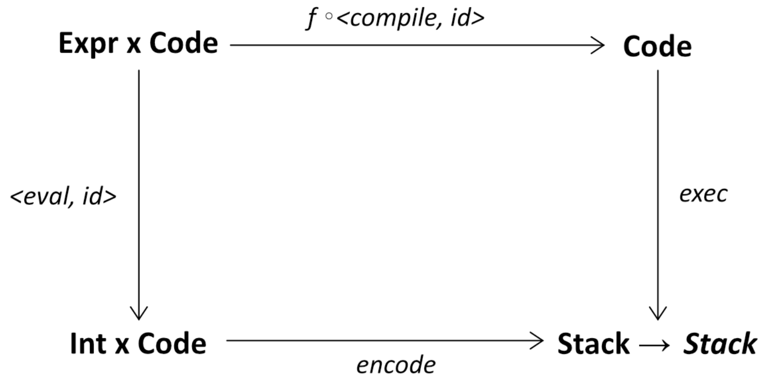

Figure 2.5: Commutative diagram illustrating compiler correctness with the inclusion of a code continuation.

The commutativity of the diagram provides the following compiler correctness equation:

$$\lambda(e,c).\ exec\ (f\ (compile\ e,\ id\ c)) = \lambda(e,c).\ encode\ (eval\ e,\ id\ c) \tag{2.3}$$

The compilation process combines the compiled code with the code continuation. This leads to the following definition for f:

$$f \ \equiv\ \lambda(x,y).\ x \mathbin{++} y$$

This definition allows us to simplify the left-hand side of equation 2.3 as follows:

$$\begin{aligned}
& \lambda(e,c).\ exec\ (f\ (compile\ e,\ id\ c)) \\
= & \quad \{ \text{ apply definition of } f \ \} \\
& \lambda(e,c).\ exec\ ((\lambda(x,y).\ x \mathbin{++} y)\ (compile\ e,\ id\ c)) \\
= & \quad \{ \text{ beta reduction } \} \\
& \lambda(e,c).\ exec\ (compile\ e \mathbin{++} id\ c) \\
= & \quad \{ \text{ apply definition of } id \ \} \\
& \lambda(e,c).\ exec\ (compile\ e \mathbin{++} ((\lambda a.\ a)\ c)) \\
= & \quad \{ \text{ beta reduction } \} \\
& \lambda(e,c).\ exec\ (compile\ e \mathbin{++} c)
\end{aligned}$$

Next, the *encode* function must map a pair consisting of an integer and the code continuation to a function that takes a stack as an argument and returns a modified stack:

$$\mathit{encode} \;\equiv\; \lambda(x, y).\lambda s.\; \mathit{exec}\; y\; (x : s)$$

The right-hand side of equation 2.3 can then be simplified as follows:

$$\begin{aligned}
& \lambda(e, c).\; \mathit{encode}\; (\mathit{eval}\; e,\; \mathit{id}\; c) \\
= & \quad \{ \text{ apply definition of } \mathit{id} \text{ } \} \\
& \lambda(e, c).\; \mathit{encode}\; (\mathit{eval}\; e,\; ((\lambda a.\; a)\; c)) \\
= & \quad \{ \text{ apply definition of } \mathit{encode} \text{ } \} \\
& \lambda(e, c).\; ((\lambda(x, y).\lambda s.\; \mathit{exec}\; y\; (x : s))\; (\mathit{eval}\; e,\; ((\lambda a.\; a)\; c))) \\
= & \quad \{ \text{ beta reduction of inner application } \} \\
& \lambda(e, c).\; ((\lambda(x, y).\lambda s.\; \mathit{exec}\; y\; (x : s))\; (\mathit{eval}\; e,\; c)) \\
= & \quad \{ \text{ beta reduction } \} \\
& \lambda(e, c).\lambda s.\; \mathit{exec}\; c\; (\mathit{eval}\; e : s)
\end{aligned}$$

Thus, the correctness equation for the compiler can be expressed as:

$$\lambda(e, c).\; \mathit{exec}\; (\mathit{compile}\; e \mathbin{+\!\!+} c) = \lambda(e, c).\lambda s.\; \mathit{exec}\; c\; (\mathit{eval}\; e : s)$$

Or more concisely by function extensionality and applying both sides with the argument s:

$$\mathit{exec}\; (\mathit{compile}\; e \mathbin{+\!\!+} c)\; s = \mathit{exec}\; c\; (\mathit{eval}\; e : s)$$

The correctness equation now states that compiling an expression e and appending the result to the code continuation c, then executing this combined code with a stack s, gives the same result as executing c with a stack where the evaluation of e has been pushed on top of the stack s.

Eliminating Append from the Compiler Correctness Equation

The need to append the continuation can be eliminated by introducing a more general function $compile'$ that inherently captures the idea of appending the compiled expression e to the code continuation c, much like the previous function f:

$$
\begin{aligned}
&\mathit{compile}' &&:: (\mathit{Expr}, \mathit{Code}) \rightarrow \mathit{Code} \\
&\mathit{compile}'\ (e,\ c) &&= \mathit{compile}\ e \mathbin{+\!\!+} c
\end{aligned}
$$

Substituting this into the compiler correctness equation yields:

$$\mathit{exec}\ (\mathit{compile}'\ (e,\ c))\ s = \mathit{exec}\ c\ (\mathit{eval}\ e : s)$$

This equation states that compiling an expression e together with a code continuation c and then executing the resulting code together with a stack s, is equivalent to evaluating e, pushing the result onto the stack s, and then executing c together with the updated stack.

Extending the Source Language

Up to this point, the compiler correctness equations have been framed for simple source languages. However, programming languages often need to handle more complex features, such as exceptions. To explore how the compiler correctness equation can be adapted to this, let's extend the source language to support exception handling.

To achieve this, we can use the datatype *Maybe a* to represent computations that can either succeed with a value *Just a* or fail with an exception *Nothing*.

$$\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

The domain of the source language semantics can then be instantiated to a *Maybe Int* type, leading to the following signature for the evaluation function *eval*:

$$\mathit{eval} :: \mathit{Expr} \rightarrow \mathit{Maybe}\ \mathit{Int}$$

Incorporating this into the diagram, we obtain the diagram shown in Figure 2.6.

The diagram leads to the following compiler correctness equation:

$$\lambda(e, c).\ \mathit{exec}\ (\mathit{compile}'\ (e,\ c)) = \lambda(e, c).\ \mathit{encode}\ (\mathit{eval}\ e,\ \mathit{id}\ c)$$

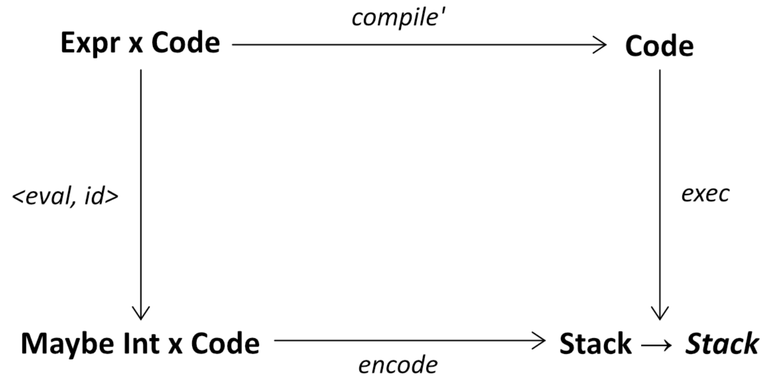

Figure 2.6: Commutative diagram illustrating compiler correctness with the semantic domain of the source language instantiated to a Maybe Int type.

With the introduction of the *Maybe Int* type, the *encode* function must be adapted to handle both normal and exceptional cases. The function now takes a *Maybe Int* value together with the code continuation and produces a function that operates on a stack.

$$encode \equiv \lambda(m, y).\lambda s.\ (m\ (\lambda x.\ exec\ y\ (x : s)))\ (exec\ y\ s)$$

To utilize lambda calculus to define the correctness equation, both *Just* and *Nothing* can be expressed as functions:

$$\begin{aligned} Just &\equiv \lambda x.\lambda just.\lambda nothing.\ just\ x \\ Nothing &\equiv \lambda just.\lambda nothing.\ nothing \end{aligned}$$

For the right-hand side of the correctness equation, consider the case where $eval\ e = \ Just\ n$:

$$\lambda(Just\ n,\ c).\ encode\ (Just\ n,\ id\ c)$$

This expression reduces to:

$$\lambda(Just\ n,\ c).\lambda s.\ exec\ c\ (n : s)$$

This means that when $eval\ e = Just\ n$, the evaluation is successful, producing the value n that is then pushed on top of the stack before executing the continuation c.

Similarly, for the case where $eval\ e = Nothing$:

$$\lambda(Nothing,\ c).\ encode\ (Nothing,\ id\ c)$$

This reduces to:

$$\lambda(Nothing,\ c).\lambda s.\ exec\ c\ s$$

In this case, since *eval e* results in *Nothing*, no value is produced, and the stack remains unchanged. The continuation c is then executed with the original stack s.

The two cases for the right-hand side of the correctness equation can be combined using a case-of expression:

$$\begin{aligned} \lambda(e,\ c).\lambda s.\ &\mathbf{case}\ eval\ e\ \mathbf{of} \\ &\quad Just\ n \quad \rightarrow\ exec\ c\ (n : s) \\ &\quad Nothing \rightarrow\ exec\ c\ s \end{aligned}$$

The compiler correctness for the extended source language, which now accounts for the possibility of exceptions, can thus be expressed by the following equation:

$$\begin{aligned} exec\ (compile'\ (e,\ c))\ s = &\ \mathbf{case}\ eval\ e\ \mathbf{of} \\ &\quad Just\ n \quad \rightarrow\ exec\ c\ (n : s) \\ &\quad Nothing \rightarrow\ exec\ c\ s \end{aligned}$$

This equation states that compiling an expression e together with a code continuation c and then executing the resulting code together with a stack s is equivalent to evaluating e. If e evaluates to *Just n*, the value n is pushed onto the stack before executing the code continuation c. However, if e evaluates to *Nothing*, the stack remains unchanged, and c is executed directly.

While this approach works, it simply ignores the failure by continuing the execution with the original stack. In a more conventional approach, the failure handling is handled explicitly rather than allowing the execution to proceed as if nothing went wrong. To introduce explicit failure handling, we can incorporate a function *fail* that takes a stack as an argument and returns a modified stack, reflecting the failure appropriately. With this *fail* function, the *encode* function is redefined as:

$$encode \equiv \lambda(m, y).\lambda s.\ (m\ (\lambda x.\ exec\ y\ (x : s)))\ (fail\ s)$$

This modification leads to the following compiler correctness equation:

$$\begin{aligned} exec\ (compile'\ (e,\ c))\ s = \textbf{case}\ eval\ e\ \textbf{of} \\ Just\ n \quad \rightarrow\ exec\ c\ (n : s) \\ Nothing \rightarrow\ fail\ s \end{aligned}$$

The inclusion of the $fail$ function instructs the virtual machine to halt the normal execution flow (the execution of the additional code c) and instead apply some form of failure handling. This approach ensures that failures are explicitly managed rather than simply allowing the continuation to proceed as if nothing went wrong.

As a result, the compiler correctness equation can now serve as a partial specification. It defines how to handle both successful evaluations and failures while leaving the specific implementation details of the failure handling mechanism to be determined during the calculation process.

Several other modifications, such as adjusting the stack to handle different data types or including additional code continuations to extend the compilation process, can similarly be integrated into the compiler correctness equation. With the reasoning behind the compiler correctness equation established, we can now explore how these equations can serve as specifications in the context of calculating compilers.

2.1.2 Calculating Compilers

To illustrate the approach of calculating compilers, I will begin by performing a complete compiler calculation for an extended arithmetic expression language. This will demonstrate the full methodology of the calculation process, showing how each component – compiler, target language, and virtual machine – can be derived from the source language syntax and semantics given specifications that captures the correctness of the compiler.

Once the approach is clear, I will shift focus to specific steps in the calculations for more complex language features, such as exceptions.

The types and functions involved in the calculations will be presented using Haskell syntax.

Simple Expression Language

Consider the simple source language of extended arithmetic expressions comprising integer values, addition and a conditional if-then-else operator:

$$\textbf{data}\ Expr = Val\ Int \mid Add\ Expr\ Expr \mid Ite\ Expr\ Expr\ Expr$$

The source language semantics is given through an evaluation function, defined compositionally, that takes an $Expr$ as input and returns an Int:

$$\begin{aligned}
&eval && :: \ Expr \rightarrow Int \\
&eval\ (Val\ n) && = n \\
&eval\ (Add\ x\ y) && = eval\ x\ +\ eval\ y \\
&eval\ (Ite\ z\ x\ y) && = \textbf{if}\ eval\ z == 0\ \textbf{then}\ eval\ y\ \textbf{else}\ eval\ x
\end{aligned}$$

In the context of an if-then-else expression, the evaluation of z resulting in 0 represents False, and the evaluation of z resulting in any other integer represents True.

Elaboration of Types and Functions

The target language is represented as the type *Code*, which comprises a list of target language operations, *Op*.

$$\begin{aligned}
&\textbf{type}\ Code = [Op] \\
&\textbf{data}\ Op
\end{aligned}$$

The specific target language operations are initially left undefined, as their definitions will naturally emerge from the calculation process.

Compiling an individual $Expr$ into a sequence of target language operations, *Code*, is captured by the function *compile* with the following signature:

$$compile\ ::\ Expr \rightarrow Code$$

The generalized compilation function $compile'$ is defined as follows:

$$compile' :: Expr \rightarrow Code \rightarrow Code$$

The function $compile'$ takes an expression and a sequence of operations, producing a new sequence of operations.

The execution of the compiled code is performed on a stack-based virtual machine, captured by the function $exec$:

$$exec :: Code \rightarrow Stack \rightarrow Stack$$

The stack is represented as a list of integers following the Last-In-First-Out (LIFO) principle:

$$\textbf{type}\ Stack = [Int]$$

Correctness Equations

The correctness of the compiler is captured by the following two equations:

$$exec\ (compile\ e)\ s = eval\ e : s \tag{2.4}$$

$$exec\ (compile'\ e\ c)\ s = exec\ c\ (eval\ e : s) \tag{2.5}$$

The first equation ensures the correctness of the compiler for the top-level compilation function $compile$. It states that compiling an expression e and executing the resulting code with an initial stack s is equivalent to evaluating the expression e and pushing the result onto the stack s.

The second equation ensures the correctness of the generalized compilation function $compile'$. It asserts that after compiling and executing an expression e with the continuation c, the result is the same as first evaluating e, pushing the result onto the stack s, and then executing c.

Calculations

To derive the compiler, target language operations and virtual machine, we use the two correctness equations directly as a specification. For the generalized compilation function, $compile'$, the goal is to rewrite the

left-hand side into the form $exec\ c'\ s$ for some code c', for each case of e. From this, we can conclude that $compile'\ e\ c = c'$ satisfies the specification.

Likewise, for the top-level compilation function, *compile*, the goal is to rewrite the left-hand side into the form $exec\ c\ s$ for some code c, which concludes that $compile\ e = c$ satisfies the specification.

We begin with the correctness equation for $compile'$ and proceed by induction on the expression e. The base case, $Val\ n$, proceeds as follows:

$$
\begin{aligned}
& exec\ (compile'\ (Val\ n)\ c)\ s \\
= \quad & \{ \text{ specification (2.5) } \} \\
& exec\ c\ (eval\ (Val\ n) : s) \\
= \quad & \{ \text{ apply defintion of } eval \ \} \\
& exec\ c\ (n : s) \\
= \quad & \{ \text{ define: } exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s) \ \} \\
& exec\ (PUSH\ n : c)\ s
\end{aligned}
$$

In the calculations, an essential step involves defining $exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s)$ to continue the calculations, as no further definitions could have been applied. Since the goal is to end up with an expression of the form $exec\ c'\ s$ for some code c', it becomes necessary to introduce a new constructor $PUSH :: Int \rightarrow Op$ in the Op type. This new constructor enables us to solve the equation:

$$exec\ c'\ s = exec\ c\ (n : s)$$

Here, c and n are unbound in the left-hand side of the equation, meaning they need to be packaged together into c'. We solve the equation by appending the the new Op type to the code continuation:

$$exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s)$$

This equation states that appending the operation $PUSH\ n$ with c, and then executing the resulting code with a stack s, is the same as executing the code, c, with a stack containing n pushed on top of the stack.

Since the final term has the form $exec\ c'\ s$, where $c' = PUSH\ n : c$, we can conclude that the specification is satisfied for the base case:

$$compile'\ (Val\ n)\ c = PUSH\ n : c$$

For the inductive case, $Ite\ z\ x\ y$, we begin in a similar manner and continue by using the induction hypothesis for x and y:

$$\begin{aligned}
& exec\ (compile'\ (Ite\ z\ x\ y)\ c)\ s \\
= & \quad \{\ \text{specification (2.5)}\ \} \\
& exec\ c\ (eval\ (Ite\ z\ x\ y) : s) \\
= & \quad \{\ \text{apply definition of } eval\ \} \\
& exec\ c\ ((\textbf{if}\ eval\ z == 0\ \textbf{then}\ eval\ y\ \textbf{else}\ eval\ x) : s) \\
= & \quad \{\ \text{distribute } (: s) \text{ over conditionals}\ \} \\
& exec\ c\ (\textbf{if}\ eval\ z == 0\ \textbf{then}\ (eval\ y : s)\ \textbf{else}\ (eval\ x : s)) \\
= & \quad \{\ \text{distribute } (exec\ c) \text{ over conditionals}\ \} \\
& \textbf{if}\ eval\ z == 0\ \textbf{then}\ exec\ c\ (eval\ y : s)\ \textbf{else}\ exec\ c\ (eval\ x : s) \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
& \textbf{if}\ eval\ z == 0\ \textbf{then}\ exec\ c\ (eval\ y : s)\ \textbf{else}\ exec\ (compile'\ x\ c)\ s \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
& \textbf{if}\ eval\ z == 0\ \textbf{then}\ exec\ (compile'\ y\ c)\ s\ \textbf{else}\ exec\ (compile'\ x\ c)\ s
\end{aligned}$$

Once again, no further definitions can be applied. To proceed, we need to solve the following equation for some code c':

$$\begin{aligned}
exec\ c'\ (eval\ z : s) = {} & \textbf{if}\ eval\ z == 0 \\
& \textbf{then}\ exec\ (compile'\ y\ c)\ s \\
& \textbf{else}\ exec\ (compile'\ x\ c)\ s
\end{aligned}$$

We start by generalizing from the specific values of $eval\ z$, $compile'\ y\ c$ and $compile'\ x\ c$:

$$exec\ c'\ (n : s) = \textbf{if}\ n == 0\ \textbf{then}\ exec\ c\ s\ \textbf{else}\ exec\ c''\ s$$

To solve this equation, we introduce a new constructor $JUMP :: Code \rightarrow Op$ in the Op type. This constructor represents a conditional jump operation:

$$exec\ (JUMP\ c'' : c)\ (n : s) = \textbf{if}\ n == 0\ \textbf{then}\ exec\ c\ s\ \textbf{else}\ exec\ c''\ s$$

This means that executing the operation $JUMP\ c''$ appended to code sequence c with a stack where n is pushed on top, proceeds by evaluating the condition n. If n is 0, the execution continues with the code c, otherwise it continues with the code c''.

The calculation then continues:

$$
\begin{aligned}
& \textbf{if } \mathit{eval}\ z == 0 \textbf{ then } \mathit{exec}\ (\mathit{compile'}\ y\ c)\ s \textbf{ else } \mathit{exec}\ (\mathit{compile'}\ x\ c)\ s \\
= \quad & \{ \text{ define: } \mathit{exec}\ (\mathit{JUMP}\ c'' : c)\ (n : s) = \textbf{if } n == 0 \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } \mathit{exec}\ c\ s \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{exec}\ c''\ s\ \} \\
& \mathit{exec}\ (\mathit{JUMP}\ (\mathit{compile'}\ x\ c) : (\mathit{compile'}\ y\ c))\ (\mathit{eval}\ z : s) \\
= \quad & \{ \text{ induction hypothesis for } z\ \} \\
& \mathit{exec}\ (\mathit{compile'}\ z\ (\mathit{JUMP}\ (\mathit{compile'}\ x\ c) : (\mathit{compile'}\ y\ c)))\ s
\end{aligned}
$$

Again, we can conclude that the specification is satisfied for the inductive case $\mathit{Ite}\ z\ x\ y$:

$$\mathit{compile'}\ (\mathit{Ite}\ z\ x\ y)\ c = \mathit{compile'}\ z\ (\mathit{JUMP}\ (\mathit{compile'}\ x\ c) : (\mathit{compile'}\ y\ c))$$

The inductive case, $\mathit{Add}\ x\ y$, proceeds in a similar fashion:

$$
\begin{aligned}
& \mathit{exec}\ (\mathit{compile'}\ (\mathit{Add}\ x\ y)\ c)\ s \\
= \quad & \{ \text{ specification (2.5) } \} \\
& \mathit{exec}\ c\ (\mathit{eval}\ (\mathit{Val}\ x\ y) : s) \\
= \quad & \{ \text{ apply definition of } \mathit{eval}\ \} \\
& \mathit{exec}\ c\ ((\mathit{eval}\ x + \mathit{eval}\ y) : s) \\
= \quad & \{ \text{ define: } \mathit{exec}\ (\mathit{ADD} : c)\ (n : m : s) = \mathit{exec}\ c\ ((m + n) : s)\ \} \\
& \mathit{exec}\ (\mathit{ADD} : c)\ (\mathit{eval}\ y : \mathit{eval}\ x : s) \\
= \quad & \{ \text{ induction hypothesis for } y\ \} \\
& \mathit{exec}\ (\mathit{compile'}\ y\ (\mathit{ADD} : c))\ (\mathit{eval}\ x : s) \\
= \quad & \{ \text{ induction hypothesis for } x\ \} \\
& \mathit{exec}\ (\mathit{compile'}\ x\ (\mathit{compile'}\ y\ (\mathit{ADD} : c)))\ s
\end{aligned}
$$

Once again, the specification is satisfied:

$$\mathit{compile'}\ (\mathit{Add}\ x\ y)\ c = \mathit{compile'}\ x\ (\mathit{compile'}\ y\ (\mathit{ADD} : c))$$

For the top-level compilation function, no induction is needed. Instead, we introduce a new constructor into the Op type that halts the execution by returning the current stack:

$$
\begin{aligned}
& \mathit{exec}\ (\mathit{compile}\ e)\ s \\
= \quad & \{ \text{ specification (2.4) } \} \\
& \mathit{eval}\ e : s \\
= \quad & \{ \text{ define: } \mathit{exec}\ [\mathit{HALT}]\ s = s\ \} \\
& \mathit{exec}\ [\mathit{HALT}]\ (\mathit{eval}\ e : s) \\
= \quad & \{ \text{ specification (2.5) } \} \\
& \mathit{exec}\ (\mathit{compile'}\ e\ [\mathit{HALT}])\ s
\end{aligned}
$$

This demonstrates that the specification is satisfied for the top-level compilation function:

$$compile\ e = compile'\ e\ [HALT]$$

Derived Definitions

In conclusion, the following definitions have been derived from the calculations:

$$
\begin{aligned}
&\textbf{type}\ Code = [Op] \\
&\textbf{data}\ Op \quad = PUSH\ Int \\
&\qquad\qquad\ \ \mid ADD \\
&\qquad\qquad\ \ \mid JUMP\ Code \\
&\qquad\qquad\ \ \mid HALT
\end{aligned}
$$

$$
\begin{aligned}
&compile && :: Expr \to Code \\
&compile\ e && = compile'\ e\ [HALT] \\
\\
&compile' && :: Expr \to Code \to Code \\
&compile'\ (Val\ n)\ c && = PUSH\ n : c \\
&compile'\ (Add\ x\ y)\ c && = compile'\ x\ (compile'\ y\ (ADD : c)) \\
&compile'\ (Ite\ z\ x\ y)\ c && = compile'\ z\ (JUMP\ (compile'\ x\ c) : (compile'\ y\ c))
\end{aligned}
$$

$$
\begin{aligned}
&exec && :: Code \to Stack \to Stack \\
&exec\ [HALT]\ s && = s \\
&exec\ (PUSH\ n : c)\ s && = exec\ c\ (n : s) \\
&exec\ (ADD : c)\ (n : m : s) && = exec\ c\ ((m + n) : s) \\
&exec\ (JUMP\ c'' : c)\ (n : s) && = \textbf{if}\ n == 0\ \textbf{then}\ exec\ c\ s\ \textbf{else}\ exec\ c''\ s
\end{aligned}
$$

The implementation of *exec* is partial and does not define behavior for all possible inputs. Specifically, $exec\ [\,]\ s$ and cases where the stack does not contain the required values for *ADD* or *JUMP* operations represent undefined behavior. However, for well-formed code produced by the compiler, the *exec* function should behave correctly.

Exceptions

To handle exceptions in the source language, we can introduce two new constructors, *Throw* and *Catch*:

```
data Expr = Val Int
          | Add Expr Expr
          | Ite Expr Expr Expr
          | Throw
          | Catch Expr Expr
```

The source language semantics can then be expressed by the function *eval*, which now maps the source language to *Maybe Int*:

```
eval             :: Expr → Maybe Int
eval (Val n)     = Just n
eval (Add x y)   = case eval x of
                     Just n  → case eval y of
                                 Just m  → Just (n + m)
                                 Nothing → Nothing
                     Nothing → Nothing
eval (Ite z x y) = case eval z of
                     Just n  → if n == 0 then eval y else eval x
                     Nothing → Nothing
eval (Throw)     = Nothing
eval (Catch x h) = case eval x of
                     Just n  → Just n
                     Nothing → eval h
```

The evaluation of *Throw* results in *Nothing*, representing an exceptional value, while evaluating *Catch x h* first evaluates the expression x. If x results in *Nothing* (indicating an exception), the expression h is evaluated as the handler. Otherwise, the result is the value of expression x.

Additionally, the stack will be modified to be extensible:

```
type Stack = [Elem]
data Elem = VAL Int
```

This modification allows us to introduce new constructors to the *Elem* type during the calculations, which can accommodate not only integer values but also other data types.

The correctness of the generalized compilation function $compile'$ is captured in the following equation, which serves as a partial specification:

$$\begin{aligned} exec\ (compile'\ e\ c)\ s = \textbf{case}\ &eval\ e\ \textbf{of} \\ &Just\ n \quad \rightarrow exec\ c\ (VAL\ n : s) \\ &Nothing \rightarrow fail\ s \end{aligned}$$

Simplification of Case Expressions

The calculations for the base cases $Val\ n$ and $Throw$ are straightforward. Similarly, the calculations for the inductive cases $Add\ x\ y$ and $Ite\ z\ x\ y$ closely resemble the same steps as those in the calculations for the simple source language of extended arithmetic expressions. For the complete calculations, refer to Appendix A.

Although the calculations are largely similar, a critical step for the source language that includes exceptions involves simplifying cases of the evaluated expression.

For example, in the base case of $Val\ n$, the simplification step involves recognizing that after applying the definition of $eval$, the result should directly match the corresponding case expression during simplification.

$$\begin{aligned}
= &\quad \{ \ldots \} \\
&\textbf{case}\ eval\ (Val\ n)\ \textbf{of} \\
&\quad Just\ n \quad \rightarrow exec\ c\ (VAL\ n : s) \\
&\quad Nothing \rightarrow fail\ s \\
= &\quad \{ \text{apply definition of } eval \} \\
&\textbf{case}\ (Just\ n)\ \textbf{of} \\
&\quad Just\ n \quad \rightarrow exec\ c\ (VAL\ n : s) \\
&\quad Nothing \rightarrow fail\ s \\
= &\quad \{ \text{simplify} \} \\
&exec\ c\ (VAL\ n : s) \\
= &\quad \{ \ldots \}
\end{aligned}$$

Likewise, for the inductive case $Add\ x\ y$, applying the definition of $eval$ expands to:

```
=   { ... }
  case eval (Add x y) of
    Just n    → exec c (VAL n : s)
    Nothing → fail s
=   { apply definition of eval }
  case (case eval x of
          Just n    → case eval y of
                         Just m    → Just (n + m)
                         Nothing → Nothing
          Nothing → Nothing) of
    Just n    → exec c (VAL n : s)
    Nothing → fail s
=   { ... }
```

At this point, a simplification step is necessary. We simplify the nested case expressions by distributing the logic down into the appropriate branches, which leads to the following simplified form:

```
=   { ... }
  case (case eval x of
          Just n    → case eval y of
                         Just m    → Just (n + m)
                         Nothing → Nothing
          Nothing → Nothing) of
    Just n    → exec c (VAL n : s)
    Nothing → fail s
=   { simplify }
  case eval x of
    Just n    → case eval y of
                  Just m    → exec c (VAL (n + m) : s)
                  Nothing → fail s
    Nothing → fail s
=   { ... }
```

Selective Application

Another noteworthy step from the calculation of the inductive case *Add x y* is the following:

```
=   { ... }
```

$$
\begin{array}{l}
\quad \textbf{case } \mathit{eval}\ x \textbf{ of} \\
\qquad \mathit{Just}\ n \quad \to \textbf{case } \mathit{eval}\ y \textbf{ of} \\
\qquad\qquad\qquad \mathit{Just}\ m \quad \to \mathit{exec}\ (\mathit{ADD} : c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
\qquad\qquad\qquad \mathit{Nothing} \to \mathit{fail}\ s \\
\qquad \mathit{Nothing} \to \mathit{fail}\ s \\
= \quad \{ \text{ define: } \mathit{fail}\ (\mathit{VAL}\ n : s) = \mathit{fail}\ s \} \\
\quad \textbf{case } \mathit{eval}\ x \textbf{ of} \\
\qquad \mathit{Just}\ n \quad \to \textbf{case } \mathit{eval}\ y \textbf{ of} \\
\qquad\qquad\qquad \mathit{Just}\ m \quad \to \mathit{exec}\ (\mathit{ADD} : c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
\qquad\qquad\qquad \mathit{Nothing} \to \mathit{fail}\ (\mathit{VAL}\ n : s) \\
\qquad \mathit{Nothing} \to \mathit{fail}\ s \\
= \quad \{ \ldots \}
\end{array}
$$

In this step, a definition of the function $fail$ is introduced, which essentially unwinds the stack. It is important to note that this definition is applied specifically in the case where the evaluation of x results in $Just\ n$. We could have applied the definition to both cases of $fail\ s$. However, the goal of this step is to make the inner case expression match the form of the specification so that we can apply the induction hypothesis for y. Essentially, we only manipulate the expression to align with the form needed for applying the induction hypothesis, rather than altering the entire structure of the expression unnecessarily. This selective application is necessary for the calculation to proceed correctly.

Flexibility in the Calculations

It is also worth noting that the extensibility of the stack allows for flexibility during the calculations. Consider the calculations for the inductive case $Ite\ z\ x\ y$:

$$
\begin{array}{l}
\quad \mathit{exec}\ (\mathit{compile'}\ (\mathit{Ite}\ z\ x\ y)\ c)\ s \\
= \quad \{ \text{ specification for } \mathit{compile'} \} \\
\quad \textbf{case } \mathit{eval}\ (\mathit{Ite}\ z\ x\ y) \textbf{ of} \\
\qquad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
\qquad \mathit{Nothing} \to \mathit{fail}\ s \\
= \quad \{ \text{ apply definition of } \mathit{eval} \} \\
\quad \textbf{case } \mathit{eval}\ z \textbf{ of} \\
\qquad \mathit{Just}\ n \quad \to \textbf{if } n == 0 \\
\qquad\qquad\qquad \textbf{then case } \mathit{eval}\ y \textbf{ of} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ m \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ m : s)
\end{array}
$$

```
                          Nothing → fail s
                  else case eval x of
                          Just m    → exec c (VAL m : s)
                          Nothing → fail s
      Nothing → fail s
=   { induction hypothesis for x }
    case eval z of
      Just n    → if n == 0
                  then case eval y of
                          Just m    → exec c (VAL m : s)
                          Nothing → fail s
                  else exec (compile′ x c) s
      Nothing → fail s
=   { induction hypothesis for y }
    case eval z of
      Just n    → if n == 0
                  then exec (compile′ y c) s
                  else exec (compile′ x c) s
      Nothing → fail s
=   { define: exec (JUMP c″ : c) (VAL n : s) = if n== 0
                                                  then exec c s
                                                  else exec c″ s }
    case eval z of
      Just n    → exec (JUMP (compile′ x c) : (compile′ y c)) (VAL n : s)
      Nothing → fail s
=   { induction hypothesis for z }
    exec (compile′ z (JUMP (compile′ x c) : (compile′ y c))) s
```

The definition of *exec* in these calculations is conceptually similar to that used in the calculations for the simple source language with extended arithmetic expressions.

However, since the stack is extensible, the alternative code c'' could have been provided on the stack by introducing a new constructor *ALT* in the *Elem* type. The calculations would then have proceeded as follows:

```
=   { ... }
    case eval z of
      Just n    → if n == 0
                  then exec (compile′ y c) s
```

$$
\begin{array}{ll}
 & \qquad\qquad\qquad \textbf{else}\ exec\ (compile'\ x\ c)\ s \\
 & \quad Nothing \rightarrow fail\ s \\
= & \{\ \text{define: } exec\ (JUMP : c)\ (VAL\ n : ALT\ c'' : s) = \textbf{if}\ n{==}\ 0 \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ exec\ c\ s \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ exec\ c''\ s\ \} \\
 & \textbf{case}\ eval\ z\ \textbf{of} \\
 & \quad Just\ n \quad \rightarrow \\
 & \qquad\quad exec\ (JUMP : (compile'\ y\ c))\ (VAL\ n : ALT\ (compile'\ x\ c) : s) \\
 & \quad Nothing \rightarrow fail\ s \\
= & \{\ \text{induction hypothesis for } z\ \} \\
 & exec\ (compile'\ z\ (JUMP : (compile'\ y\ c)))\ (ALT\ (compile'\ x\ c) : s) \\
= & \{\ \text{define: } exec\ (BRANCH\ c' : c)\ s = exec\ c\ (ALT\ c' : s)\ \} \\
 & exec\ (BRANCH\ (compile'\ x\ c) : (compile'\ z\ (JUMP : (compile'\ y\ c))))\ s
\end{array}
$$

This approach allows for a more flexible control flow and might be better suited to some compiler specifications.

The calculations of extending the source language to handle state closely resemble those for exception handling. To keep this discussion concise and avoid repetition, the detailed calculations for a source language including primitives to handle state are not included here. However, readers interested in exploring the calculations for a source language with state can refer to Appendix B.

2.2 Rewriting Systems

Rewriting, or reduction, involves the step-wise transformation of objects, where these transformation are represented by a relation $\rightarrow$, known as the *reduction relation*.

Consider the following arithmetic expression:

$$10 \cdot 5 + 4 \cdot 3$$

This expression might simplify as follows:

$$10 \cdot 5 + 4 \cdot 3 \rightarrow 50 + 4 \cdot 3 \rightarrow 50 + 12 \rightarrow 62$$

In the first simplification step, the sub-expression $4 \cdot 3$ could have been simplified before $10 \cdot 5$. This illustrates that the simplification process can be *non-deterministic*, meaning there are multiple possible sequences of simplification steps that can be taken. Nevertheless, in this case, all such sequences would lead to the same final result, 62, known as the *normal form*. The property that different reduction sequences yield the same result, as in this case, is called *uniqueness of normal form*.

Now consider the counting of integers starting from 0:

$$0 \rightarrow 1 \rightarrow 2 \rightarrow \ ...$$

Here, there is no normal form, as the counting process would continue indefinitely. The absence of a normal form is referred to as *non-termination*.

Two crucial concepts in rewriting systems are *termination* and *confluence*. Termination ensures that a normal form exists, while confluence ensures the uniqueness of normal forms.

To examine these concepts and provide a formal foundation for rewriting, the following section offers a brief introduction to Abstract Reduction Systems. Subsequently, a more detailed exploration of First-Order Term Rewriting Systems is presented. These sections are grounded in the material from Terese (2003).

2.2.1 Abstract Reduction Systems

As mentioned earlier, rewriting involves the step-wise transformation of objects, with these transformations represented by a reduction relation $\rightarrow$. To formalize this process, we use the framework of an *Abstract Reduction System (ARS)*. The generality of ARS allows us to study rewriting systems independently of the specific nature of the objects being transformed.

Basic Concepts of ARS

An ARS is defined as a structure $\mathcal{A} = (A, \{\rightarrow_\alpha \mid \alpha \in I\})$, where A is a set of objects and $\rightarrow_\alpha$, is a binary relation, indexed by a set I, that represents possible transformations of these objects.

In this context:

- The symbol $\equiv$ is used to denote identity on A . This means that two objects in A are considered identical if they are syntactically the same.

- The symbol $=$ is used to represent the equivalence relation generated by $\rightarrow$, called *convertibility*.

Reduction
A *reduction step* is a specific instance of the relation $\rightarrow_\alpha$ within a *reduction sequence*. For example, in a sequence $a_0 \rightarrow_\alpha a_1 \rightarrow_\alpha a_2 \rightarrow_\alpha ...$, where $a_0, a_1, a_2 \in A$, each pair $(a_x, a_{x+1}) \in \rightarrow_\alpha$ represents a reduction step from a_x to a_{x+1}. Here, a_{x+1} is called a *one-step reduct* of a_x.

Reduction sequences can be *finite*, as shown in the arithmetic expression example where the sequence terminates at 62, or *infinite*, as in the case of counting integers, where the process continues indefinitely without reaching a normal form.

Conversion
A *conversion sequence* in relation to $\rightarrow_\alpha$ is a reduction sequence with respect to the symmetric closure $\leftrightarrow_\alpha = \leftarrow_\alpha \cup \rightarrow_\alpha$, associated with $\rightarrow_\alpha$.

Two objects $a \in A$ and $b \in A$ are said to be *convertible* in relation to $\rightarrow_\alpha$ if there exists a finite conversion sequence $a_0 \leftrightarrow_\alpha a_1 \leftrightarrow_\alpha \ldots \leftrightarrow_\alpha a_n$ where $a \equiv a_0$ and $b \equiv a_n$. The convertibility between a and b is then denoted $a =_\alpha b$.

This means that convertibility between a pair $a, b \in A$ with $a =_\alpha b$ implies the existence of a finite conversion sequence where each step is either a reduction or an expansion that connects a and b.

Confluence
An object $a \in A$ is said to be *confluent* if, whenever it can be reduced to two different objects $b \in A$ and $c \in A$ by a finite reduction sequence, there exists some object $d \in A$ such that both b and c can be further reduced to d through a finite reduction sequence. Additionally, a reduction relation $\rightarrow$ is confluent if every $a \in A$ is confluent.

Normalization
An object $a \in A$ is a *normal form* if there is no object $b \in A$ such that $a \rightarrow b$. In other words, a is a normal form if it cannot be further reduced using the reduction relation $\rightarrow$.

An object $a \in A$ is *weakly normalizing* if there exists at least one finite reduction sequence starting from a that leads to a normal form. This means that a can be reduced to a normal form, but not necessarily through every possible reduction sequence.

An object $a \in A$ is *strongly normalizing* if every possible reduction sequence starting from a is finite. This implies that regardless of the reduction path taken, the sequence will always terminate in a normal form. Additionally, a reduction relation $\rightarrow$ is *terminating* if every $a \in A$ is strongly normalizing.

Completeness
A reduction relation $\rightarrow$ is said to be *complete* if it is both confluent and terminating. This means that every reduction sequence eventually terminates and that all reduction paths leads to an unique normal form.

2.2.2 First-Order Term Rewriting Systems

A *Term Rewriting System (TRS)* is a specific type of abstract reduction system where the objects being transformed are *first-order terms*, and the reduction relations are defined by a set of *rewrite rules*.

Terms

A *term* is constructed from symbols drawn from an *alphabet*. The alphabet consists of:

- *Function symbols* defined in a *signature*.
- A countably infinite set of *variables*.

A *signature* Σ comprises a non-empty set of *function symbols*, each associated with a specific arity. These function symbols are denoted by $F, G, ...,$ each with a fixed arity. For example, a function symbol F with arity n can be applied to n terms, forming a term of the form $F(t_1, ...t_n)$ where each t_i is a term by itself. The terms $t_1, ..., t_n$ are called the *arguments* of the term $F(t_1, ..., t_n)$.

Variables are placeholders that can represent any term. They are denoted by x, y, z, x', y', z' or indexed as $x_0, x_1, ...x_n,$. For example, in the term $F(x, t_1)$, x is a variable, while t_1 is a term.

The set of all possible terms that can be formed using the signature Σ is denoted by $Ter(\Sigma)$ and the set of variables is denoted by Var.

Substitution
Substitution is the operation of replacing variables within a term with other terms. Formally, a substitution is a mapping $\sigma : Var \to Ter(\Sigma)$, which satisfies:

$$\sigma(F(t_1, ..., t_n)) \equiv F(\sigma(t_1), ..., \sigma(t_n))$$

This means that applying a substitution to a term involving the function symbol F is equivalent to applying the substitution to each of its arguments t_i. If F is a nullary (0-arity) function symbol, then $\sigma(F) \equiv F$,

as there are no variables within F to substitute.

Substitutions can be represented as a finite set of assignments to variables. For example, the substitution σ with $\sigma(x_i) \equiv s_i$ can be denoted by the set: $\{ x_1 \mapsto s_1, ..., x_n \mapsto s_n \}$. This means that in the substitution σ, each variable x_i is replaced by a corresponding term s_i whenever σ is applied to a term.

Matching

Matching involves determining whether there exists a substitution that, when applied to a term, makes it identical to another term. Given two terms t and s, matching tries to find a substitution σ such that:

$$\sigma(t) \equiv s$$

When a substitution exists, the term t is said to match its instance s by the substitution σ.

A matching algorithm can be utilized to determine if there exists a substitution that matches the term t to its instance s. The input is the two terms and the output is a substitution σ such that $\sigma(t) \equiv s$, if one exists.

A matching algorithm can be described as follows:

1. **Initialize** a substitution σ as an empty set $\emptyset$.
2. If $t \in Vars$, e.g. $t \equiv x$, then add the substitution $\{ x \mapsto s \}$ to σ, unless σ contains $x \mapsto s_1$ where $s \not\equiv s_1$, in which case fail with no substitution.
3. If $s \in Vars$, while $t \notin Vars$ then fail with no substitution.
4. If $t \equiv F(t_1, ...t_n)$ and $s \equiv F(s_1, ...s_n)$ then apply matching to each corresponding pair of arguments t_i and s_i and merge the resulting substitutions with σ.
5. If $t \equiv F(t_1, ...t_n)$ and $s \equiv G(s_1, ...s_n)$, where $F \not\equiv G$, then fail with no substitution.
6. If all steps succeed, return the substitution σ.

Illustrative Example of Matching

Consider the signature Σ consisting of function symbols for addition, the successor function and the constant zero, with arities 2, 1 and 0 respectively:

$$\Sigma = \{Plus/2,\ Succ/1,\ Zero/0\}$$

Let the set of variables be:

$$Vars = \{x,\ y\}$$

Terms can be constructed from the following alphabet:

$$Alphabet = \Sigma\ \cup\ Vars$$

Using this alphabet, we can construct terms such as:

$$t \equiv Plus(x, y)$$
$$s \equiv Plus(Zero, Succ(Zero))$$

We seek to find a substitution by matching the two terms, such that:

$$\sigma(Plus(x, y)) \equiv Plus(Zero, Succ(Zero))$$

The matching process begins with an empty substitution, denoted $\sigma = \emptyset$. The first step in this case, is to compare the top-level function symbols of the terms t and s. Since both t and s have the function symbol *Plus* at the top level, the process proceeds by matching each corresponding pair of arguments.

The first argument of t is the variable x, and the first argument of s is the term *Zero*. This creates the substitution $\sigma_1 = \{\ x \mapsto Zero\ \}$, which is then merged with the current substitution σ. Since σ is initially empty, the merged substitution becomes $\sigma = \emptyset \cup \sigma_1 = \sigma_1$.

Next, the second argument of t, which is the variable y, is matched with the second argument of s, which is the term $Succ(Zero)$. This yields the substitution $\sigma_2 = \{\ y \mapsto Succ(Zero)\ \}$. This new substitution is then merged with the current substitution σ, resulting in $\sigma = \sigma_1 \cup \sigma_2 = \{\ x \mapsto Zero,\ y \mapsto Succ(Zero)\ \}$.

However, in the scenario where $t \equiv Plus(x, x)$, the substitution $\{ x \mapsto Zero \}$ would be present in σ after matching the first argument. When attempting to match the second argument, which would provide the substitution $\sigma_2 = \{ x \mapsto Succ(Zero) \}$, a conflict arises. Since x cannot be substituted by both $Zero$ and $Succ(Zero)$, the matching process fails, and no valid substitution exists for $\sigma(Plus(x, x)) \equiv Plus(Zero, Succ(Zero))$.

Rewrite Rules

In term rewriting systems, *rewrite rules* (or *reduction rules*) are essential components that define how terms can be transformed into other terms.

A rewrite rule consists of a pair of terms $\langle l, r \rangle$ from the set $Ter(\Sigma)$, where l is referred to as the left-hand side and r as the right-hand side. These rules are typically expressed as $\rho : l \to r$, where ρ denotes the name of the rule or simply $l \to r$ without providing a name to the rule.

There are typically two restrictions imposed on these rules:

1. The left-hand side l is not a variable.
2. Every variable occurring in the right-hand side r must appear in l as well.

If a system contains rewrite rules that violates one or both of these restrictions, it is termed a *pseudo-TRS*.

When a substitution σ is applied to a rewrite rule ρ, it produces a specific instance of the rule, leading to an atomic rewrite step: $\sigma(l) \to_\rho \sigma(r)$. Here, $\sigma(l)$ is known as the *redex*, while $\sigma(r)$ is its *contractum*.

Consider the term s and the rewrite rule $\rho : l \to r$:

$$s \equiv Plus(Zero, Succ(Zero))$$
$$\rho : Plus(Zero, x) \to x$$

The matching algorithm can be applied to find a substitution, such that $\sigma(l) \equiv s$, if any exists. This provides the following substitution:

$$\sigma = \{ x \mapsto Succ(Zero) \}$$

This substitution is then applied to the rewrite rule to obtain the atomic rewrite step:

$$Plus(Zero, Succ(Zero)) \to_{\rho} Succ(Zero)$$

In this case, the term $Plus(Zero, Succ(Zero))$ is the redex, and the term $Succ(Zero)$ is its contractum.

A term might include multiple redexes, each of which can be replaced by its contractum. A rewrite step involves selecting one of these redexes and replacing it with its contractum. To describe rewriting in a broader context, a rewrite step using the rule $\rho : l \to r$ can be generalized as contracting a redex within an arbitrary context C:

$$C[\sigma(l)] \to_{\rho} C[\sigma(r)]$$

Consider the term s and the rewrite rule $\rho : l \to r$:

$$s \equiv Plus(Succ(Zero), Plus(Succ(Zero), Zero))$$
$$\rho : Succ(x) \to x$$

The atomic rewrite step is provided by a substitution σ, with $\sigma(x) \equiv Zero$:

$$Succ(Zero) \to_{\rho} Zero$$

Contracting the redex in the context $\Box$ in $Plus(\Box, Plus(Succ(Zero), Zero))$, results in the following rewrite step:

$$Plus(Succ(Zero), Plus(Succ(Zero), Zero))$$
$$\to_{\rho} Plus(Zero, Plus(Succ(Zero), Zero))$$

Term Rewriting Systems

Now that terms and rewrite rules have been specified, a formal definition of a TRS is provided.

A TRS can be described as a pair $\mathcal{R} = (\Sigma, R)$, where Σ represents the signature, and R is the set of rewrite rules associated with that signature.

The *one-step reduction* relation for a TRS is denoted $\rightarrow$, and is defined as the union of all individual rewrite rules $\rightarrow_\rho$ for each rule $\rho \in R$.

A TRS $\mathcal{R} = (\Sigma, R)$ can be associated with a corresponding ARS in the form $(Ter(\Sigma), \rightarrow)$. Consequently, all properties from Abstract Reduction Systems, such as confluence, normalization and completeness, apply to the TRS.

Overlap and Critical Pairs

In a TRS, the concepts of *overlap* and *critical pairs* are central to the investigation of confluence.

A *pattern* of a rewrite rule $\rho : l \rightarrow r$ is the left-hand side l where all variables are replaced by holes $\square$. For example, the pattern of the rewrite rule $Plus(x, Zero) \rightarrow x$ would be $Plus(\square, Zero)$.

A *redex pattern* corresponds to the pattern of the rewrite rule where the variables have been replaced by the specific subterms from the redex.

For example, given the redex $s \equiv Plus(Succ(Zero), zero)$ according to the rewrite rule $\rho : Plus(x, Zero) \rightarrow x$, the redex pattern would be $Plus(\square, Zero)$, where the hole $\square$ corresponds to the subterm $Succ(Zero)$.

Overlap between redex occurrences occur when a term t contains two redex occurrences s_1 and s_2, where the redex pattern of s_1 and s_2 has at least one symbol in common. This means that there is some part of the term t where the patterns of the two redexes intersect.

For example, given the term $t \equiv Plus(Plus(Zero, Zero), Zero)$, with the two redex occurrences $s_1 \equiv Plus(Plus(Zero, Zero), Zero)$ according to the rewrite rule $\rho_1 : Plus(x, Zero) \rightarrow x$, and $s_2 \equiv Plus(Zero, Zero)$ according to the rewrite rule $\rho_2 : Plus(Zero, Zero) \rightarrow Zero$, then the redex pattern for s_1 is $Plus(\square, Zero)$ with $\square$ corresponding to $Plus(Zero, Zero)$, while s_2 has the pattern $Plus(Zero, Zero)$. These redeces overlap since they share the $Plus$ function symbol.

Overlap between two rewrite rules ρ_1 and ρ_2 occurs if a term t contains two redexes s_1 and s_2 that overlap, where s_1 is a ρ_1-*redex* and s_2 is a

ρ_2*-redex*.

Consider the two rewrite rules ρ_1 and ρ_2:

$$\rho_1 : Succ(x) \to x$$
$$\rho_2 : Plus(Succ(x), y) \to y$$

In a term $t \equiv Plus(Succ(Zero), Zero)$, the redex $s_1 \equiv Succ(Zero)$ can be reduced using the rule ρ_1, resulting in the term $Plus(Zero, Zero)$. However, s_1 is also part of a larger redex $s_2 \equiv Plus(Succ(Zero), Zero)$, which can be reduced using the rule ρ_2, resulting in the term $Zero$.

Here, the redexes s_1 and s_2 overlap because the term $Succ(Zero)$ appears in both redexes. This overlap demonstrates how two rewrite rules can apply to different parts of a term, leading to different possible reductions.

A *critical pair* is a pair of terms that arise from overlapping redexes where different rules can be applied. Specifically, if s_1 and s_2 overlap in a term t as discussed, applying ρ_1 to s_1 and ρ_2 to s_2 could lead to two different terms. These two terms form a critical pair, such that in the above example, we have the critical pair $\langle Plus(Zero, Zero),\ Zero \rangle$. If the two terms have a common reduct, the critical pair is called *convergent*. If a TRS contains any non-convergent critical pairs, then it is not confluent.

In this chapter, the concepts of compiler correctness and rewriting systems have been thoroughly explored. The discussion began with an examination of compiler correctness, offering insights into how to formulate equations that capture the correctness of a compiler. These equations served as the foundation for illustrating the approach of calculating compilers – a systematic approach where the compiler, target language, and virtual machine are derived from the source language's syntax and semantics. The chapter then shifted to an examination of rewriting systems, starting with abstract reduction systems to introduce key concepts and properties, and advancing to first-order term rewriting systems. The theoretical concepts introduced in this chapter provide the foundation for the subsequent Implementation chapter, where these principles are applied.

Chapter 3

Implementation

The implementation of CoCa will be examined from both conceptual and technical perspectives. Initially, a conceptual model of CoCa will be outlined, offering a high-level understanding of the specialized tool, its capabilities, and usage. Following this, a technical model will be presented, focusing on CoCa's implementation. To maintain focus on the core ideas, surface syntax will be used rather than concrete syntax to abstract away from the specific implementation details.

3.1 Conceptual Model of CoCa

The primary purpose of CoCa is twofold:

1. To check and verify compiler calculations.

2. To facilitate the automation of compiler calculations.

While these two goals can be viewed and addressed independently, they share the same underlying principle: Ensuring that each step in the compiler calculations adheres to a set of transformational rules that preserve the semantics of the source language. This principle is realized through the interaction of three key languages: The Object Languages, the Meta Language, and the Proof Language.

- The **Object Language** represents the source and target language for the compiler and is the primary object of study.

- The **Meta Language** is used to define, analyze, and transform the syntax and semantics of the Object Languages.

- The **Proof Language** is used to issue commands that facilitate various steps in the compiler calculation process. It is the language in which specifications, justifications and proof directives are expressed.

The relationship among the three languages can be summarized as follows: The Proof Language provides commands that operates on the Meta Language, which in turn defines and manipulates the Object Languages.

To illustrate this interaction, consider a scenario where an expression e in the Meta Language undergoes a transformation through a command in the Proof Language. The transformation is expressed as follows:

$$\begin{aligned} & e \\ = & \quad \{ \text{ command } \} \\ & e' \end{aligned}$$

This shows that the command from the Proof Language transforms the expression e in the Meta Language into e'. These transformations follow rules derived from the specification and definitions, ensuring that the semantics of the Object Language expressions are preserved throughout the proof process.

While the conceptual model of CoCa encompasses both the verification and automation of compiler calculations, the actual implementation is limited to verifying compiler calculations.

The following sections will explore the interaction between the Object Languages, Meta Language and Proof Language in the context of compiler calculations.

3.1.1 Defining an Object Language with the Meta Language

The syntax and semantics of an Object Language can be defined using construct from the Meta Language. For instance, the source Object Language *Expr*, which represents an extended arithmetic expression language, can be defined in the Meta Language as follows:

$$\textbf{data}\ Expr = Val\ Int \mid Add\ Expr\ Expr \mid Ite\ Expr\ Expr\ Expr$$

This *Expr* type represents the syntax of the language by specifying the valid forms of expressions in the source Object Language.

The semantics of these expression are defined in the Meta Language through a function like *eval*, which describes how expressions in the source Object Language are evaluated:

$$\begin{aligned}
&eval && :: \ Expr \to Int \\
&eval\ (Val\ n) && = n \\
&eval\ (Add\ x\ y) && = eval\ x\ +\ eval\ y \\
&eval\ (Ite\ z\ x\ y) && = \textbf{if}\ eval\ z == 0\ \textbf{then}\ eval\ y\ \textbf{else}\ eval\ x
\end{aligned}$$

Restrictions for Defining Object Language Semantics

To ensure that the semantics of Object Languages are both consistent and complete, the function defining these semantics must adhere to the following restrictions:

- **No Free Variables**: Every variable appearing on the right-hand side of a function definition must be either:
  - Explicitly bound by the pattern on the left-hand side of the function definition, or
  - Bound within the body of the function.

  This ensures that no free variables are introduced and that all variables are properly bound within their respective scopes.

- **Complete Pattern Matching**: The function must include definitions for every possible expression in the Object Language. This guarantees that each expression in the Object Language has a defined meaning.

3.1.2 Expanding the Meta Language for Compiler Calculations

To support compiler calculations, the Meta Language is extended with constructs that specify additional types and function signatures necessary for defining the compilation and execution processes. For instance, the target language, stack type, and the signatures for compiling expressions from the source Object Language and executing the resulting target Object Language could be represented as follows:

Target language:
type $Code = [Op]$
data Op

The Stack Type:
type $Stack = [Int]$

Compilation:
$compile\ :: Expr \to Code$
$compile' :: Expr \to Code \to Code$

Execution:
$exec :: Code \rightarrow Stack \rightarrow Stack$

3.1.3 Expressing Compiler Correctness with the Proof Language

The Proof Language can be used to express specifications that define the expected behavior of the compiler, ensuring that the compiler is semantics-preserving.

For example, the following specification can be expressed in the Proof Language:

$$forall\ e\ c\ s.\ \ exec\ (compile'\ e\ c)\ s = exec\ c\ (eval\ e : s)$$

This specification asserts that for any expression e, code c, and stack s, executing the compiled code should produce the same result as directly evaluating the expression e, pushing the result onto the stack s, and then executing c with the modified stack.

From this specification, the goal is to rewrite the left-hand side into the form $exec\ c'\ s$ for some code c', thereby concluding that $compile'\ e\ c = c'$ for the given expression e in the source Object Language.

3.1.4 Proof Steps and Their Relation to the Model

This section explores the various proof steps involved in compiler calculations, specifically within the framework of the conceptual model of CoCa. These proof steps are designed to ensure that each transformation adheres to predefined rules intended to preserve the semantics of the source Object Languages.

Setting the Context

The first proof step is to identify the specific expression in a source Object Language that we want to analyze. This step is directed by a command in the Proof Language, which specifies how the proof should proceed. For example, if we consider the expression $Val\ n$ in the source Object Language, then we can use the Proof Language to instantiate the general expression e from the specification with $Val\ n$:

$Let\ e = Val\ n$

This command sets the context for the proof, specifying that e should be understood as $Val\ n$ in the subsequent steps.

If another variable had been used in the specification to represent expressions in the Object Language, such as a, then we would use the proof command with the variable a instead of e:

$Let\ a = Val\ n$

The Initial Expression

With the context established, the initial expression is derived by applying this context to the left-hand side of the specification. Thus, for the context where e is $Val\ n$, we have the following expression in the Meta Language:

$exec\ (compile'\ (Val\ n)\ c)\ s$

At this point, the next logical step is to use a command that rewrites the expression to match the right-hand side of the specification. Given that $e = Val\ n$, this results in:

$$\begin{aligned} & exec\ (compile'\ (Val\ n)\ c)\ s \\ = \quad & \{ \text{ specification for compile' } \} \\ & exec\ c\ (eval\ (Val\ n) : s) \end{aligned}$$

Since all compiler calculations typically follow this pattern, we can simplify the process by omitting this intermediate step in the implementation. Instead, when using the Proof Language command to instantiate the general expression e, we directly obtain the right-hand side expression from the specification with the context applied. For the case where $e = Val\ n$, we get the following initial expression:

$exec\ c\ (eval\ (Val\ n) : s)$

Applying the Semantic Rules of the Source Object Language

At this point, we might choose to use a command from the Proof Language to apply the definition of $eval$. The $eval$ function, defined in the Meta Language, specifies the semantic rules for evaluating expression in the source Object Language. By applying the definition of $eval$, the expression from the source Object Language is transformed into its corresponding value.

For the initial expression:

$$exec\ c\ (eval\ (Valn) : s)$$

We can use a Proof Language command $eval/1$ to specify that the first definition of $eval$ should be applied:

$$\begin{array}{ll} & exec\ c\ (eval\ (Val\ n) : s) \\ = & \{\ \text{eval/1}\ \} \\ & exec\ c\ (n : s) \end{array}$$

In this step, the Proof Language command $eval/1$ applies the rule $eval\ (Val\ n) = n$, which is part of the Meta Language's definition of $eval$. This rule functions as a rewrite rule from left to right:

$$eval\ (Val\ n) \rightarrow n$$

When applying the command, the left-hand side of the rewrite rule $eval\ (Val\ n)$ is matched with any part of the expression $exec\ c\ (eval\ (Val\ n) : s)$. If a match is found, the rewrite rule is applied, transforming the expression by replacing $eval\ (Val\ n)$ with n. The behavior of the Proof Language command is thus to transform the expression in the Meta Language to reflect how the source Object Language expression $Val\ n$ is interpreted according to its semantics, as defined in the Meta Language through the function $eval$.

However, only specific commands in the Proof Language are applicable to any given expression. Consider the case where the second definition of $eval$, defined as $eval\ (Add\ x\ y) = eval\ x + eval\ y$, is incorrectly applied to the initial expression:

$$
\begin{aligned}
& exec\ c\ (eval\ (Val\ n) : s) \\
= \quad & \{\ \text{eval/2}\ \} \\
& exec\ c\ (eval\ (Val\ n) : s)
\end{aligned}
$$

When this command is applied, no match is found for $eval\ (Add\ x\ y)$ in the expression. As a result, the command is unable to apply the intended rewrite rule, and therefore the expression does not undergo any transformation. In such cases, an exception message should be provided, detailing why the command was unable to transform the expression.

Introducing Definitions

In addition to a command that applies the semantics of a source Object Language expression within the Meta Language, the Proof Language also includes a command to introduce a definition. Introducing a definition through the Proof Language serves as a transformation rule, modifying an expression in the Meta Language accordingly.

Consider the following expression from the Meta Language:

$$exec\ c\ (n : s)$$

This expression does not yet match the required form $exec\ c'\ s$ for some target code c'. To continue, we would need to solve the equation:

$$exec\ c'\ s = exec\ c\ (n : s)$$

To do this, we can introduce a new constructor $PUSH :: Int \to Op$ into the target language that binds the variable n. This constructor is then appended to the code continuation c, leading to the following definition:

$$exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s)$$

This definition can be applied to the Meta Language expression using the Proof Language command $define$, followed by the definition:

$$
\begin{aligned}
& exec\ c\ (n : s) \\
= \quad & \{\ \text{define: } exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s)\ \} \\
& exec\ (PUSH\ n : c)\ s
\end{aligned}
$$

When this command is used, the definition $exec\ (PUSH\ n : c)\ s = exec\ c\ (n : s)$ is introduced as a transformation rule. This rule can be expressed as a rewrite rule from the right-hand side to the left-hand side:

$$exec\ c\ (n : s) \rightarrow exec\ (PUSH\ n : c)\ s$$

Upon applying the command to a Meta Language expression, the left-hand side of the rewrite rule $exec\ c\ (n : s)$ is matched with any part of the expression. In this case, it matches the entire expression, transforming it by replacing $exec\ c\ (n : s)$ with $exec\ (PUSH\ n : c)\ s$.

However, as we will explore in the technical model, relying solely on this form of transformation – using the introduced definition as a rewrite rule from the right-hand side to the left-hand side – may be insufficient in practice. One issues is that certain definitions, when inverted to a rewrite rule, such as $exec\ [HALT]\ s = s$, can lead to ambiguity when the left-hand side of the rewrite rule s is matched against an expression in the Meta Language.

Restrictions on Introduced Definitions

To ensure that definitions introduced using the $define$ command are well-formed, they must adhere to the following restrictions:

- **No Free Variables**: Every variable appearing on the right-hand side of an introduced definition must be either:
  - Explicitly bound by the pattern on the left-hand side of the definition, or
  - Bound within the body of the definition.

  For example, the following definition is not well-formed because n is a free variable in the body of the definition:

  $$exec\ (PUSH : c)\ s = exec\ c\ (n : s)$$

- **Well-Defined Left-Hand side Pattern**: The left-hand side of the definition must be a well-defined pattern that does not involve function calls. Consider this problematic definition:

  $$fail\ (HAN\ (compile'\ h\ c) : s) = exec\ (compile'\ h\ c)\ s)$$

This definition is not well-formed because it includes a function call, *compile'*, within the pattern on the left-hand side, making the pattern unclear and improperly defined.

- **Consitency**: The introduction of new definitions must not conflict with existing ones. They should be consistent with previously introduced definitions to avoid ambiguity.

  For instance, if a definition like the following had already been introduced:

  $$fail\ (HAN\ c : s) = exec\ c\ s$$

  Introducing a new definition such as:

  $$fail\ (HAN\ c : s) = fail\ s$$

  would conflict with the previous one, as it provides a different transformation for the same pattern, leading to inconsistency.

Applying the Induction Hypothesis

In the context of calculating compilers, the induction hypothesis is a critical component of the proof process. The Proof Language facilitates this by providing a command that allows the transformation of expressions in the Meta Language based on the induction hypothesis. This transformation is grounded in the provided specification.

Consider the following specification expressed in the Proof Language:

$$forall\ e\ c\ s.\ \ exec\ (compile'\ e\ c)\ s = exec\ c\ (eval\ e : s)$$

The induction hypothesis is applied by transforming this equation into a rewrite rule from the right-hand side to the left-hand side:

$$exec\ c\ (eval\ e : s) \rightarrow exec\ (compile'\ e\ c)\ s$$

When the command *induction* is used, followed by specifying a variable to which the induction hypothesis should be applied, it transforms an expression in the Meta Language according to this rewrite rule.

Consider the Meta Language expression:

$$exec\ (ADD : c)\ (eval\ y : eval\ x : s)$$

Applying the command *induction* with the target variable y, yields the following result:

$$\begin{aligned} & exec\ (ADD : c)\ (eval\ y : eval\ x : s) \\ = \quad & \{ \text{ induction } e = y\ \} \\ & exec\ (compile'\ y\ (ADD : c))\ (eval\ x : s) \end{aligned}$$

In this scenario, the left-hand side of the rewrite rule $exec\ c\ (eval\ e : s)$, is matched with the expression $exec\ (ADD : c)\ (eval\ y : eval\ x : s)$, resulting in the following mappings:

$$\{\ c \mapsto (ADD : c),\ e \mapsto y,\ s \mapsto eval\ x : s\ \}$$

These mappings specifies that e correspond to y, which is consistent with the specified target variable $e = y$ in the command. Since they are consistent, the mappings are applied to the right-hand side of the rewrite rule, resulting in:

$$exec\ (compile'\ y\ (ADD : c))\ (eval\ x : s)$$

However, using the command *induction* with the target variable x for the same expression, would lead to ambiguity. The mapping for e derived from the expression is $\{\ e \mapsto y\ \}$, while it has been specified as $e = x$. This conflict indicates that applying the induction hypothesis with x as the target variable would be incorrect in this context. As a result, the expression should not undergo any transformation if the target variable had been specified as $e = x$.

To maintain the integrity of transformations during the proof process, the *induction* command must be applied comprehensively. Specifically, the induction hypothesis should be applied to all relevant sub-expression of the source Object Language expression under consideration. For example, in the case of the Object Language expression $Add\ x\ y$, the induction hypothesis should be applied to both x and y because each represents a sub-expression.

It is important to note that the *induction* command only applies a transformation for one instance of the induction hypothesis at a time. This

means that the command itself does not guarantee that all necessary transformations have been applied to all sub-expression. Therefore, it is essential to verify, after the fact, that the *induction* command has been applied to all relevant sub-expression to ensure the proof's correctness.

Applying the Distributive Property

In compiler calculations for a source language that includes a conditional if-then-else operator, an important transformation is the use of the distributive property. Distributivity allows functions and operators to be applied across the branches of the conditional operator.

This property can be described as follows:

$$f\ (\textbf{if}\ e\ \textbf{then}\ e1\ \textbf{else}\ e2) = \textbf{if}\ e\ \textbf{then}\ f\ e1\ \textbf{else}\ f\ e2$$

For example, consider the following expression in the Meta Language:

$$exec\ c\ ((\textbf{if}\ eval\ z == 0\ \textbf{then}\ eval\ y\ \textbf{else}\ eval\ x) : s)$$

Using the distributive property, this expression can be rewritten as:

$$exec\ c\ (\textbf{if}\ eval\ z == 0\ \textbf{then}\ (eval\ y : s)\ \textbf{else}\ (eval\ x : s))$$

A Proof Language command like *distribute* could be used to justify such transformations:

$$\begin{array}{ll} & exec\ c\ ((\textbf{if}\ eval\ z == 0\ \textbf{then}\ eval\ y\ \textbf{else}\ eval\ x) : s) \\ = & \quad \{\ \text{distribute}\ (:\ s)\ \text{over conditionals}\ \} \\ & exec\ c\ (\textbf{if}\ eval\ z == 0\ \textbf{then}\ (eval\ y : s)\ \textbf{else}\ (eval\ x : s)) \end{array}$$

However, implementing a *distribute* command that applies the distributive property as a general rule can be challenging in practice, as it might not accurately match a specific expression. Therefore, it may be more practical to define distributive rules that directly apply to the expressions encountered in the calculations.

Simplifying Case Expressions

When calculating compilers, an intermediate step might involve simplifying calculations, particularly calculations involving case expressions. The Proof Language provides a *simplify* command that handles such transformations.

Consider the following expression in the Meta Language, which includes nested case expressions:

$$
\begin{array}{l}
\textbf{case } (\textbf{case } \mathit{eval}\ x \textbf{ of} \\
\qquad\qquad \begin{array}{ll} \mathit{Just}\ n & \rightarrow \textbf{case } \mathit{eval}\ y \textbf{ of} \\
 & \qquad \begin{array}{ll} \mathit{Just}\ m & \rightarrow \mathit{Just}\ (n + m) \\ \mathit{Nothing} & \rightarrow \mathit{Nothing} \end{array} \\
\mathit{Nothing} & \rightarrow \mathit{Nothing}) \textbf{ of} \end{array} \\
\quad \begin{array}{ll} \mathit{Just}\ n & \rightarrow \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
\mathit{Nothing} & \rightarrow \mathit{fail}\ s \end{array}
\end{array}
$$

Applying the *simplify* command to this expression transforms it by collapsing the nested case expressions. Specifically, it simplifies the expression by directly applying the logic of the outer case expression to the results of the inner case expressions, yielding the following expression in the Meta Language:

$$
\begin{array}{l}
\textbf{case } \mathit{eval}\ x \textbf{ of} \\
\quad \begin{array}{ll} \mathit{Just}\ n & \rightarrow \textbf{case } \mathit{eval}\ y \textbf{ of} \\
 & \qquad \begin{array}{ll} \mathit{Just}\ m & \rightarrow \mathit{exec}\ c\ (\mathit{VAL}\ (n + m) : s) \\ \mathit{Nothing} & \rightarrow \mathit{fail}\ s \end{array} \\
\mathit{Nothing} & \rightarrow \mathit{fail}\ s \end{array}
\end{array}
$$

Similarly, consider the following expression:

$$
\begin{array}{l}
\textbf{case } (\mathit{Nothing}) \textbf{ of} \\
\quad \begin{array}{ll} \mathit{Just}\ n & \rightarrow \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
\mathit{Nothing} & \rightarrow \mathit{fail}\ s \end{array}
\end{array}
$$

Applying the *simplify* command to this expression simplifies it by directly collapsing the case structure. Since the inner expression is *Nothing*, the outer case expression immediately matches the *Nothing* branch, yielding the following simplified expression in the Meta Language:

$fail\ s$

3.1.5 Finalizing the Proof

After all cases have been proven, the proof can be finalized using the *QED* command from the Proof Language, indicating that the proof is complete. The *QED* command checks whether there is a case for every syntactical element in the source Object Language, and if so, concludes the proof.

3.1.6 Overview of Commands in the Proof Language

The Proof Language provides a set of commands designed to facilitate various steps in compiler calculations. These commands operate on expressions in the Meta Language to ensure that transformations adhere to rules derived from the specification, the semantics of the source Object Languages, and definitions introduced during the transformation process. By manipulating expressions in the Meta Language with these commands, the semantics of the source and target Object Languages are preserved throughout the transformation process.

The Proof Language commands can be divided into three different categories:

- **Specification**: This command defines the correctness of the compiler by specifying its expected behavior.
- **Justifications**: These commands apply specific transformations to expressions in the Meta Language, ensuring that each transformation adheres to a specific rule.
  - $eval/x$: Applies a specific semantic rule x to an expression in the Meta Language.
  - $define$: Introduces and applies a definition to an expression in the Meta Language.
  - $induction$: Applies the induction hypothesis to a target variable within an expression in the Meta Language.

  - *distribute*: Applies the distributive property across conditional branches within an expression in the Meta Language.
  - *simplify*: Simplifies a case expression in the Meta Language.

- **Proof Directives**: These commands structure and guide the proof process.

  - *Let*: Sets the context by specifying the expression in the source Object Language that is being analyzed.
  - *QED*: Concludes the proof and verifies that all cases have been addressed.

3.2 Technical Model of CoCa

The technical model of CoCa examines its internal architecture, highlighting the components that enable the verification of compiler calculations. This section explores how the various components of CoCa interact to facilitate the verification process.

3.2.1 Technical Overview

CoCa (Compiler Calculator) is a specialized tool for compiler calculations. It is composed of four distinct components:

- Lexical Analyzer
- Parser
- Term Rewriting System
- Proof system

When a source file is provided, the Lexical Analyzer tokenizes the input, breaking it down into fundamental units like keywords, identifiers, and operators (Marlow et al., 2022a). These tokens establish the lexical structure of CoCa's source language.

The Parser processes the sequence of tokens using productions rules defined by a context-free grammar to construct an Abstract Syntax Tree (AST) that represents the syntactic structure of the source code (Marlow et al., 2022b).

The Proof System utilizes the AST to verify compiler calculations, and the Term Rewriting System, guided by the Proof System, then performs transformations on the AST based on a set of rewrite rules.

Together, the Lexical Analyzer and Parser define the source language of CoCa, while the Proof System and Term Rewriting System ensure the correctness and transformation of compiler calculations.

Additionally, CoCa incorporates I/O functionalities to handle input and output operations. Currently, CoCa focuses on verifying compiler calculations, with potential for future enhancements aimed at facilitating

the automation of compiler calculations.

3.2.2 Source Language

This section presents the syntax of the source language for CoCa, which shares many similarities to Haskell. The Haskell 2010 Language Report served as a key inspiration in developing the source language for CoCa (Marlow, 2010).

First, the notational conventions used to describe the source language are provided. This is followed by a description of the lexical structure of CoCa, detailing the syntax of tokens such as identifiers and operators. Finally, the context-free grammar, which defines various syntactic elements such as definitions, expressions and statements, is presented.

Notational Conventions

The following notational conventions are used to present the syntax of the source language:

[*pattern*]	optional
{ *pattern* }	zero or more repetitions
(*pattern*)	grouping
$pat_1 \mid pat_2$	choosing
$pat_{<pat'>}$	difference - elements generated by *pat*, excluding those generated by *pat′*
PascalCase	non-terminal syntax
`typewriter`	terminal syntax

Additionally, BNF-like syntax is used, with productions taking the form:

$$NonTerm \rightarrow Alt_1 \mid ... \mid Alt_n$$

This indicates that the non-terminal *NonTerm* can be replaced by any of the alternatives Alt_1 through Alt_n.

Lexical Structure

The lexical structure specifies how identifiers, literals, operators, and other tokens are formed in CoCa. These tokens are then processed by the parser to construct an AST according to a set of grammar rules.

The following outlines the lexical structure for some tokens. The complete lexical program structure is available in Appendix C.

Identifiers
Identifiers in CoCa can be divided into two main categories: *Reserved Identifiers* and *User-Defined Identifiers*.

The *Reserved Identifiers* serve as keywords in the language and have a special meaning in the context of CoCa. They cannot be used for other purposes, such as naming constructors or functions:

$$
\begin{array}{lll}
\textit{ReservedId} & \rightarrow & \texttt{induction} \mid \texttt{forall} \mid \texttt{QED} \mid \texttt{data} \mid \texttt{case} \mid \texttt{of} \\
 & \mid & \texttt{type} \mid \texttt{Let} \mid \texttt{ref} \mid \texttt{simplify} \mid \texttt{not}
\end{array}
$$

The *User-Defined Identifiers* are defined by the user and represents variables, functions and constructors. These are further categorized by the following rules:

$$
\begin{array}{lll}
\textit{BoundVarId} & \rightarrow & \texttt{\$}\ (\ \textit{VarId} \mid \textit{IConId} \mid \textit{ConId} \mid \textit{FunId}\ ) \\
\textit{VarId} & \rightarrow & \textit{Letter}\ \{\ \textit{Prime}\ \} \\
\textit{IConId} & \rightarrow & (\ \textit{Large Large}\ \{\ \textit{Large}\ \}\ )_{\langle \textit{ReservedId} \rangle} \\
\textit{ConId} & \rightarrow & (\ \textit{Large}\ \{\ \textit{Letter}\ \}\ \textit{Small}\ \{\ \textit{Letter}\ \}\ )_{\langle \textit{ReservedId} \rangle} \\
\textit{FunId} & \rightarrow & (\ \textit{Small FunIdElem}\ \{\ \textit{FunIdElem} \mid \textit{Prime}\ \}\ )_{\langle \textit{ReservedId} \rangle} \\
\textit{FunIdElem} & \rightarrow & \textit{Letter} \mid \textit{Digit} \mid \texttt{_} \mid \texttt{-} \mid \texttt{?}
\end{array}
$$

The *User-Defined Identifiers* have the following roles:

- *VarId*: Represents variable identifiers, starting with either a lower or uppercase letter, possibly followed by any number of prime characters (′).
    - Examples: x, y'', X, Y'

- *BoundVarId*: Represents bound variable identifiers, which use $ to distinguish them. A bound variable can take the form of any other *User-Defined Identifier*.
  - Examples: $\$x$, $\$M'$, $\$Bound$, $\$BOUND_X'$
- *ConId*: Represents constructor identifiers, starting with an uppercase letter, followed by any number of letters, with at least one lowercase letter included.
  - Examples: Val, Add, $Code$, Op
- *IConId*: Represents introduced constructor identifiers, which must contain at least two uppercase letters.
  - Examples: $PUSH$, $JUMP$, HAN
- *FunId*: Represents function identifiers, starting with a lowercase letter, followed by one *FunIdElem*, and then possibly additional *FunIdElem* elements or prime characters (′).
  - Examples: $exec$, $eval$, $compile'$

Operators
CoCa supports a variety of operator symbols, grouped into the following categories:

ReservedOp → *ArithmeticOp* | *EqualityOp* | *ComparisonOp*
  | *LogicalOp* | *ListOp* | *MiscOp*
ArithmeticOp → `+` | `-` | `*` | `/` | `^` | `^^` | `**`
EqualityOp → `==` | `/=` | `!=`
ComparisonOp → `<` | `<=` | `>` | `>=`
LogicalOp → `&&` | `||` | `~`
ListOp → `:` | `++`
MiscOp → `==>` | `<==` | `::` | `->` | `|`

These operators are recognized in source files but are not executed by CoCa itself. They are used to represent various operations and relations within the compiler calculations.

Whitespace and Newlines
In CoCa, the source language is sensitive to newlines, which play a significant role in structuring the content in source files. This means *NewLines* generates a token:

NewLines	→	*NewLine* { *NewLine* }
NewLine	→	*Return Linefeed* | *Return* | *Linefeed* | *Formfeed*
Return	→	`a carriage return`
Linefeed	→	`a line feed`
Formfeed	→	`a form feed`

However, whitespace is ignored. This also includes a *NewLine* followed by any amount of whitespace characters – spaces, horizontal tabs and vertical tabs – which is treated as indented code and does not generate a token. While *NewLines* produce a token, a *NewLine* followed by whitespace characters is ignored.

Expressions

Expressions are a crucial part of CoCa's source language, allowing various operations and expressions from compiler calculations to be represented. The grammar rules for expressions are defined as follows:

Exp	→	`-` *Exp*	(prefix neg)
	|	*LogicNeg Exp*	(logical neg)
	|	`case` *Exp* `of` `|` *Alt* { `|` *Alt* }	(case exp)
	|	*Exp BinOp Exp*	(binary exp)
	|	*Exp LogicOp Exp*	(logic exp)
	|	*FApp*	
Alt	→	*Exp* `->` *Exp*	
FApp	→	[*FApp*] *FAtom*	(fun app)
FAtom	→	`(` *Exp* `)`	(parenthesized)
	|	`[` Exp_1 `,` ... `,` Exp_n `]`	(list, n>=0)
	|	`(` Exp_1 `,` ... `,` Exp_n `)`	(tuple, n>=2)
	|	*EAtom*	(exp atom)

EAtom	$\rightarrow$	*EPower*	
	|	*VarId*	(variable)
	|	*BoundVarId*	(bound variable)
	|	*IConId*	(introduced con)
	|	*ConId*	(constructor)
	|	*FunId*	(fun name)
	|	*Integer*	(integer)
	|	*Float*	(floating point)
EPower	$\rightarrow$	*Integer Raised Integer*	(integer)
	|	*Float Raised* (*Integer* | *Float*)	(floating point)
Raised	$\rightarrow$	`^` | `^^` | `**`	
BinOp	$\rightarrow$	`+` | `-` | `*` | `/` | `:` | `++` | `==`	
	|	`/=` | `!=` | `<` | `<=` | `>` | `>=`	
LogicNeg	$\rightarrow$	`not` | $\sim$	
LogicOp	$\rightarrow$	`&&` | `||`	

Expressions in CoCa closely resemble those found in compiler calculations. For instance, an expression in compiler calculations such as:

$$exec\ c\ (eval\ (Val\ n) : s)$$

Would be represented identically in CoCa.

However, there are some differences, particularly with case expressions. In CoCa, a case expression delimits patterns with | as shown in the example below:

$$\begin{array}{l}\textbf{case}\ eval\ (Val\ n)\ \textbf{of} \\ \quad |\ Just\ \$n\ \ \rightarrow exec\ c\ (VAL\ \$n : s) \\ \quad |\ Nothing \rightarrow fail\ s\end{array}$$

For correct parsing, it is crucial that the cases in a case expression are either on the same line or indented. For instance, the following expression would produce a parse error because the patterns are separated by a newline without any indentation:

$$\begin{array}{l}\textbf{case}\ e_1\ \textbf{of} \\ |\ True\ \ \rightarrow e_2 \\ |\ False \rightarrow e_3\end{array}$$

In nested case expressions, parenthesizing is necessary:

$$
\begin{aligned}
&\textbf{case } e_1 \textbf{ of}\\
&\quad \mid \mathit{True} \ \ \rightarrow (\textbf{case } e_2 \textbf{ of}\\
&\qquad\qquad\qquad\quad \mid \mathit{True} \ \ \rightarrow e_3\\
&\qquad\qquad\qquad\quad \mid \mathit{False} \rightarrow e_4)\\
&\quad \mid \mathit{False} \rightarrow e_5
\end{aligned}
$$

Notably, CoCa does not have a direct if-then-else construct. However, this can be represented using a case expression since the following identity holds (Marlow, 2010):

$$
\begin{aligned}
\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 = {} & \textbf{case } e_1 \textbf{ of}\\
& \mid \mathit{True} \ \ \rightarrow e_2\\
& \mid \mathit{False} \rightarrow e_3
\end{aligned}
$$

Declarations

Declarations in CoCa are used to specify type synonyms and data declarations. The grammar for these declarations is defined as follows:

TopDecl	→	`type` *SimpleType* = *FunType*	(type decl)
	|	`data` *SimpleType* [= *Constrs*]	(data decl)

Below are some examples of type and data declarations in CoCa:

$$
\begin{aligned}
&\textbf{type } \mathit{Code} = [\mathit{Op}]\\
&\textbf{type } \mathit{Stack} = [\mathit{Elem}]\\
&\textbf{data } \mathit{Op}\\
&\textbf{data } \mathit{Expr} = \mathit{Val\ Int} \mid \mathit{Add\ ExprExpr}\\
&\textbf{data } \mathit{Maybe\ a} = \mathit{Just\ a} \mid \mathit{Nothing}
\end{aligned}
$$

Both the type synonyms and data declarations align with how types and data are represented in Haskell-like syntax. For the specific grammar rules governing *SimpleType*, *FunType*, and *Constr*, please refer to Appendix D.

Function Signatures

Function signatures in CoCa define the types associated with functions, The grammar for function signatures *TypeSig* is defined as follows:

TypeSig	$\rightarrow$	*TyFun* `::` *FunType*	(signature)
TyFun	$\rightarrow$	*FunId*	(fun name)

Examples of function signatures in CoCa include:

$$
\begin{aligned}
&eval &&:: Expr \rightarrow Int \\
&compile' &&:: Expr \rightarrow Code \rightarrow Code \\
&exec &&:: Code \rightarrow Stack \rightarrow Stack
\end{aligned}
$$

These function signatures bear a resemblance to Haskell-like syntax, though though with limitations to the expressiveness.

Definitions

Function definitions in CoCa are represented as equations, with optional references that can be used to refer to specific function definitions during the proof process. The grammar for function definitions is defined as follows:

FunDef	$\rightarrow$	*Equation* [`{ ref` *Ref* `}`]	(fun def)
Equation	$\rightarrow$	*Exp* = *Exp*	
Ref	$\rightarrow$	*Exp*	

An example of a function definition in CoCa is as follows:

$$eval\ (Val\ n) = n \qquad \{\textbf{ref}\ eval/1\}$$

In this example, the reference $eval/1$ is associated with this specific function definition. This reference can be used later in the proof process to justify steps based on this particular definition.

Specifications

Specifications in CoCa are used to capture the correctness of a compiler by defining formal statements that describe how the compiler should behave. The grammar for writing specifications is defined as follows:

CorEq	$\rightarrow$	`forall` *Vars* `.` *Equation*	(corr equation)
Vars	$\rightarrow$	$Var_1 \ldots Var_n$	(n >= 1)
Var	$\rightarrow$	*VarId*	(variable)

Examples of specifications in CoCa include:

$$\textbf{forall}\ e\ s.\ exec\ (compile\ e)\ s = eval\ e : s$$

$$\textbf{forall}\ e\ c\ s.\ exec\ (compile'\ e\ c)\ s = exec\ c\ (eval\ e : s)$$

These examples demonstrate how specifications can be used to formally assert the expected behavior of compilers, ensuring that the execution of compiled code produces the correct results as defined by the specifications.

However, it should be noted that the arrangement of expressions in the specification is significant. The right-hand side expression is used by CoCa as the initial expression during a proof case. For instance, if the compiler specification is defined as:

$$\textbf{forall}\ e\ s.\ eval\ e : s = exec\ (compile\ e)\ s$$

CoCa would expect the initial expression in a proof case to be:

$$exec\ (compile\ e)\ s$$

where e is replaced by the specific case being considered, such as $Val\ n$.

Statements

Statements in CoCa play a crucial role in directing the flow of transformations during compiler calculations. These statements, which include actions like applying an equation either from right-to-left or left-to-right, or simplifying expressions, relate to the justificational proof commands

outlined in the conceptual model of CoCa. The grammar for these statements is defined as follows:

Comment	$\rightarrow$	`=` { *Statement* }	
Statement	$\rightarrow$	`<==` (*Equation* | *Ref*)	(apply R to L)
	|	`==>` (*Equation* | *Ref*)	(apply L to R)
	|	`induction` *Equation*	(preset induction)
	|	`<== induction` *Equation*	(induction R to L)
	|	`==> induction` *Equation*	(induction L to R)
	|	`simplify`	(simplification)

Compared to the conceptual model of CoCa, these statements provide additional expressiveness. For example, it is possible to specify that induction should be applied either from left-to-right or from right-to-left. Additionally the *Statement* with induction where no direction is specified defaults to applying the induction hypotheses – provided by the specification – from right-to-left.

The *ProofEnd* can be used to mark the completion of a proof:

ProofEnd $\rightarrow$ `QED`

Newline Sensitivity

CoCa is sensitive to newline placement, which plays a crucial role in the correct parsing of source content. For instance, when defining the syntax and semantics of a source language for a compiler in a source file, each construct must be separated by at least one newline to avoid a parse error:

$$
\begin{array}{ll}
\textbf{data}\ Expr & = Val\ Int \mid Add\ Expr\ Expr\P \\
\P & \\
eval & ::\ Expr \rightarrow Int\P \\
eval\ (Val\ n) & = n\P \\
eval\ (Add\ x\ y) & = eval\ x\ +\ eval\ y\P
\end{array}
$$

In this example, if the constructs were not separated by a newline ¶, the parser would likely generate an error due to the lack of clear separation

between the definitions.

Additionally, throughout a proof in a source file, indentation affects how expressions are parsed. A newline combined with indentation is ignored by the lexer, meaning that the subsequent line is considered a continuation of the previous one rather than a new construct. For example:

$$
\begin{array}{l}
exec\ c\ (n : s)\P \\
=\ \{ \text{<==}\ exec\ (PUSH\ n : c)\ s = exec\ c(n : s)\ \}\P \\
\textvisiblespace\textvisiblespace\textvisiblespace exec\ (PUSH\ n : c)\ s
\end{array}
$$

In this example, the spaces ␣ before $exec\ (PUSH\ n : c)\ s$ lead to a parse error, as $exec\ (PUSH\ n : c)\ s$ is not parsed as an independent entity but is instead interpreted as a continuation of the *Comment* due to the missing newline token. This happens because when a *NewLine* ¶ is followed by whitespace characters such as spaces ␣ no newline token is produced.

Example of Compiler Calculations in CoCa

The following example demonstrates how a source file containing compiler calculations for a simple arithmetic expression language could be structured. The example shows how compiler calculations should be written in a format that CoCa understands, adhering to the lexical and grammatical rules described earlier:

$$\mathbf{data}\ Expr \quad = Val\ Int \mid Add\ Expr\ Expr$$

$$
\begin{array}{lll}
eval & ::\ Expr \rightarrow Int & \\
eval\ (Val\ n) & = n & \{\mathbf{ref}\ eval/1\} \\
eval\ (Add\ x\ y) & = eval\ x\ +\ eval\ y & \{\mathbf{ref}\ eval/2\}
\end{array}
$$

$$\mathbf{forall}\ e\ c\ s.\ exec\ (compile'\ e\ c)\ s = exec\ c\ (eval\ e : s)$$

$$\mathbf{Let}\ e = Val\ n$$

$$
\begin{array}{l}
exec\ c\ (eval\ (Val\ n) : s) \\
=\ \{ \text{==>}\ eval/1\ \} \\
exec\ c\ (n : s) \\
=\ \{ \text{<==}\ exec\ (PUSH\ n\ c)\ s = exec\ c\ (n : s)\ \}
\end{array}
$$

$exec\ (PUSH\ n\ c)\ s$

Let $e = Add\ x\ y$

$exec\ c\ (eval\ (Add\ x\ y) : s)$
$= \{$ ==> $eval/2\ \}$
$exec\ c\ ((eval\ x + eval\ y) : s)$
$= \{$ <== $exec\ (ADD\ c)\ (m : n : s) = exec\ c\ ((n+m) : s)\ \}$
$exec\ (ADD\ c)\ (eval\ y : eval\ x : s)$
$= \{$ induction $e = y\ \}$
$exec\ (comp\ y\ (ADD\ c))\ (evalx : s)$
$= \{$ induction $e = x\ \}$
$exec\ (comp\ x\ (comp\ y\ (ADD\ c)))\ s$

QED

Although some of the constructs here differ slightly in syntax from those in the conceptual model, they serve the same purpose.

3.2.3 Proof System

The Proof System is responsible for checking and verifying compiler calculations. It operates on an AST that contains the necessary information for this verification process.

A TRS is employed to perform the underlying transformations, based on the rules provided in the AST. The TRS enables the Proof System to execute step-wise transformations according to the applied proof commands (Statements). This design ensures that the Proof System explicitly controls the application of rules, guiding the transformations as needed. Thus, the TRS is integrated into the Proof System in a controlled manner.

The following sections offers insight into the technical details of the Proof System. First, the implementation of the TRS is examined, highlighting how matching, substitution, and rewriting are achieved. Following this, the process of verifying compiler calculations is detailed,

demonstrating how the Proof System performs this task while also considering alternative strategies.

Implementation of a Term Rewriting System

Recall that a term rewriting system can be described as a pair $\mathcal{R} = (\Sigma, R)$ of a signature Σ and a set of rewrite rules R for Σ.

In CoCa, the signature Σ is defined by the $Expr$ data type, which represents the various forms of expressions that can be manipulated by the term rewriting system. The $Expr$ type is derived from the production rules of the Exp grammar, which specifies the syntactic structure of expressions in CoCa. For example, the $Expr$ data type includes constructors such as $ENeg$, $EBinOp$, and $FApp$, each corresponding to a different kind of expression in the grammar:

$$
\begin{aligned}
&\textbf{data}\ Expr\\
&\quad = ENeg\ (Expr)\\
&\quad\ \mid ELogicNeg\ (Expr)\\
&\quad\ \mid ECaseOf\ (Expr)\ [Alt]\\
&\quad\ \mid EBinOp\ (BinOp)\ (Expr)\ (Expr)\\
&\quad\ \mid ELogicOp\ (LogicOp)\ (Expr)\ (Expr)\\
&\quad\ \mid FApp\ (Expr)\ (Expr)\\
&\quad\ \mid EList\ [Expr]\\
&\quad\ \mid ETuple\ [Expr]\\
&\quad\ \mid EAtom\ (Atom)
\end{aligned}
$$

Terms within the TRS can be formed from this $Expr$ type.

The set of rewrite rules R is initially empty, as there are no internal rewrite rules in the system. However, the rewrite rules are provided in the generated AST. For instance, after providing a source file to CoCa the source language semantics along with the specification – provided in the AST – will be transformed into rewrite rules, ready to be applied through *Statements*. Likewise, the definitions introduced during the proof process are also transformed and incorporated into the set of rewrite rules.

The functionality of the TRS is provided by three main algorithms:

- A Matching algorithm that returns a substitution σ, given two terms t and s such that $\sigma(t) \equiv s$, if one exists.
- A Substitution algorithm that applies a substitution σ to a term t, returning $\sigma(t)$, such that $\sigma(F(t_1, ..., t_n)) \equiv F(\sigma(t_1), ..., \sigma(t_n))$ is satisfied.
- A Rewrite algorithm that takes a rewrite rule $\rho : l \to r$ and a term t, applies the Matching algorithm with l and t, to get a substitution, such that $\sigma(l) \equiv t$. Then it applies the substitution σ to the rewrite rule ρ to obtain an atomic rewrite step: $\sigma(l) \to_\rho \sigma(r)$. Lastly, replaces all occurrences of the ρ-redex $\sigma(l)$ in the term t with its contractum $\sigma(r)$.

The three algorithms are explained in more detail below, providing insight into their implementation and role within the term rewriting system.

Matching

The Matching algorithm is responsible for finding a substitution σ that, when applied to a term t, makes it equivalent to another term s (i.e., $\sigma(t) \equiv s$).

In CoCa, this functionality is provided by the *match* function:

$$match :: (Expr,\ Expr) \to Maybe\ (Substitution)$$

The function takes a pair of terms (represented by the *Expr* type) and returns a *Maybe Substitution*, indicating whether a valid substitution exists. The *Maybe* type encapsulates the possibility of failure, with *Just* (*Substitution*) indicating success and *Nothing* indicating that no matching substitution could be found.

A substitution in this context is a mapping from variables *Var* to terms *Expr*, represented as:

type *Substitution* = *Map Var Expr*

The *match* function works by recursively traversing the structure of the terms $Expr's$ to find a suitable substitution.

In addition to standard matching, CoCa also addresses special cases through functions like *matchCases*, which specifically manage the matching of two case expressions. This function ensures that all patterns in case expressions are matched appropriately, even when bound variables are involved.

Substitution
The Substitution algorithm is responsible for applying a substitution σ to a term t such that $\sigma(t)$ is returned, satisfying the condition $\sigma(F(t_1, ..., t_n)) \equiv F(\sigma(t_1), ..., \sigma(t_n))$.

In CoCa, this functionality is provided by the $applySubstitution$ function:

$$applySubstitution :: Substitution \rightarrow Expr \rightarrow Expr$$

The $applySubstitution$ function takes a $Substitution$ and an $Expr$, recursively applying the substitution to the term structure. If a variable within the term matches a variable from the substitution, it is replaced by the corresponding expression from the substitution.

Rewriting
The Rewriting algorithm in CoCa is responsible for transforming terms according to a given rewrite rule. Given a term t and a rewrite rule $\rho : l \rightarrow r$, the algorithm proceeds as follows:

1. *Matching*: The algorithm attempts to match the left-hand side of the rewrite rule l with the term t.

2. *Application of Substitution*: If a matching substitution σ is found, the algorithm applies this substitution to both l and r of the rewrite rule. This provides an atomic rewrite step $\sigma(l) \rightarrow_\rho \sigma(r)$.

3. *Rewriting*: The algorithm then replaces all occurrences of the ρ-redex $\sigma(l)$ in the term t with its contractum $\sigma(r)$. This produces the rewritten term, effectively transforming t into a new term according to the rewrite rule.

This process is encapsulated in the *applyRewriteRule* function in CoCa, which enforces certain constraints, such as ensuring that the left-hand side l of the rewrite rule is not a variable and that every non-bound variable occurring in the right-hand side r also occurs in l. CoCa also includes a function *applyIntroducedRewriteRule*, which relaxes the constraint that all non-bound variables in r must appear in l. The rationale and implications of this relaxation are discussed in the subsequent section.

Verifying Compiler Calculations

At the core of the verification process is the Proof System. The Proof System operates on an abstract syntax tree (AST) generated from a source file.

To verify compiler calculations, the source file must contain the source language syntax and semantics, a specification, and calculations, written in accordance with the grammatical rules of CoCa.

The Proof System verifies each step of the calculations by systematically applying transformation rules and ensuring that each resulting expression aligns with the calculations from the source file.

Proof Initialization
The verification process begins with the *proofCheck* function, which initiates the verification process. This function takes as input the source language syntax and semantics, along with the specification and calculations provided. The system then collects the definitions $l = r$ of the source language semantics, transforming them into a list of rewrite rules $l \to r$.

Verifying each Proof Case
The calculations may be composed of multiple proof cases, each corresponding to a syntactical element of the source language. Each proof case must be verified. The *checkProofCases* function systematically processes these cases one by one. For each proof case, the system follows these steps:

1. *Apply the Case*: The specific case, such as $Val\ n$ or $Add\ x\ y$, is applied to the right-hand side of the specification. This step generates the initial expression that serves as the basis for subsequent transformations.

2. *Process Calculations*: The system then applies a series of transformations to the initial expression, based on the justifications ($Statement$s) provided in the calculations. For each transformation, the system checks the resulting expression against the corresponding calculation in the source file to verify that they align.

3. *Check Against Specification*: After processing the calculations, the resulting expression is checked against the specification. This step determines whether the final expression satisfies the specification, indicating that the proof case is correct. This requires that the final expression is in the required form, such as $exec\ c'\ s$ for some code c', though the exact form depends on the specification.

4. *Handle Exceptions*: If any part of the transformation process does not align with the calculations provided in the source file, an exception message is generated. This message provides detailed information on why the compiler calculations in the source file contain inconsistencies or errors, highlighting the specific step where the misalignment occurred.

Currently, the Proof System processes compiler calculations from top-to-bottom. However, introducing definitions during the calculations, such as:

$$= \ \{ <== fail\ (VAL\ n : s) = fail\ s\ \}$$

This definition is applied as a rewrite rule from right-to-left:

$$fail\ s \rightarrow fail\ (VAL\ n : s)$$

Here, the variable n is introduced, which violates the common constrain in TRS, that all variables on the right-hand side must appear on the left-hand side aswell. To address this, the $applyIntroducedRewriteRule$ function is included. While this approach can work in a controlled environment, it introduces the risk of non-termination, as the rule could be applied indefinitely, continuously adding $VAL\ n$. An alternative approach is to verify by traversing the calculations from bottom to top, where the rewrite rule would become:

$$fail\ (VAL\ n : s) \rightarrow fail\ s$$

This approach aligns more closely with how the definition is viewed but would not be suitable for facilitating the automation of compiler calculations, as the complete calculations would not be available.

Chapter 4

Reflections and Limitations

CoCa has demonstrated its capability in verifying compiler calculations for source languages including arithmetic expressions, exceptions, and state. The lexical analyzer, parser, proof system and term rewriting system has been necessary in ensuring the verification of these calculations. However, the current version of CoCa is primarily focused on verification rather than automation, leaving room for further development in this area.

When introducing definitions within a context of a case expressions, CoCa provides some expressiveness, as seen in the following example:

$$
\begin{array}{l}
\textbf{case}\ eval\ x\ \textbf{of} \\
\quad \mid Just\ \$n \rightarrow (\textbf{case}\ eval\ y\ \textbf{of} \\
\qquad\qquad\qquad \mid Just\ \$m \rightarrow exec\ (ADD\ c)\ (VAL\ \$m : VAL\ \$n : s) \\
\qquad\qquad\qquad \mid Nothing \rightarrow fail\ s) \\
\quad \mid Nothing \rightarrow fail\ s \\
= \ \{ <==\ fail\ (VAL\ \$n : s) = fail\ s\ \} \\
\textbf{case}\ eval\ x\ \textbf{of} \\
\quad \mid Just\ \$n \rightarrow (\textbf{case}\ eval\ y\ \textbf{of} \\
\qquad\qquad\qquad \mid Just\ \$m \rightarrow exec\ (ADD\ c)\ (VAL\ \$m : VAL\ \$n : s) \\
\qquad\qquad\qquad \mid Nothing \rightarrow fail\ (VAL\ \$n : s)) \\
\quad \mid Nothing \rightarrow fail\ s
\end{array}
$$

By defining the definition with an bound variable $\$n$, the corresponding rewrite rule applies only within the context where $eval\ x$ evaluates to $Just\ \$n$. However, this approach has a downside, as the definition

introduces a bound variable in its declaration, which is undesirable. Moreover, this technique does not work had $\$n$ been a concrete type.

The primary limitation of CoCa lies in its expressiveness, particularly in allowing definitions to be applied selectively within specific contexts rather than across the entire expression. This constraint limits CoCa's ability to handle more complex or varied steps in compiler calculations.

Chapter 5

Conclusion and Outlook on Future Work

This thesis has introduced and developed CoCa, a specialized tool designed to verify compiler calculations for source languages that include arithmetic expressions, exceptions, and state. By utilizing a proof system that derives and applies a set of transformational rules through a term rewriting system, CoCa ensures that each step in the compiler calculations adheres to these rules. If these rules are defined and applied such that each case in the compiler calculations meets the specification (achieves the required form), the proof is verified to be correct.

While CoCa demonstrates its effectiveness in verifying compiler calculations for a variety of source languages, there are several opportunities for further development to enhance and expand its capabilities.

1. Partially Automating Compiler Calculations:
Currently, CoCa is limited to verifying compiler calculations. A significant area for future work involves extending CoCa to facilitate the automation of compiler calculations. This would require enabling CoCa to manage and manipulate intermediate states during the proof process, as well as incorporating additional I/O functionality. However, the capability of applying a transformation based on a specific justification step is already present in CoCa.

2. Deriving Definitions and Type Inference:
Another important extension involves enhancing CoCa's ability to derive and provide output with the definitions of both the compiler and the virtual machine from the calculations. While deriving these

definitions is relatively straightforward, determining the types of introduced constructors presents a more complex challenge. To address this, CoCa would need to incorporate a type inference system capable of identifying the types of constructors introduced during the compiler calculations. This system would allow CoCa to correctly determine whether a constructor belongs to, for example, the *Op* type or the *Elem* type. One possible approach could involve requiring all function signatures to be specified in the source file, allowing these signatures to be used in the type inference process.

3. Expanding Capabilities to Handle Advanced Language Features:
Another area for future work is to expand CoCa's capabilities to handle source languages that include advanced language features such as lambda calculi. This expansion would require significant changes or extensions to all components of CoCa, including the parser, term rewriting system and proof system. Enhancing CoCa in this way would increase its capability, enabling it to support a broader range of compiler calculations.

In conclusion, this thesis contributes to the domain of compiler calculations by presenting CoCa, a specialized and more accessible alternative to generalist tools like Coq and Agda. While acknowledging its current limitations, CoCa demonstrates the feasibility of verifying compiler calculations for a variety of source languages, reducing the need for more complex tools. The outlined future work offers clear directions for enhancing and expanding CoCa, paving the way for its progression from its current implementation into a more robust tool for compiler calculations.

References

Backhouse, Roland. (2003). *Program Construction: Calculating Implementations from Specifications*. John Wiley & Sons, Ltd.

Bahr, Patrick & Hutton, Graham (2015). *Calculating Correct Compilers*. Journal of Functional Programming, 25(e14).

Hutton, Graham (2016). *Programming in Haskell*. Second Edition, Cambridge University Press.

McCarthy, John & Painter, James (1967). *Correctness of a Compiler for Arithmetic Expressions*. (a reprint with minor changes of "Correctness of a Compiler for Arithmetic Expressions" by John McCarthy and James Painter published in *Mathematical Aspects of Computer Science 1*, Volume 19 of Proceedings of Symposia in Applied Mathematics by the American Mathematical Society in 1967).

Marlow, S., Abel, A., Ericson, J., Dornan, C., Jones, I., & Alex developers. (2022a). *Alex*. Online manual (Revision 48f6cfb6). https://haskell-alex.readthedocs.io/en/latest/introduction.html

Marlow, S. Gill, A., Abel, A., Zavialov, V., Ericson, J., & Happy developers. (2022b). *Using Happy*. Online manual (Revision a45e10ea). https://haskell-happy.readthedocs.io/en/latest/using.html

Marlow, Simon (2010). *The Haskell 2010 Language (Part 1)*. Cambridge, April 2010. Online report: https://www.haskell.org/onlinereport/haskell2010/haskellpa1.html#x5-8000I

Morris, F. Lockwood (1973). *Advice on Structuring Compilers and Proving them correct*. Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages. Pages 144-152. 1 October 1973.

Scott, Dana S., & Strachey, Christopher (1971). *Towards a Mathematical Semantics for Computer Languages*. Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group.

Terese (2003). *Term Rewriting Systems*. Manuscript used in the course Termherschrijfsystemen given at the VU, spring 2003. Contributors to the manuscript: Erik Barendsen, Inge Bethke, Marc Bezem, Jan Heering, Richard Kennaway, Paul Klint, Jan Willem Klop, Vincent van Oostrom, Femke van Raamsdonk, Fer-Jan de Vries, Roel de Vrijer and Hans Zantema.

Appendix A - *Calculations for Source Language with Exceptions*

In this appendix, the complete compiler calculations for a source language with primitives for handling exceptions are presented.

Source Language Syntax:

```
data Expr = Val Int
          | Add Expr Expr
          | Ite Expr Expr Expr
          | Throw
          | Catch Expr Expr
```

Source Language Semantics:

```
eval               :: Expr → Maybe Int
eval (Val n)       = Just n
eval (Add x y)     = case eval x of
                       Just n    → case eval y of
                                     Just m    → Just (n + m)
                                     Nothing → Nothing
                       Nothing → Nothing
eval (Ite z x y)   = case eval z of
                       Just n    → if n == 0 then eval y else eval x
                       Nothing → Nothing
eval (Throw)       = Nothing
eval (Catch x h)   = case eval x of
                       Just n    → Just n
                       Nothing → eval h
```

Top-Level Compilation:

```
compile :: Expr → Code
```

Generalized Compilation:

```
compile′ :: Expr → Code → Code
```

Target Language:

```
type Code = [Op]
data Op
```

Virtual Machine:

$exec :: Code \rightarrow Stack \rightarrow Stack$
$fail :: Stack \rightarrow Stack$

The Stack Type:

$\textbf{type}\ Stack = [Elem]$
$\textbf{data}\ Elem = VAL\ Int$

Specification for *compile*:

$$exec\ (compile\ e)\ s = \textbf{case}\ eval\ e\ \textbf{of}\quad Just\ n \rightarrow VAL\ n : s;\quad Nothing \rightarrow fail\ s$$

Specification for *compile'*:

$$exec\ (compile'\ e\ c)\ s = \textbf{case}\ eval\ e\ \textbf{of}\quad Just\ n \rightarrow exec\ c\ (VAL\ n : s);\quad Nothing \rightarrow fail\ s$$

Calculations

Base case, $e = Val\ n$:

$exec\ (compile'\ (Val\ n)\ c)\ s$
$=$ { specification for *compile'* }
$\textbf{case}\ eval\ (Val\ n)\ \textbf{of}$
$\quad Just\ n \rightarrow exec\ c\ (VAL\ n : s)$
$\quad Nothing \rightarrow fail\ s$
$=$ { apply definition of *eval* }
$exec\ c\ (VAL\ n : s)$
$=$ { define: $exec\ (PUSH\ n : c)\ s = exec\ c\ (VAL\ n : s)$ }
$exec\ (PUSH\ n : c)\ s$

Base case, $e = Throw$:

$exec\ (compile'\ (Throw)\ c)\ s$
$=$ { specification for *compile'* }
$\textbf{case}\ eval\ (Throw)\ \textbf{of}$
$\quad Just\ n \rightarrow exec\ c\ (VAL\ n : s)$

$$
\begin{aligned}
&\quad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
= &\quad \{ \text{ apply definition of } \mathit{eval} \ \} \\
&\mathit{fail}\ s \\
= &\quad \{ \text{ define: } \mathit{exec}\ [\mathit{FAIL}]\ s = \mathit{fail}\ s \ \} \\
&\mathit{exec}\ [\mathit{FAIL}]\ s
\end{aligned}
$$

Inductive case, $e = Add\ x\ y$:

$$
\begin{aligned}
&\mathit{exec}\ (\mathit{compile}'\ (\mathit{Add}\ x\ y)\ c)\ s \\
= &\quad \{ \text{ specification for } \mathit{compile}' \ \} \\
&\textbf{case}\ \mathit{eval}\ (\mathit{Add}\ x\ y)\ \textbf{of} \\
&\quad \mathit{Just}\ n \quad \rightarrow \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
&\quad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
= &\quad \{ \text{ apply definition of } \mathit{eval} \ \} \\
&\textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
&\quad \mathit{Just}\ n \quad \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
&\qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : s) \\
&\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
&\quad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
= &\quad \{ \text{ define: } \mathit{exec}\ (\mathit{ADD} : c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
&\qquad\qquad = \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : s) \ \} \\
&\textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
&\quad \mathit{Just}\ n \quad \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
&\qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ (\mathit{ADD} : c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
&\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
&\quad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
= &\quad \{ \text{ define: } \mathit{fail}\ (\mathit{VAL}\ n : s) = \mathit{fail}\ s \ \} \\
&\textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
&\quad \mathit{Just}\ n \quad \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
&\qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ (\mathit{ADD} : c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
&\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fail}\ (\mathit{VAL}\ n : s) \\
&\quad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
= &\quad \{ \text{ Induction hypothesis for } y \ \} \\
&\textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
&\quad \mathit{Just}\ n \quad \rightarrow \mathit{exec}\ (\mathit{compile}'\ y\ (\mathit{ADD} : c))\ (\mathit{VAL}\ n : s) \\
&\quad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
= &\quad \{ \text{ Induction hypothesis for } x \ \} \\
&\mathit{exec}\ (\mathit{compile}'\ x\ (\mathit{compile}'\ y\ (\mathit{ADD} : c)))\ s
\end{aligned}
$$

Inductive case, $e = Ite\ z\ x\ y$:

```
  exec (compile′ (Ite z x y) c) s
=   { specification for compile′ }
  case eval (Ite z x y) of
    Just n    → exec c (VAL n : s)
    Nothing → fail s
=   { apply definition of eval }
  case eval z of
    Just n    → if n == 0
                  then case eval y of
                         Just m    → exec c (VAL m : s)
                         Nothing → fail s
                  else case eval x of
                         Just m    → exec c (VAL m : s)
                         Nothing → fail s
    Nothing → fail s
=   { induction hypothesis for x }
  case eval z of
    Just n    → if n == 0
                  then case eval y of
                         Just m    → exec c (VAL m : s)
                         Nothing → fail s
                  else exec (compile′ x c) s
    Nothing → fail s
=   { induction hypothesis for y }
  case eval z of
    Just n    → if n == 0
                  then exec (compile′ y c) s
                  else exec (compile′ x c) s
    Nothing → fail s
=   { define: exec (JUMP c″ : c) (VAL n : s) = if n== 0
                                                 then exec c s
                                                 else exec c″ s }
  case eval z of
    Just n    → exec (JUMP (compile′ x c) : (compile′ y c)) (VAL n : s)
    Nothing → fail s
=   { induction hypothesis for z }
  exec (compile′ z (JUMP (compile′ x c) : (compile′ y c))) s
```

Inductive case, $e = Catch\ x\ h$:

$$
\begin{aligned}
& \mathit{exec}\ (\mathit{compile}'\ (\mathit{Catch}\ x\ h)\ c)\ s \\
= \quad & \{ \text{ specification for } \mathit{compile}' \ \} \\
& \textbf{case}\ \mathit{eval}\ (\mathit{Catch}\ x\ h)\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \to \mathit{fail}\ s \\
= \quad & \{ \text{ apply definition of } \mathit{eval} \ \} \\
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \to \textbf{case}\ \mathit{eval}\ h\ \textbf{of} \\
& \qquad\qquad\qquad \mathit{Just}\ m \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ m : s) \\
& \qquad\qquad\qquad \mathit{Nothing} \to \mathit{fail}\ s \\
= \quad & \{ \text{ induction hypothesis for } h \ \} \\
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \to \mathit{exec}\ (\mathit{compile}'\ h\ c)\ s \\
= \quad & \{ \text{ define: } \mathit{fail}\ (\mathit{HAN}\ c' : s) = \mathit{exec}\ c'\ s \ \} \\
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{HAN}\ (\mathit{compile}'\ h\ c) : s) \\
= \quad & \{ \text{ define: } \mathit{exec}\ (\mathit{UNMARK} : c)\ (\mathit{VAL}\ n : \mathit{HAN}\ _ : s) \\
& \qquad\quad = \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \ \} \\
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to \mathit{exec}\ (\mathit{UNMARK} : c)\ (\mathit{VAL}\ n : \mathit{HAN}\ (\mathit{compile}'\ h\ c) : s) \\
& \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{HAN}\ (\mathit{compile}'\ h\ c) : s) \\
= \quad & \{ \text{ induction hypothesis for } x \ \} \\
& \mathit{exec}\ (\mathit{compile}'\ x\ (\mathit{UNMARK} : c))\ (\mathit{HAN}\ (\mathit{compile}'\ h\ c) : s) \\
= \quad & \{ \text{ define: } \mathit{exec}\ (\mathit{MARK}\ c' : c)\ s = \mathit{exec}\ c\ (\mathit{HAN}\ c' : s) \ \} \\
& \mathit{exec}\ (\mathit{MARK}\ (\mathit{compile}'\ h\ c) : (\mathit{compile}'\ x\ (\mathit{UNMARK} : c)))\ s
\end{aligned}
$$

Top-level compilation function:

$$
\begin{aligned}
& \mathit{exec}\ (\mathit{compile}\ e)\ s \\
= \quad & \{ \text{ specification for } \mathit{compile} \ \} \\
& \textbf{case}\ \mathit{eval}\ e\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to \mathit{VAL}\ n : s \\
& \quad \mathit{Nothing} \to \mathit{fail}\ s \\
= \quad & \{ \text{ define: } \mathit{exec}\ [\mathit{HALT}]\ s = s \ \} \\
& \textbf{case}\ \mathit{eval}\ e\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to \mathit{exec}\ [\mathit{HALT}]\ (\mathit{VAL}\ n : s)
\end{aligned}
$$

$$
\begin{aligned}
&\quad\; \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
= &\quad \{ \text{ specification for } \mathit{compile}' \ \} \\
&\; \mathit{exec}\ (\mathit{compile}'\ e\ [\mathit{HALT}])\ s
\end{aligned}
$$

Derived Definitions

Target Language:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Code} = [\mathit{Op}] \\
&\textbf{data}\ \mathit{Op} \ = \mathit{PUSH}\ \mathit{Int} \\
&\qquad\qquad \mid \mathit{FAIL} \\
&\qquad\qquad \mid \mathit{ADD} \\
&\qquad\qquad \mid \mathit{JUMP}\ \mathit{Code} \\
&\qquad\qquad \mid \mathit{MARK}\ \mathit{Code} \\
&\qquad\qquad \mid \mathit{UNMARK} \\
&\qquad\qquad \mid \mathit{HALT}
\end{aligned}
$$

Compiler:

$$
\begin{aligned}
&\mathit{compile} && :: \mathit{Expr} \rightarrow \mathit{Code} \\
&\mathit{compile}\ e && = \mathit{compile}'\ e\ [\mathit{HALT}] \\
\\
&\mathit{compile}' && :: \mathit{Expr} \rightarrow \mathit{Code} \rightarrow \mathit{Code} \\
&\mathit{compile}'\ (\mathit{Val}\ n)\ c && = \mathit{PUSH}\ n : c \\
&\mathit{compile}'\ (\mathit{Throw})\ c && = [\mathit{FAIL}] \\
&\mathit{compile}'\ (\mathit{Add}\ x\ y)\ c && = \mathit{compile}'\ x\ (\mathit{compile}'\ y\ (\mathit{ADD} : c)) \\
&\mathit{compile}'\ (\mathit{Ite}\ z\ x\ y)\ c && = \mathit{compile}'\ z\ (\mathit{JUMP}\ (\mathit{compile}'\ x\ c) : (\mathit{compile}'\ y\ c)) \\
&\mathit{compile}'\ (\mathit{Catch}\ x\ h)\ c && = \mathit{MARK}\ (\mathit{compile}'\ h\ c) : (\mathit{compile}'\ x\ (\mathit{UNMARK} : c))
\end{aligned}
$$

Virtual Machine:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Stack} = [\mathit{Elem}] \\
&\textbf{data}\ \mathit{Elem} = \mathit{VAL}\ \mathit{Int} \mid \mathit{HAN}\ \mathit{Code}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{exec} && :: \mathit{Code} \rightarrow \mathit{Stack} \rightarrow \mathit{Stack} \\
&\mathit{exec}\ [\mathit{HALT}]\ s && = s \\
&\mathit{exec}\ (\mathit{PUSH}\ n : c)\ s && = \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
&\mathit{exec}\ [\mathit{FAIL}]\ s && = \mathit{fail}\ s \\
&\mathit{exec}\ (\mathit{ADD} : c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) && = \mathit{exec}\ c\ (\mathit{VAL}\ (n + m) : s) \\
&\mathit{exec}\ (\mathit{JUMP}\ c'' : c)\ (\mathit{VAL}\ n : s) && = \textbf{if}\ n == 0 \\
& && \qquad \textbf{then}\ \mathit{exec}\ c\ s \\
& && \qquad \textbf{else}\ \mathit{exec}\ c''\ s \\
&\mathit{exec}\ (\mathit{UNMARK} : c)\ (\mathit{VAL}\ n : \mathit{HAN}\ _ : s) && = \mathit{exec}\ c\ (\mathit{VAL}\ n : s)
\end{aligned}
$$

$$
\begin{aligned}
&exec\ (MARK\ c' : c)\ s && = exec\ c\ (HAN\ c' : s) \\
\\
&fail && :: Stack \rightarrow Stack \\
&fail\ (VAL\ n : s) && = fail\ s \\
&fail\ (HAN\ c' : s) && = exec\ c'\ s
\end{aligned}
$$

Appendix B - *Calculations for Source Language with State*

In this appendix, the complete compiler calculations for a source language with primitives for handling state are presented.

Source Language Syntax:

$$
\begin{aligned}
\textbf{data}\ Expr\ &= Val\ Int\\
&\mid Add\ Expr\ Expr\\
&\mid Ite\ Expr\ Expr\ Expr\\
&\mid Throw\\
&\mid Catch\ Expr\ Expr\\
&\mid Get\\
&\mid Put\ Expr\ Expr
\end{aligned}
$$

Source Language Semantics:

$$
\begin{aligned}
&eval :: Expr \to State \to (Maybe\ Int,\ State)\\
&eval\ (Val\ n)\ q = (Just\ n, q)\\
&eval\ (Add\ x\ y)\ q = \textbf{case}\ eval\ x\ q\ \textbf{of}\\
&\qquad (Just\ n, q') \to \textbf{case}\ eval\ y\ q'\ \textbf{of}\\
&\qquad\qquad (Just\ m, q'') \to (Just\ (n+m), q'')\\
&\qquad\qquad (Nothing, q'') \to (Nothing, q'')\\
&\qquad (Nothing, q') \to (Nothing,\ q')\\
&eval\ (Ite\ z\ x\ y)\ q = \textbf{case}\ eval\ z\ q\ \textbf{of}\\
&\qquad (Just\ n, q') \to \textbf{if}\ n == 0\ \textbf{then}\ eval\ y\ q'\ \textbf{else}\ eval\ x\ q'\\
&\qquad (Nothing, q') \to (Nothing, q')\\
&eval\ (Throw)\ q = (Nothing, q)\\
&eval\ (Catch\ x\ h)\ q = \textbf{case}\ eval\ x\ q\ \textbf{of}\\
&\qquad (Just\ n, q') \to (Just\ n, q')\\
&\qquad (Nothing, q') \to eval\ h\ q'\\
&eval\ (Get)\ q = (Just\ q, q)\\
&eval\ (Put\ x\ y)\ q = \textbf{case}\ eval\ x\ q\ \textbf{of}\\
&\qquad (Just\ n, q') \to eval\ y\ n\\
&\qquad (Nothing, q') \to (Nothing, q')
\end{aligned}
$$

The State Type:

$$\textbf{type}\ State = Int$$

Top-Level Compilation:

$compile :: Expr \rightarrow Code$

Generalized Compilation:

$compile' :: Expr \rightarrow Code \rightarrow Code$

Target Language:

$\textbf{type}\ Code = [Op]$
$\textbf{data}\ Op$

Virtual Machine:

$exec :: Code \rightarrow Conf \rightarrow Conf$
$fail :: Conf \rightarrow Conf$

The Configuration Type:

$\textbf{type}\ Conf = (Stack,\ State)$

The Stack Type:

$\textbf{type}\ Stack = [Elem]$
$\textbf{data}\ Elem = VAL\ Int$

Specification for *compile*:

$$\begin{aligned} exec\ (compile\ e)\ (s,q) = \textbf{case}\ & eval\ e\ q\ \textbf{of} \\ & (Just\ n, q') \quad \rightarrow (VAL\ n : s, q') \\ & (Nothing, q') \rightarrow fail\ (s, q') \end{aligned}$$

Specification for *compile′*:

$$\begin{aligned} exec\ (compile'\ e\ c)\ (s,q) = \textbf{case}\ & eval\ e\ q\ \textbf{of} \\ & (Just\ n, q') \quad \rightarrow exec\ c\ (VAL\ n : s, q') \\ & (Nothing, q') \rightarrow fail\ (s, q') \end{aligned}$$

Calculations

Base case, $e = Val\ n$:

$$
\begin{aligned}
& \quad exec\ (compile'\ (Val\ n)\ c)\ (s,q) \\
= & \quad \{ \text{ specification for } compile' \ \} \\
& \quad \mathbf{case}\ eval\ (Val\ n)\ q\ \mathbf{of} \\
& \qquad (Just\ n, q') \quad \to exec\ c\ (VAL\ n : s, q') \\
& \qquad (Nothing, q') \to fail\ (s, q') \\
= & \quad \{ \text{ apply definition of } eval \ \} \\
& \quad exec\ c\ (VAL\ n : s, q) \\
= & \quad \{ \text{ define: } exec\ (PUSH\ n : c)\ (s,q) = exec\ c\ (VAL\ n : s, q) \ \} \\
& \quad exec\ (PUSH\ n : c)\ (s,q)
\end{aligned}
$$

Base case, $e = Throw$:

$$
\begin{aligned}
& \quad exec\ (compile'\ (Throw)\ c)\ (s,q) \\
= & \quad \{ \text{ specification for } compile' \ \} \\
& \quad \mathbf{case}\ eval\ (Throw)\ q\ \mathbf{of} \\
& \qquad (Just\ n, q') \quad \to exec\ c\ (VAL\ n : s, q') \\
& \qquad (Nothing, q') \to fail\ (s, q') \\
= & \quad \{ \text{ apply definition of } eval \ \} \\
& \quad fail\ (s,q) \\
= & \quad \{ \text{ define: } exec\ [FAIL]\ (s,q) = fail\ (s,q) \ \} \\
& \quad exec\ [FAIL]\ (s,q)
\end{aligned}
$$

Inductive case, $e = Add\ x\ y$:

$$
\begin{aligned}
& \quad exec\ (compile'\ (Add\ x\ y)\ c)\ (s,q) \\
= & \quad \{ \text{ specification for } compile' \ \} \\
& \quad \mathbf{case}\ eval\ (Add\ x\ y)\ q\ \mathbf{of} \\
& \qquad (Just\ n, q') \quad \to exec\ c\ (VAL\ n : s, q') \\
& \qquad (Nothing, q') \to fail\ (s, q') \\
= & \quad \{ \text{ apply definition of } eval \ \} \\
& \quad \mathbf{case}\ eval\ x\ q\ \mathbf{of} \\
& \qquad (Just\ n, q') \quad \to \mathbf{case}\ eval\ y\ q'\ \mathbf{of} \\
& \qquad\qquad\qquad\qquad (Just\ m, q'') \quad \to exec\ c\ (VAL\ (n+m) : s, q'') \\
& \qquad\qquad\qquad\qquad (Nothing, q'') \to fail\ (s, q'') \\
& \qquad (Nothing, q') \to fail\ (s, q') \\
= & \quad \{ \text{ define: } exec\ (ADD : c)\ (VAL\ m : VAL\ n : s, q) \\
& \qquad\qquad = exec\ c\ (VAL\ (n+m) : s, q) \ \} \\
& \quad \mathbf{case}\ eval\ x\ q\ \mathbf{of} \\
& \qquad (Just\ n, q') \quad \to \mathbf{case}\ eval\ y\ q'\ \mathbf{of}
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad (Just\ m, q'') \quad \rightarrow exec\ (ADD : c)\ (VAL\ m : VAL\ n : s, q'') \\
\qquad\qquad\qquad\qquad (Nothing, q'') \rightarrow fail\ (s, q'') \\
\quad (Nothing, q') \rightarrow fail\ (s, q') \\
= \quad \{ \text{ define: } fail\ (VAL\ n : s, q) = fail\ (s, q) \ \} \\
\textbf{case}\ eval\ x\ q\ \textbf{of} \\
\quad (Just\ n, q') \quad \rightarrow \textbf{case}\ eval\ y\ q'\ \textbf{of} \\
\qquad\qquad\qquad\qquad (Just\ m, q'') \quad \rightarrow exec\ (ADD : c)\ (VAL\ m : VAL\ n : s, q'') \\
\qquad\qquad\qquad\qquad (Nothing, q'') \rightarrow fail\ (VAL\ n : s, q'') \\
\quad (Nothing, q') \rightarrow fail\ (s, q') \\
= \quad \{ \text{ Induction hypothesis for } y \ \} \\
\textbf{case}\ eval\ x\ q\ \textbf{of} \\
\quad (Just\ n, q') \quad \rightarrow exec\ (compile'\ y\ (ADD : c))\ (VAL\ n : s, q') \\
\quad (Nothing, q') \rightarrow fail\ (s, q') \\
= \quad \{ \text{ Induction hypothesis for } x \ \} \\
exec\ (compile'\ x\ (compile'\ y\ (ADD : c)))\ (s, q)
\end{array}
$$

Inductive case, $e = Ite\ z\ x\ y$:

$$
\begin{array}{l}
exec\ (compile'\ (Ite\ z\ x\ y)\ c)\ (s, q) \\
= \quad \{ \text{ specification for } compile' \ \} \\
\textbf{case}\ eval\ (Ite\ z\ x\ y)\ q\ \textbf{of} \\
\quad (Just\ n, q') \quad \rightarrow exec\ c\ (VAL\ n : s, q') \\
\quad (Nothing, q') \rightarrow fail\ (s, q') \\
= \quad \{ \text{ apply definition of } eval \ \} \\
\textbf{case}\ eval\ z\ q\ \textbf{of} \\
\quad (Just\ n, q') \quad \rightarrow \textbf{if}\ n == 0 \\
\qquad\qquad\qquad\qquad \textbf{then case}\ eval\ y\ q'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad (Just\ m, q'') \quad \rightarrow exec\ c\ (VAL\ m : s, q'') \\
\qquad\qquad\qquad\qquad\qquad (Nothing, q'') \rightarrow fail\ (s, q'') \\
\qquad\qquad\qquad\qquad \textbf{else case}\ eval\ x\ q'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad (Just\ m, q'') \quad \rightarrow exec\ c\ (VAL\ m : s, q'') \\
\qquad\qquad\qquad\qquad\qquad (Nothing, q'') \rightarrow fail\ (s, q'') \\
\quad (Nothing, q') \rightarrow fail\ (s, q') \\
= \quad \{ \text{ induction hypothesis for } x \ \} \\
\textbf{case}\ eval\ z\ q\ \textbf{of} \\
\quad (Just\ n, q') \quad \rightarrow \textbf{if}\ n == 0 \\
\qquad\qquad\qquad\qquad \textbf{then case}\ eval\ y\ q'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad (Just\ m, q'') \quad \rightarrow exec\ c\ (VAL\ m : s, q'') \\
\qquad\qquad\qquad\qquad\qquad (Nothing, q'') \rightarrow fail\ (s, q'') \\
\qquad\qquad\qquad\qquad \textbf{else}\ exec\ (compile'\ x\ c)\ (s, q')
\end{array}
$$

$(Nothing, q') \rightarrow fail\ (s, q')$

$=\quad$ { induction hypothesis for y }

$\textbf{case}\ eval\ z\ q\ \textbf{of}$

$(Just\ n, q') \quad \rightarrow \textbf{if}\ n == 0$

$\textbf{else}\ exec\ (compile'\ y\ c)\ (s, q')$

$\textbf{else}\ exec\ (compile'\ x\ c)\ (s, q')$

$(Nothing, q') \rightarrow fail\ (s, q')$

$=\quad$ { define: $exec\ (JUMP\ c'' : c)\ (VAL\ n : s, q) = \textbf{if}\ n == 0$

$\textbf{then}\ exec\ c\ (s, q)$

$\textbf{else}\ exec\ c''\ (s, q)$ }

$\textbf{case}\ eval\ z\ q\ \textbf{of}$

$(Just\ n, q') \rightarrow exec\ (JUMP\ (compile'\ x\ c) : (compile'\ y\ c))\ (VAL\ n : s, q')$

$(Nothing, q') \rightarrow fail\ (s, q')$

$=\quad$ { induction hypothesis for z }

$exec\ (compile'\ z\ (JUMP\ (compile'\ x\ c) : (compile'\ y\ c)))\ (s, q)$

Inductive case, $e = Catch\ x\ h$:

$exec\ (compile'\ (Catch\ x\ h)\ c)\ (s, q)$

$=\quad$ { specification for $compile'$ }

$\textbf{case}\ eval\ (Catch\ x\ h)\ q\ \textbf{of}$

$(Just\ n, q') \quad \rightarrow exec\ c\ (VAL\ n : s, q')$

$(Nothing, q') \rightarrow fail\ (s, q')$

$=\quad$ { apply definition of $eval$ }

$\textbf{case}\ eval\ x\ q\ \textbf{of}$

$(Just\ n, q') \quad \rightarrow exec\ c\ (VAL\ n : s, q')$

$(Nothing, q') \rightarrow \textbf{case}\ eval\ h\ q'\ \textbf{of}$

$(Just\ m, q'') \quad \rightarrow exec\ c\ (VAL\ m : s, q'')$

$(Nothing, q'') \rightarrow fail\ (s, q'')$

$=\quad$ { induction hypothesis for h }

$\textbf{case}\ eval\ x\ q\ \textbf{of}$

$(Just\ n, q') \quad \rightarrow exec\ c\ (VAL\ n : s, q')$

$(Nothing, q') \rightarrow exec\ (compile'\ h\ c)\ (s, q')$

$=\quad$ { define: $fail\ (HAN\ c' : s, q) = exec\ c'\ (s, q)$ }

$\textbf{case}\ eval\ x\ q\ \textbf{of}$

$(Just\ n, q') \quad \rightarrow exec\ c\ (VAL\ n : s, q')$

$(Nothing, q') \rightarrow fail\ (HAN\ (compile'\ h\ c) : s, q')$

$=\quad$ { define: $exec\ (UNMARK : c)\ (VAL\ n : HAN\ _ : s, q)$

$= exec\ c\ (VAL\ n : s, q)$ }

$$
\begin{array}{ll}
& \textbf{case } \mathit{eval}\ x\ q\ \textbf{of} \\
& \quad (\mathit{Just}\ n, q') \to \mathit{exec}\ (\mathit{UNMARK} : c)\ (\mathit{VAL}\ n : \mathit{HAN}\ (\mathit{compile}'\ h\ c) : s, q') \\
& \quad (\mathit{Nothing}, q') \to \mathit{fail}\ (\mathit{HAN}\ (\mathit{compile}'\ h\ c) : s, q') \\
= & \quad \{ \text{ induction hypothesis for } x \ \} \\
& \mathit{exec}\ (\mathit{compile}'\ x\ (\mathit{UNMARK} : c))\ (\mathit{HAN}\ (\mathit{compile}'\ h\ c) : s, q) \\
= & \quad \{ \text{ define: } \mathit{exec}\ (\mathit{MARK}\ c' : c)\ (s, q) = \mathit{exec}\ c\ (\mathit{HAN}\ c' : s, q) \ \} \\
& \mathit{exec}\ (\mathit{MARK}\ (\mathit{compile}'\ h\ c) : (\mathit{compile}'\ x\ (\mathit{UNMARK} : c)))\ (s, q)
\end{array}
$$

Base case, $e = \mathit{Get}$:

$$
\begin{array}{ll}
& \mathit{exec}\ (\mathit{compile}'\ (\mathit{Get})\ c)\ (s, q) \\
= & \quad \{ \text{ specification for } \mathit{compile}' \ \} \\
& \textbf{case } \mathit{eval}\ (\mathit{Get})\ q\ \textbf{of} \\
& \quad (\mathit{Just}\ n, q') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s, q') \\
& \quad (\mathit{Nothing}, q') \to \mathit{fail}\ (s, q') \\
= & \quad \{ \text{ apply definition of } \mathit{eval} \ \} \\
& \mathit{exec}\ c\ (\mathit{VAL}\ q : s, q) \\
= & \quad \{ \text{ define: } \mathit{exec}\ (\mathit{LOAD} : c)\ (s, q) = \mathit{exec}\ c\ (\mathit{VAL}\ q : s, q) \ \} \\
& \mathit{exec}\ (\mathit{LOAD} : c)\ (s, q)
\end{array}
$$

Inductive case, $e = \mathit{Put}\ x\ y$:

$$
\begin{array}{ll}
& \mathit{exec}\ (\mathit{compile}'\ (\mathit{Put}\ x\ y)\ c)\ (s, q) \\
= & \quad \{ \text{ specification for } \mathit{compile}' \ \} \\
& \textbf{case } \mathit{eval}\ (\mathit{Put}\ x\ y)\ q\ \textbf{of} \\
& \quad (\mathit{Just}\ n, q') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s, q') \\
& \quad (\mathit{Nothing}, q') \to \mathit{fail}\ (s, q') \\
= & \quad \{ \text{ apply definition of } \mathit{eval} \ \} \\
& \textbf{case } \mathit{eval}\ x\ q\ \textbf{of} \\
& \quad (\mathit{Just}\ n, q') \quad \to \textbf{case } \mathit{eval}\ y\ n\ \textbf{of} \\
& \qquad\qquad\qquad\qquad (\mathit{Just}\ m, q'') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ m : s, q'') \\
& \qquad\qquad\qquad\qquad (\mathit{Nothing}, q'') \to \mathit{fail}\ (s, q'') \\
& \quad (\mathit{Nothing}, q') \to \mathit{fail}\ (s, q') \\
= & \quad \{ \text{ induction hypothesis for } y \ \} \\
& \textbf{case } \mathit{eval}\ x\ q\ \textbf{of} \\
& \quad (\mathit{Just}\ n, q') \quad \to \mathit{exec}\ (\mathit{compile}'\ y\ c)\ (s, n) \\
& \quad (\mathit{Nothing}, q') \to \mathit{fail}\ (s, q') \\
= & \quad \{ \text{ define: } \mathit{exec}\ (\mathit{SAVE} : c)\ (\mathit{VAL}\ n : s, q') = \mathit{exec}\ c\ (s, n) \ \} \\
& \textbf{case } \mathit{eval}\ x\ q\ \textbf{of}
\end{array}
$$

$$
\begin{aligned}
&\quad (Just\ n, q') \quad \rightarrow exec\ (SAVE : (compile'\ y\ c))\ (VAL\ n : s, q') \\
&\quad (Nothing, q') \rightarrow fail\ (s, q') \\
=&\quad \{ \text{ induction hypothesis for } x \ \} \\
&exec\ (compile'\ x\ (SAVE : (compile'\ y\ c)))\ (s, q)
\end{aligned}
$$

Top-level compilation function:

$$
\begin{aligned}
&exec\ (compile\ e)\ (s, q) \\
=&\quad \{ \text{ specification for } compile \ \} \\
&\textbf{case}\ eval\ e\ q\ \textbf{of} \\
&\quad (Just\ n, q') \quad \rightarrow (VAL\ n : s, q') \\
&\quad (Nothing, q') \rightarrow fail\ (s, q') \\
=&\quad \{ \text{ define: } exec\ [HALT]\ (s, q) = (s, q) \ \} \\
&\textbf{case}\ eval\ e\ q\ \textbf{of} \\
&\quad (Just\ n, q') \quad \rightarrow exec\ [HALT]\ (VAL\ n : s, q') \\
&\quad (Nothing, q') \rightarrow fail\ (s, q') \\
=&\quad \{ \text{ specification for } compile' \ \} \\
&exec\ (compile'\ e\ [HALT])\ (s, q)
\end{aligned}
$$

Derived Definitions

Target Language:

```
type Code = [Op]

data Op   = PUSH Int
          | FAIL
          | ADD
          | JUMP Code
          | MARK Code
          | UNMARK
          | LOAD
          | SAVE
          | HALT
```

Compiler:

```
compile                :: Expr → Code
compile e              = compile′ e [HALT]

compile′               :: Expr → Code → Code
compile′ (Val n) c     = PUSH n : c
compile′ (Throw) c     = [FAIL]
```

$$
\begin{aligned}
&compile'\ (Add\ x\ y)\ c &&= compile'\ x\ (compile'\ y\ (ADD : c))\\
&compile'\ (Ite\ z\ x\ y)\ c &&= compile'\ z\ (JUMP\ (compile'\ x\ c) : (compile'\ y\ c))\\
&compile'\ (Catch\ x\ h)\ c &&= MARK\ (compile'\ h\ c) : (compile'\ x\ (UNMARK : c))\\
&compile'\ (Get)\ c &&= LOAD : c\\
&compile'\ (Put\ x\ y)\ c &&= compile'\ x\ (SAVE : (compile'\ y\ c))
\end{aligned}
$$

Virtual Machine:

$$
\begin{aligned}
&\textbf{type}\ Stack = [Elem]\\
&\textbf{data}\ Elem = VAL\ Int \mid HAN\ Code
\end{aligned}
$$

$$
\begin{aligned}
&exec &&:: Code \rightarrow Conf \rightarrow Conf\\
&exec\ [HALT]\ (s,q) &&= (s,q)\\
&exec\ (PUSH\ n : c)\ (s,q) &&= exec\ c\ (VAL\ n : s,q)\\
&exec\ [FAIL]\ (s,q) &&= fail\ (s,q)\\
&exec\ (ADD : c)\ (VAL\ m : VAL\ n : s,q) &&= exec\ c\ (VAL\ (n+m) : s,q)\\
&exec\ (JUMP\ c'' : c)\ (VAL\ n : s,q) &&= \textbf{if}\ n == 0\\
& &&\quad \textbf{then}\ exec\ c\ (s,q)\\
& &&\quad \textbf{else}\ exec\ c''\ (s,q)\\
&exec\ (UNMARK : c)\ (VAL\ n : HAN\ _ : s,q) &&= exec\ c\ (VAL\ n : s,q)\\
&exec\ (MARK\ c' : c)\ (s,q) &&= exec\ c\ (HAN\ c' : s,q)\\
&exec\ (LOAD : c)\ (s,q) &&= exec\ c\ (VAL\ q : s,q)\\
&exec\ (SAVE : c)\ (VAL\ n : s,q') &&= exec\ c\ (s,n)
\end{aligned}
$$

$$
\begin{aligned}
&fail &&:: Conf \rightarrow Conf\\
&fail\ (VAL\ n : s,q) &&= fail\ (s,q)\\
&fail\ (HAN\ c' : s,q) &&= exec\ c'\ (s,q)
\end{aligned}
$$

Appendix C - *Lexical Program Structure*

In this appendix, the complete lexical program structure of CoCa is presented. This structure outlines how valid tokens are formed, including identifiers, literals, operators, and other lexical elements.

Program → { *Lexeme* | *WhiteSpace* }
Lexeme → *ReservedOp* | *Identifier* | *Literal* | *Special* | *NewLines*

NewLines → *NewLine* { *NewLine* }
NewLine → *Return Linefeed* | *Return* | *Linefeed* | *Formfeed*
Return → `a carriage return`
Linefeed → `a line feed`
Formfeed → `a form feed`

WhiteSpace → *WhiteElement* { *WhiteElement* }
WhiteElement → *IndentCode* | *WhiteChar* | *Comment*
IndentCode → *NewLine WhiteChar* { *WhiteChar* }
WhiteChar → *Space* | *Tab* | *VerTab*
Space → `a space`
Tab → `a horizontal tab`
VerTab → `a vertical tab`

Comment → [*NewLines*] *Dashes* { *AnyLine* }
  | [*NewLines*] *OpenComment* { *Any* } *CloseComment*
Dashes → `--` { `-` }
OpenComment → `{-`
CloseComment → `-}`
Any → *AnyLine* | *NewLines* | *VerTab*
AnyLine → *Graphic* | *Space* | *Tab*
Graphic → *Letter* | *Digit* | *Symbol* | *Special* | *Prime*

Symbol → `!` | `"` | `#` | `$` | `%` | `&` | `*` | `+` | `-` | `/` | `|`
  | `:` | `<` | `>` | `?` | `@` | `\` | `^` | `‘` | `~` | `´` | `;` | `_`

Special → `.` | `,` | `[` | `]` | `(` | `)` | `{` | `}` | `=`

ReservedOp → *ArithmeticOp* | *EqualityOp* | *ComparisonOp*
  | *LogicalOp* | *ListOp* | *MiscOp*

ArithmeticOp → + | - | * | / | ^ | ^^ | **

EqualityOp → == | /= | !=

ComparisonOp → < | <= | > | >=

LogicalOp → && | || | ~

ListOp → : | ++

MiscOp → ==> | <== | :: | -> | |

Identifier → *ReservedId* | *BoundVarId* | *VarId*
  | *IConId* | *ConId* | *FunId*

ReservedId → induction | forall | QED | data | case | of
  | type | Let | ref | simplify | not

BoundVarId → $ (*VarId* | *IConId* | *ConId* | *FunId*)

VarId → *Letter* { *Prime* }

IConId → (*Large Large* { *Large* })$_{(\mathit{ReservedId})}$

ConId → (*Large* { *Letter* } *Small* { *Letter* })$_{(\mathit{ReservedId})}$

FunId → (*Small FunIdElem* { *FunIdElem* | *Prime* })$_{(\mathit{ReservedId})}$

FunIdElem → *Letter* | *Digit* | _ | - | ?

Literal → *Integer* | *Float*

Integer → *Decimal*

Float → *Decimal* . *Decimal* [*Exponent*]
  | *Decimal Exponent*

Decimal → *Digit* { *Digit* }

Exponent → (e | E) [+ | -] *Decimal*

Letter → *Small* | *Large*

Small → a | b | c | d | e | f | g | h | i | j | k | l | m
  | n | o | p | q | r | s | t | u | v | x | y | z

Large → A | B | C | D | E | F | G | H | I | J | K | L | M
  | N | O | P | Q | R | S | T | U | V | X | Y | Z

Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

$Prime \rightarrow$ '

Appendix D - *Context-Free Grammar*

In this appendix, the complete context-free grammar defining the source language for CoCa is presented. This grammar specifies the rules that govern how various syntactic elements of the language, such as expressions, declarations, and statements, are structured and combined.

Root	→	*Input* | *File*	
Input	→	*Statement*	
File	→	[*NewLines*] *FileStructure*	
FileStructure	→	*TopSpecs CorEq* [(*NewLines* | *OptProof*)]	
OptProof	→	*NewLines Proof* [*NewLines*]	
TopSpecs	→	$TopSpec_1$ ... $TopSpec_n$	(n >= 1)
TopSpec	→	*TopDecl NewLines*	
	|	*TypeSig NewLines*	
	|	*FunDef NewLines*	
TopDecl	→	`type` *SimpleType* = *FunType*	(type decl)
	|	`data` *SimpleType* [= *Constrs*]	(data decl)
TypeSig	→	*TyFun* `::` *FunType*	(signature)
TyFun	→	*FunId*	(fun name)
SimpleType	→	*TyCon* [*TyVars*]	
TyCon	→	*ConId*	(constructor)
TyVars	→	$TyVar_1$ ... $TyVar_n$	(n >= 1)
TyVar	→	*VarId*	(variable)
Constrs	→	$Constr_1$ `|` ... `|` $Constr_n$	(n >= 1)
Constr	→	*Con* [*AppType*]	
Con	→	*ConId*	(constructor)
	|	*IConId*	(introduced con)

FunType	→	*AppType* [`->` *FunType*]	(fun type)
AppType	→	[*AppType*] *AtomType*	(app type)
AtomType	→	*TyCon*	
	|	*TyVar*	
	|	`(` *FunType* `)`	(parenthesized)
	|	`[` $FunType_1$ `,` ... `,` $FunType_n$ `]`	(list type, n>=0)
	|	`(` $FunType_1$ `,` ... `,` $FunType_n$ `)`	(tuple type n>=2)
FunDef	→	*Equation* [`{` `ref` *Ref* `}`]	(fun def)
Equation	→	*Exp* `=` *Exp*	
Ref	→	*Exp*	
Exp	→	`-` *Exp*	(prefix neg)
	|	*LogicNeg* *Exp*	(logical neg)
	|	`case` *Exp* `of` `|` *Alt* { `|` *Alt* }	(case exp)
	|	*Exp* *BinOp* *Exp*	(binary exp)
	|	*Exp* *LogicOp* *Exp*	(logic exp)
	|	*FApp*	
Alt	→	*Exp* `->` *Exp*	
FApp	→	[*FApp*] *FAtom*	(fun app)
FAtom	→	`(` *Exp* `)`	(parenthesized)
	|	`[` Exp_1 `,` ... `,` Exp_n `]`	(list, n>=0)
	|	`(` Exp_1 `,` ... `,` Exp_n `)`	(tuple, n>=2)
	|	*EAtom*	(exp atom)
EAtom	→	*EPower*	
	|	*VarId*	(variable)
	|	*BoundVarId*	(bound variable)
	|	*IConId*	(introduced con)
	|	*ConId*	(constructor)
	|	*FunId*	(fun name)
	|	*Integer*	(integer)
	|	*Float*	(floating point)

```
EPower       →  Integer Raised Integer                          (integer)
             |  Float Raised ( Integer | Float )                (floating point)
Raised       →  ^ | ^^ | **

BinOp        →  + | - | * | / | : | ++ | ==
             |  /= | != | < | <= | > | >=
LogicNeg     →  not | ∼
LogicOp      →  && | ||

CorEq        →  forall Vars . Equation                          (corr equation)
Vars         →  Var_1 ... Var_n                                 (n >= 1)
Var          →  VarId                                           (variable)

Proof        →  ProofCases ProofEnd
ProofCases   →  ProofCase_1 ... ProofCase_n                     (n >= 1)
ProofCase    →  Case NewLines Calculations

Case         →  Let Var = Constr

Calculations →  Calculation NewLines [ CComment ]
CComment     →  Comment NewLines Calculations
Calculation  →  Exp

Comment      →  = { Statement }
Statement    →  <== ( Equation | Ref )                          (apply R to L)
             |  ==> ( Equation | Ref )                          (apply L to R)
             |  induction Equation                              (preset induction)
             |  <== induction Equation                          (induction R to L)
             |  ==> induction Equation                          (induction L to R)
             |  simplify                                        (simplification)

ProofEnd     →  QED
```