<table>
<thead>
<tr>
<th>Title</th>
<th>Verifying the Correctness of Compiler for an Imperative Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>Trinh, Bao Ngoc Quoc</td>
</tr>
<tr>
<td>Citation</td>
<td></td>
</tr>
<tr>
<td>Issue Date</td>
<td>2011-03</td>
</tr>
<tr>
<td>Type</td>
<td>Thesis or Dissertation</td>
</tr>
<tr>
<td>Text version</td>
<td></td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hdl.handle.net/10119/9639">http://hdl.handle.net/10119/9639</a></td>
</tr>
<tr>
<td>Rights</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Supervisor: Professor Futatsugi Kokichi, School of Information, Master Course</td>
</tr>
<tr>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td></td>
</tr>
</tbody>
</table>
Verifying the Correctness of Compiler for an Imperative Programming Language

By Trinh Bao Ngoc Quoc

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Kokichi Futatsugi

March, 2011
Verifying the Correctness of Compiler for an Imperative Programming Language

By Trinh Bao Ngoc Quoc (0910039)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Kokichi Futatsugi

and approved by
Professor Kokichi Futatsugi
Associate Professor Kazuhiro Ogata
Associate Professor Toshiaki Aoki

February, 2011 (Submitted)
Abstract

Algebraic denotational semantics (ADS) is our approach to describe the semantics of an imperative programming language in the order sorted equational logic. Using CafeOBJ, an algebraic specification language, the syntax of an imperative language, Minila, is specified into modules of sorts, operators and rewriting rules, and its algebraic semantics are specified by the Environment. The compiler of Minila language is also specified into a module whose rewriting rules describe the transformation from Minila programs into sequences of machine instructions. In addition, the semantics of machine language is considered to guarantee the semantics preservation of the Compiler. The main part of our research is verifying the correctness of compiler for Minila language using CafeOBJ proof assistant.

Keywords: ADS, CafeOBJ, Minila, compiler, semantic preservation, correctness, Environment, verification, proof score
Acknowledgement

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. In the first place, I would like to express my deep sense of gratitude to my supervisor, Professor Kokichi Futatsugi, for his supervision, advice, assistance and guidance during the whole period of my Master’s course. He has provided me kind encouragements and supports not only in my research but also in my life.

I have furthermore to thank Assistant Professor Kazuhiro Ogata for his advice, support and reviews. During my research, Mr Ogata had worked with me as the second supervisor. I had learned very much from his research knowledge, problem solving and working methodology.

I would like to thank Assistant Professor Toshiaki Aoki, Associate Professor Yuki Chiba, and all members of Futatsugi and Aoki Laboratory in JAIST for their valuable comments, hints that helped me enhance the quality of my research.

I also would like to thank TIS company who provided me the JAIViT scholarship for this Master’s course.

At last, I would like to give my special thanks to my family members, my friends, especially my parents whose encouragements and sharing enabled me to complete this research.
# Contents

1 Introduction .................................................. 5
   1.1 Background ............................................. 5
   1.1.1 Imperative Programming Language ................. 5
   1.1.2 Algebraic Denotational Semantics ................. 6
   1.1.3 CafeOBJ .............................................. 6
   1.1.4 Compiler .............................................. 8
   1.2 Minila .................................................. 9
   1.2.1 Definition ........................................... 9
   1.2.2 The correctness of Minila’s Compiler ............. 11
   1.3 Motivation ............................................... 12

2 Specification ................................................ 13
   2.1 Previous Works ......................................... 13
   2.2 Specifications ......................................... 13
      2.2.1 Minila language .................................. 14
      2.2.2 Environment ....................................... 18
      2.2.3 Machine language ................................. 20
      2.2.4 Compiler .......................................... 22
      2.2.5 Sort Hierarchy .................................... 24
   2.3 Environment Hierachy ................................. 25
   2.4 Error Handling ......................................... 26
      2.4.1 Interpreter ....................................... 27
      2.4.2 Virtual Machine .................................. 30
      2.4.3 3-steps Detection of Error Handling ............. 31

3 Compiler Verification ....................................... 32
   3.1 Correctness of Minila’s Compiler .................... 32
      3.1.1 Previous Formalization ......................... 32
      3.1.2 Formalization for Correctness of The Compiler .... 33
   3.2 Verification by Using CafeOBJ ....................... 34
      3.2.1 Proof Scores ..................................... 34
      3.2.2 Case Splitting ................................... 35
   3.3 Correctness’s Verification ............................ 37
      3.3.1 Theorem th ...................................... 37
3.3.2 Finding Lemmas ............................................. 39
3.3.3 More Specifications ........................................... 42
3.3.4 Theorem \texttt{th4} ............................................. 46
3.3.5 The source code ................................................ 50

4 Conclusion .......................................................... 51
4.1 Summary ........................................................... 51
  4.1.1 Completed Specifications ..................................... 51
  4.1.2 Completed Verifications ..................................... 52
4.2 Related Works ..................................................... 52
4.3 Future Research ................................................... 53
List of Figures

1.1 The Morris Correctness Diagram ........................................ 9
1.2 The Correctness of Minila’s Compiler Diagram .......................... 11

2.1 Sort hierarchy for the Expression ....................................... 25
2.2 The Environment Hierarchy ............................................... 25
2.3 The alternative sort hierarchy for Environment ....................... 27

3.1 Theorem Relation Diagram .............................................. 50
Chapter 1

Introduction

1.1 Background

1.1.1 Imperative Programming Language

“In computer science, imperative programming is a programming paradigm that describes computation in terms of statements that change a program state. In much the same way that imperative mood in natural languages expresses commands to take action, imperative programs define sequences of commands for the computer to perform”\(^1\). This is one definition about the imperative programming language. Nowadays, imperative programming languages are utilized widely in the computer industry for calculations with many examples such as FORTRAN, ALGOL, COBOL, Pascal, C, BASIC, Ada and many more.

A program of imperative language normally is a sequence of order statements, in which each statement is computed in turn. A storage contains the state before a program executed is an essential feature of an imperative language. The values, results of the execution, are usually put into this storage. The storage is an abstract entity which associates values with the variables of imperative language. A initial state of programs could be described by variables and the values. After executed, the results could be accessed from these variables. An imperative language certainly supports some kinds of statements including assignment, sequential, conditional, iterative, procedure calls etc.

There are many approaches to describe the semantics of imperative languages which can be divided into three groups: operation, axiom and denotation. An operational semantics describes the meaning of an imperative language by describing a way of executing its programs. This could be done by giving an interpreter or a compiler for the language. In axiomatic approaches, programming language features are defined by writing axioms in some logical system. First order logic is the most popular, since it is the logical system most widely used in mathematics and its foundations. And the last one, denotational approaches build models of language features, these models are called denotations. The denotation of an imperative program is constructed by composing the denotations of sub components.

\(^1\)http://en.wikipedia.org/wiki/Imperative_programming_language
1.1.2 Algebraic Denotational Semantics

Our approach might be called algebraic denotational semantics [9] which is abbreviated as ADS. This approach combines aspects of denotational, axiomatic and operational semantics. The denotational aspect arises because everything specified has a denotation in an algebra; the axiomatic aspect arises from the fact that we specify these algebras using equations; and the operational aspect arises from the fact that we symbolically execute programs using the term rewriting facility of OBJ, a famous algebraic specification language.

If an imperative programming language is specified as a model which is denoted by a *Program* algebra and the storage of this language is denoted by a *Store* algebra, then the semantics of this language which described in ADS will be formalized as follows:

**Formalization 1.1.1**

\[ \_ ; \_ : \text{Store} \rightarrow \text{Program} \rightarrow \text{Store} \]

The ";" notation separates the syntax of operation which declared by "\_ ; \_" and the rest of formulas. In the form of the "\_ ; \_" operation, the underbars are place holders that indicate where input entities of algebra *Store* and *Program* go, respectively. The "\rightarrow" notation indicates that entities on the left are the input algebras and one entity on the right (*Store*) is the output algebra.

Intuitively, when an imperative programming language is considered with respect in a storage, the behaviours of this program will effect on the state of the storage. In the ADS context, the formalization 1.1.1 shows that after the *Program* is considered with respect to the left *Store*, semantics of this *Program* is specified by the right *Store*.

In this document, we will consider the CafeOBJ specification language in order to specify the algebras semantics of an imperative programming language and a storage. Moreover, it could be used to verify properties of this language.

1.1.3 CafeOBJ

CafeOBJ [3] is a most advanced formal specification language which is an implementation of OBJ algebraic specification language. It inherits many advanced features from OBJ such as flexible mix-fix syntax, powerful and clear typing sytem with ordered sorts, parameterised modules and views for instantiating the parameters and module expressions. CafeOBJ is a language for writing formal specifications of models for wide varieties of software and systems. Because of an executable language, properties of systems could be verified by writing and executing *proof scores* in CafeOBJ.

In order to specify imperative language, we introduce to some basic denotations in CafeOBJ language. Now, we begin with the *signature*, a collection of *sorts* and *operation’s* declarations.

**Example 1.1.3.1** Let’s consider the signature of the data type of natural number which is defined by Peone.

\[
\text{mod! BASIC-NAT} \{
\]
In this example, \texttt{Zero} and \texttt{NzNat} are sorts which denote zero number and non zero number, respectively. Intuitively, zero or non zero numbers are all considered as general natural numbers, therefore \texttt{Zero} and \texttt{NzNat} are subsorts of the \texttt{Nat}, general natural numbers, and the \texttt{subsort} relation is denoted by the "<" symbol. This is the \textbf{order sorted signatures} feature of CafeOBJ language.

The sucessor of a natural number is a non zero number which is declared

\begin{verbatim}
  op s : Nat -> NzNat \{constr\}
\end{verbatim}

where CafeOBJ keyword "op" indicates that the syntax of an operation is being declared, with the output sort after the "\rightarrow" and the input sorts listed between the ":" and the "\rightarrow". The \texttt{s} is called a pre-fix operator (function) of CafeOBJ. There are other structures such as the in-fix, post-fix and mix-fix syntax. In this document, the in-fix and mix-fix operators are mainly used, an example is showed as follows,

\begin{verbatim}
  op _=_ : Nat Nat -> Bool \{comm\}
  op if\_then\_else\_fi : Exp Stm Stm -> Stm
\end{verbatim}

We will explain in detail these operators in next sections.

\textit{Modules} are the basic building blocks of CafeOBJ. Roughly, a module is a CafeOBJ structure which contains \textit{sorts}, operation's declarations (\textit{operators}) and \textit{re-writing rules}. Modules usually specify models, the denotations of algebras in CafeOBJ lanuguage. CafeOBJ also supports the \textit{parameterized modules} feature, a module might have many parameters, and the \textit{importation} feature, one module could import other modules.

\textbf{Example 1.1.3.2} A basic natural number only has data sorts and comparison operations. It’s specified by a module as follows,

\begin{verbatim}
mod! BASIC-NAT {
  [Zero NzNat < Nat]
  op 0 : -> Zero \{constr\}
  op s : Nat -> NzNat \{constr\}
  --
  op _=_ : Nat Nat -> Bool \{comm\}
  op _<_ : Nat Nat -> Bool
  -- _=_
    eq (0 = s(N:Nat)) = false .
    eq (N:Nat = N) = true .
    eq (s(M:Nat) = s(N:Nat)) = (M = N) .
  -- _<_
    eq (N:Nat < N) = false .
}
\end{verbatim}
\[
\begin{align*}
\text{eq } 0 < s(N:\text{Nat}) &= \text{true }. \\
\text{eq } s(M:\text{Nat}) < 0 &= \text{false }. \\
\text{eq } s(M:\text{Nat}) < s(N:\text{Nat}) &= M < N .
\end{align*}
\]

A operation declared by op sentences is defined by some re-writing rules. Re-writing rules, the sentences beginning with a "eq" keyword specify the semantic or meaning of operations. In above example, three re-writing rules,

\[
\begin{align*}
\text{eq } (0 = s(N:\text{Nat})) &= \text{false }. \\
\text{eq } (N:\text{Nat} = N) &= \text{true }. \\
\text{eq } (s(M:\text{Nat}) = s(N:\text{Nat})) &= (M = N) .
\end{align*}
\]

specify the semantics of equations between two numbers.

Using CafeOBJ, an imperative language is constructed from many components, each component is an algebra which denoted by a model. These models will be specified into modules. However, we will show in detail specifications of each module in next chapter. Now, we introduce one important component of an imperative language which is called "compiler".

1.1.4 Compiler

There are gaps between the natural language of humanity and the language for computers (the computer code). Therefore, it's very difficult for people to write and understand programs using computer codes. In order to make the programming more simple, programming language should be built close to the human language (high level language). After that, written programs will be transform into the languages that computer could understand (low level language). The transformation may occur in many steps from high level to low level languages. Each step of transformation is done by a program called "Compiler". Because the intention of programmer is described by programming languages but the execution is done by computer languages, there might have a difference when transforming. As a result, Compilers must sure that the intention is preserved at computer language. This is a big challenge for building a Compiler of programming language.

A compiler of an imperative programming language is a program which transforms programs writing in the imperative language (the source language) into another computer language (the target language). One significant property of a compiler is that the target language preserves the semantics of the source language and it's called the correctness. Nowadays, almost all imperative languages have own compilers, therefore verifications of the compiler's correctness become very important. Many approaches were proposed and we now consider an approach which is first introduced by Morris [1].

In this diagram, the nodes are algebras and the arrows are homomorphisms. Not only the source, target languages but also the source, target semantics are specified into algebra modules. The compiler which translates the source language into target language is defined by equations which are specified by re-writing rules in CafeOBJ. The notation $\psi$
and $\phi$ used to specify the algebraic semantics of the source and target language, respectively are also defined by equations that specified by re-writing rules. The decoding arrow means that the semantic of a target program will represent in some adequate way the semantic of the corresponding source program (and not require it to be exactly the same), and this is a definition about the correctness of a compiler for an imperative language.

In the Morris diagram, the source and target are theoretical languages that might accept a program containing unlimited statements. However, in practice, the number statements of a giving source program is finite. Moreover, a Compiler must sure that all source programs that be transformed into target programs will terminate with limited time. Compiler could be seen as a total function in mathematics. In contrast, the $\psi$ and $\phi$ are partial functions, they might not return the value (not be terminated) depending on the condition of source programms and target programs. The non-termination of $\psi$ and $\phi$ execution will be considered carefully in this research.

In the next section, we define an imperative programming language which is called Minila. And in detail, we will consider the correctness diagram of Minila’s Compiler which based on the idea of the Morris’s diagram.

1.2 Minila

1.2.1 Definition

Minila (Mini-language) is an imperative programming language. It’s quite simple but having many essential features of an imperative language. In this document, Minila language is considered as the "source" language in the Morris’s diagram (Figure 1.1)

Minila has a storage which contains variables and corresponding values, the data value is only the natural number. CafeOBJ’s specification of the storage is called Environment, which is a list of pairs of a variable and a natural number. The objects of natural number, variable and Environment are specified into BASIC-NAT, VAR and ENV modules, respectly. Minila also has expressions which are constructed from natural numbers, variables or combinations of two expressions. The combinations of expressions are addition (denoted by ++ symbol), subtraction (−−), multiplication (**), division (/ /), modulo (%%) and comparations including equal (==), not equal (! ==), less than (< <), greater than (>>), and (&&), and or (||). The expression object is specified into the EXP module.

Minila language supports some statement structures of an imperative programming
language including empty statement, assignment, sequential, conditional and iterative statements. A Minila program is a list of statements which has form such as \( S_1 S_2 \cdots S_n \). Statement \( S_i \) could be one of structures as follows:

+ \textit{estm} \quad \text{(empty statement)}
+ \textit{v(i) := E ;} \quad \text{(assignment statement)}
+ \textit{if E then } S_a \text{ else } S_b \text{ fi} \quad \text{(conditional statement)}
+ \textit{while E do S od} \quad \text{(iterative statement)}
+ \textit{for v(i) from E_a to E_b do S od} \quad \text{(iterative statement)}
+ \textit{repeat S until E} \quad \text{(iterative statement)}

In above structures, \( v(i) \) is a variable with the index \( i \), \( E, E_a, E_b \) are expressions and \( S, S_a, S_b \) are statements. The statement object of Minila is specified into a STM module. The empty statement of Minila does nothing, it means that the semantics of programs will not be changed when this statement is executed.

This is an example of a Minila program, which is written in CafeOBJ language:

\[
\begin{align*}
v(0) &:= 0 ; \quad v(s(0)) := s(0) ; \quad \text{while } v(0) \ll s(s(s(0))) \text{ do} \\
& \quad v(0) := v(0) ++ s(0) ; \\
& \quad v(s(0)) := v(s(0)) ** v(0) ; \\
& \quad \text{od .}
\end{align*}
\]

The program is created from three statements, two assignment and one while statements. The while statement contains two assignment statements inside. When this program is executed, one component of Minila language, called Interpreter, directly describles the semantics of each statement of this program into Environment. The Environment containing semantics of this program is as follows,

\[
((v(0) \ll s(s(s(s(s(s(0))))))) >) | ((v(0) \ll s(s(0)))) >) | \\
((v(s(0)) \ll s(s(0)))) >) | ((v(0) \ll s(s(0)))) >) | \\
((v(s(0)) \ll s(0)) >) | ((v(0) \ll s(0)) >) | \\
((v(s(0)) \ll s(0)) >) | ((v(0) \ll 0) >) | \text{empty})))))
\]

A Machine language that defined as sequences of machine instructions is determined as a "target" language in the Morris diagram (Figure 1.1). Intuitively, instructions of the Machine language are divided into three groups. The data modification group contains \{push,load,store\} instructions. The mathematical operation contains \{multiply,divide\}, \{mod,add,minus,lessThan,greaterThan,equal,and,or\} instructions. The control group contains \{jump,bjump,jumpOnCond,quit\} instructions, especially the "quit" instruction is added at the end to terminate Machine programs. The Machine instructions of the above Minila program is as follows,
Similar to Minila language, the Machine language requires a Virtual Machine to describe the semantics of its programs. After executed, the Environment returned by the Virtual Machine has the value as same as the above Environment of the Interpreter. In this example, two returned Environments are semantically equivalent.

1.2.2 The correctness of Minila’s Compiler

Compiler of Minila must sure that a sequence of machine instructions that generated from a Minila program preserves the semantic of this program. However, the semantics of Minila programs and machine instructions are specified into Environments by apply Interpreter and Virtual Machine, respectly. Therefore, verifying the correctness of Compiler in particular is just comparing two Environments of corresponding Minila program and machine instructions. The Compiler’s correctness is defined as follows

**Definition 1.2.1** When a machine instruction sequence is generated from an arbitrary Minila program, the Compiler must satisfy that the Environment returned from execution of this program by Interpreter is semantically equivalent to the Environment returned from execution of corresponding instruction sequence by Virtual Machine.

This definition is illustrated by the following diagram,

![Diagram](image)

Figure 1.2: The Correctness of Minila’s Compiler Diagram

In this diagram, all nodes are algebras which are specified by CafeOBJ modules and the arrows denote algebraic functions. Using CafeOBJ specification, the arrows are specified into Interpreter, Compiler and Virtual Machine modules that contain re-writing rules defined the meaning of `interpret`, `compile` and `vm` functions, respectly. The "≡" symbol means the semantic equivalence between two Environments. On other words, it indicates the correctness of Compiler for Minila language.
1.3 Motivation

As mentioned in the background section, the imperative programming language and its compiler have significant roles in computer science. Compiler verification is a big challenge for high-assurance software. In this document, we attempt to verify the correctness of the compiler for Minila language. The research utilizes CafeOBJ language for specification and verification. The specific objectives are as follows,

- Specify the Minila language and its components.
  The Compiler, Environment, Interpreter, Virtual Machine etc.
- Formalize the correctness of Compiler.
- Verify the correctness property by conducting proof scores.

The successful verification will be an important evidence for the compiler verification of imperative programming language.

This document contains 4 chapters. Chapter 1 introduces the background, the Minila and the motivation. Chapter 2 shows specifications of the Minila and its components in detail. Chapter 3 demonstrates the main part of this research, verifying the correctness of Compiler for Minila language. And chapter 4 shows the conclusion.
Chapter 2

Specification

2.1 Previous Works

Before showing specifications, we should mention agains previous achievements. In previous works, we verified the correctness of Compiler for expressions of Minila language. Because just focused on the expression aspect, the specifications and verifications are too limited. The storage feature of imperative language (the Environment) did not consider in previous specifications. Expressions were be evaluated without the information of Environment. In summary, the current specifications and verifications are too different with that of previous works. We will show the differences in next sections, but now we continue showing current specifications.

2.2 Specifications

In the example 1.1.3.2, the BASIC-NAT module doesn’t declare combinations of many natural numbers. However, we need some arithmetic operations for further evaluations, hence the PNAT module is constructed as follows,

```plaintext
mod! PNAT principal-sort Nat {
  pr(BASIC-NAT)
  --
  op _+_ : Nat Nat -> Nat {comm assoc}
  op _*_ : Nat Nat -> Nat {comm assoc}
  op sd : Nat Nat -> Nat
  op _quo_ : Nat NzNat -> Nat
  op _rem_ : Nat NzNat -> Nat
  -- _+_ 
    eq 0 + N:Nat = N .
    eq s(M:Nat) + N:Nat = s(M + N) .
  -- _*_ 
    eq 0 * N:Nat = 0 .
```
eq \( s(M:\text{Nat}) \times N:\text{Nat} = N + (M \times N) \).

-- sd
eq sd(0,N:\text{Nat}) = N .
eq sd(s(M:\text{Nat}),0) = s(M) .
eq sd(s(M:\text{Nat}),s(N:\text{Nat})) = sd(M,N) .

-- quo
eq M:\text{Nat} quo s(N:\text{Nat}) = (\text{if } M < s(N) \text{ then } 0 \text{ else } sd(M,s(N)) \text{ quo } s(N) \text{ fi}) .

-- rem
eq M:\text{Nat} rem s(N:\text{Nat}) = (\text{if } M < s(N) \text{ then } M \text{ else } sd(M,s(N)) \text{ rem } s(N) \text{ fi}) .

Using the importation feature, pr(BASIC-NAT), the PNAT module inherits all properties of the BASIC-NAT. It also declares some arithmetic operators, "+" for the addition, "*" for the subtraction, sd for the symmetric difference, quo for the division and rem for the modulo.

In the PNAT module, the quo is specified with the second input belongs NzNat sort, it means the divisor of division is not zero. A divisor equals zero is an exception case in PNAT; the same things also exist for the rem operator. In this document, these exceptions will be ignored because natural numbers only used for specifying the values. Therefore, the value of data should be well-defined and in general, its exceptions could not effect on the exceptions of Minila language.

2.2.1 Minila language

When considering the whole language, the role of Enviroment is very important, effecting not only on the initial state but also the semantics of programs. Therefore, we have to refine previous works, as well as adding the effects of Environment into Minila’s specifications. Let’s begin with a basic component of Minila language, the variable.

Variable

A function \( v(i) \) with the index \( i \) is a natural number determines the name of Minila’s variables. The variable is specified as follows,

mod! VAR {
  pr(BASIC-NAT)
  [Var]
  --
  op v : Nat -> Var {constr} .
op _=_ : Var Var -> Bool {comm} .
eq (V:Var = V) = true .
eq (v(N1:Nat) = v(N2:Nat)) = if (N1 = N2) then true else false fi .
}
Note 2.2.1. We use upper case letters to distinguish objects’ name. Names with all upper case (i.e. VAR) are module’s names but names with first upper case (i.e. Var) are sort’s names.

Variable requires natural numbers to index its name. An index is just a basic number which does not contain arithmetic operations, hence the BASIC-NAT is imported into the VAR module replaced for the PNAT. The operator v specifies the function \( v(i) \) and it’s the form of sort Var. Therefore, we use the "constr" attribute of CafeOBJ, constructure of a sort, for the v operator.

The name of variables could be compared by using the \( \_\_\_\_ \) operator. The equation between two names has the commutativity property, so it’s declared with comm attribute. Two names are determined same if both are similar or having a same index, and this meaning is already defined by re-writing rules. However, a Minila variable is a pair of the name and its value, hence it’s specified by a module called ”Entry”.

**Entry**

mod! ENTRY principal-sort Entry {
    pr(BASIC-NAT) pr(VAR)
    [Entry]
    --
    op \<_\_,_\> : Var Nat -> Entry
    op \_=\_ : Entry Entry -> Bool {comm} .
    -- equations
    eq (\<_V1:Var , N1:Nat \> = \<_V2:Var , N2:Nat \>) = ((V1 = V2) and (N1 = N2)) .
}

The value of a variable is also a basic natural number, so the BASIC-NAT is imported. Two variables are determined equal if they have the similar name and the same value.

**Expression**

A Minila expression is defined as a natural number, a variable or the combinations of expressions. It is specified as follows

mod! EXP {
    pr(BASIC-NAT) pr(VAR)
    [Nat Var < Exp ]
    [Exp ExpErr < ExpFull]
    --
    op exp-err : -> ExpErr
    --
    eq (E:ExpFull = E) = true .
    eq (E:Exp = exp-err) = false .
}
In the EXP module, the Nat and Var are subsorts of the Exp which denotes expressions. Consequently, every instants belong Nat or Var are considered as expressions. Moreover, an expression is the combination of two expressions by many operations which denoted by \{ ++, --, **, //, %%%, ==, !==, <<, >>, &&, || \} operators. In order that operators of the natural number don’t appear inside expressions, the EXP imports the BASIC-NAT module replace for the PNAT. It means that expressions containing \"\{+ , sd, *, quo, rem\}\" will cause errors.

When the complex combinations of many expressions with different kinds of operators are evaluated, the operators need associativity and precedence propterties. In the EXP, they are specified by l-assoc (left associativity) and prec: attributes. The number after prec: is bigger, the precedence of corresponding operator is lower. For example, 
\(e1 ++ e2 ** e3 // e4\) is evaluated as 
\((e1 ++ ((e2 ** e3) // e4))\) with 
\(e1, e2, e3, e4\) are expressions.

Intuitively, the Minila expressions have the data type is natural numbers. Therefore, there are exceptions (i.e. second input of the ”//” operator equals zero) when we evaluate combinations of the expressions. We will consider in detail methods for exceptions at section 2.4. The value of a Minila expression is evaluated with respect into an Environment, hence the semantics of operations are not defined by re-writing rules in the EXP. It’ll be evaluated inside the Interpreter and the Compiler which will be shown in next sections. Now, we consider basic structures of Minila programs, the Statement.
Statement

From the definition 1.2.1, the syntax structure of Minila statements are formalized into operators in CafeOBJ. However, the empty statement is determined as a constant, an operator without input sorts. The specification in STM module is as follows,

```
mod! STM {
  pr(EXPR)
  [Stm]
  --
  op estm : -> Stm {constr} . -- empty statement
  op _=_ : Stm Stm -> Bool .
  --
  eq (S:Stm = S) = true .
  -- assignment statement
  op _:=_; : Var Exp -> Stm {constr}
  -- if statement
  op if_then_else_fi : Exp Stm Stm -> Stm
  -- while statement
  op while_do_od : Exp Stm -> Stm
  -- repeat statement
  op repeat_until_ : Stm Exp -> Stm
  -- for statement
  op for_ from _ to _do_od : Var Exp Exp Stm -> Stm
  -- sequential statement
  op (_ _) : Stm Stm -> Stm {r-assoc id: estm}
}
```

The empty and assignment statement are considered as the basic structure of Minila’s statements because their structures do not depend on other statements. Therefore, they are declared with the constr attribute.

The sequential statement, denoted by "(\_)\)\)\", describes that two statements are constituted as a statement with the r-assoc (right associativity) attribute. The id (identity) attribute is estm means that two empty statements are constituted an empty statement. Intuitively, with definition of the sequential statement, a list of statetements are considered as a statement. Comparing with the definition that a Minila program is a list of statement, we have the following definition.

**Formalization 2.2.1** A Minila program is considered as a Minila statement.

\[
\text{A program } \iff \text{A statement}
\]

From now, a statement is used to instead of Minila programs. And the correctness of Compiler for Minila programs becomes the correctness of Compiler for the statements, which will be mentioned in detail in further sections.
2.2.2 Environment

The Environment model is a denotation of the Store entity in the formalization 1.1.1. The Environment contains variables and the data values which is utilized to describe the semantics of Minila programs. Therefore, the semantic of a program is specified with respect into an Environment.

An implementation of the Environment model is a list of pairs of variables and the values, hence before considering the specification of Environment, we should refer to specification of a general list, which is specified in a LIST module as follows,

```
mod! LIST (M :: EQTRIV) {
  pr(PNAT)
  [Nil NnList < ListNormal]
  [ListNormal ListErr < ListHalt]
  [ListHalt ListNonHalt < List]

  op nil : -> Nil {constr} .
  op list-err : -> ListErr {constr} .
  op list-non-halt : -> ListNonHalt {constr} .
  op _|_ : Elt.M ListNormal -> NnList {constr} .

  op _=_ : List List -> Bool {comm}
  op _@_ : ListNormal ListNormal -> ListNormal {assoc}
  op nth : ListNormal Nat -> Elt.M
  op len : ListNormal -> Nat

  eq (nil = (X:Elt.M | L:ListNormal)) = false .
      if (X = Y) then L = L1 else false fi .
  eq (L:List = L) = true .
  eq (L:ListNormal = list-err) = false .
  eq (L1:ListHalt = list-non-halt) = false .

  op nil @ L:ListNormal = L .
  eq (X:Elt.M | L1:ListNormal) @ L:ListNormal = X | (L1 @ L) .

  nth
  eq nth((X:Elt.M | L:ListNormal),0) = X .
  eq nth((X:Elt.M | L:ListNormal),NN:NzNat) = nth(L,sd(NN,s(0))) .

  len
  eq len(nil) = 0 .
  eq len(X:Elt.M | L:ListNormal) = s(0) + len(L) .
  eq len(L1:ListNormal @ L:ListNormal) = len(L1) + len(L) .
}
```

The LIST module is a parameterized module [3]. LIST is a general list, which has one
parameter declared as $M::EQTRIV$. The $M$ is specified as an abstract element of $EQTRIV$ module which has the $Elt$ sort and the $=$ operator,

$$\text{mod}^* \text{ EQTRIV } \{ \\
\text{ [Elt] } \\
\quad \text{ op } =_\_ : \text{ Elt Elt } \to \text{ Bool } . \\
\}$$

The sort hierarchy of $\text{LIST}$ is quite complicated

$$\begin{align*}
\text{ [Nil NnList } < \text{ ListNormal] } \\
\text{ [ListNormal ListErr } < \text{ ListHalt] } \\
\text{ [ListHalt ListNonHalt } < \text{ List] }
\end{align*}$$

because it’s used to describe the Environment Hierachy as section 2.2

An element of List sort is the $\text{nil}$ (an empty list) or the $\_\_\_\_$ syntax and these forms are determined as $\text{constr}$ forms. Other $\text{constr}$ forms including $\text{list-err}$ and $\text{list-non-halt}$ are determined as exception cases which are used for section 2.4.

The general $\text{LIST}$ supports some operations, the $=$ for equations, the $\emptyset$ for combinations of lists, the $\text{nth}$ outputs the first element of a given list, and the $\text{len}$ calculates the number elements of a given list.

A view is a mapping from the $\text{EQTRIV}$ into the $ENTRY$ module

$$\text{view } \text{EQTRIV2ENTRY from EQTRIV to ENTRY } \{ \\
\quad \text{ sort Elt } \to \text{ Entry, } \\
\quad \text{ op } (=_\_ ) \to (=_\_ ) \\
\}$$

and be used as an instant replaced for the parameter $M$. The general List now, becomes a list of Entries and it’s an implementation of the Environment which is specified as follows

$$\text{mod! ENV principal-sort Env } \{ \\
\text{ pr(PNAT) pr(EXP) } \\
\text{ pr(LIST(EQTRIV2ENTRY) * } \{ \\
\text{ sort Nil } \to \text{ Empty, } \\
\text{ sort List } \to \text{ Env, } \\
\text{ sort ListNonHalt } \to \text{ EnvNonHalt, } \\
\text{ sort ListHalt } \to \text{ EnvHalt, } \\
\text{ sort ListNormal } \to \text{ EnvNormal, } \\
\text{ sort ListErr } \to \text{ EnvErr, } \\
\text{ op nil } \to \text{ empty, } \\
\text{ op list-err } \to \text{ env-err, } \\
\text{ op list-non-halt } \to \text{ env-non-halt } \\
\} \}$$
op update : Var Nat EnvNormal -> EnvNormal
op update : Var ExpFull Env -> Env
--
op lookup : Var EnvNormal -> ExpFull .
-- update: insert into Env even if the variable already .
eq update(V:Var,exp-err,EV:Env) = env-err .
eq update(V:Var,N:Nat,EV:EnvNormal) = (< V , N > | EV) .
-- lookup
  eq lookup(V:Var,empty) = exp-err .
eq lookup(V:Var,< V1:Var,N1:Nat > | E:EnvNormal) =
    if V1 = V then N1 else lookup(V,E) fi .
}

In the ENV module, after imported, the specific List with Entry parameter changes the sorts' name and these sorts become sorts of the ENV. Variables inside the Env sort are modified by using update and lookup operators. The update is just adding a new pair of variable and value into the Environment. In general, variables or expressions of Minila language are evaluated with respect into an Env sort. This is exactly what we mentioned before at the definition of Minila language.

### 2.2.3 Machine language

A Machine language is defined as an ordered sequence of machine instructions. A sequence of instructions of the Machine language is considered as a list of commands. A Command sort denotes all kinds of instructions and is specified in a COMMAND module as follows,

**Command**

mod! COMMAND principal-sort Command {
  pr(BASIC-NAT) pr(VAR)
  [Command]
  --
op push : Nat -> Command
op load : Var -> Command
op store : Var -> Command
op add : -> Command
op minus : -> Command
op multiply : -> Command
op divide : -> Command
op mod : -> Command
op lessThan : -> Command
op greaterThan : -> Command
op equal : -> Command
op notEqual : -> Command
op and : -> Command
op or : -> Command
op jump : Nat -> Command
op bjump : Nat -> Command
op jumpOnCond : Nat -> Command
op quit : -> Command

--

op _=_ : Command Command -> Bool
vars N1 N2 : Nat
vars V1 V2 : Var
eq (C:Command = C) = true .
eq (push(N1) = push(N2)) = (if N1 = N2 then true else false fi) .
eq (load(V1) = load(V2)) = if (V1 = V2) then true else false fi .
eq (store(V1) = store(V2)) = if (V1 = V2) then true else false fi .
eq (jump(N1) = jump(N2)) = if N1 = N2 then true else false fi .
eq (bjump(N1) = bjump(N2)) = if N1 = N2 then true else false fi .
eq (jumpOnCond(N1) = jumpOnCond(N2)) = if N1 = N2 then true else false fi .
}

Elements belong the Command might be constants (i.e. add, minus,...) that don’t have input sorts or normal operators (i.e. push, load,...). We also define the equation between two commands, it will be used when the Machine language is compared.

The Machine language is defined as a list of instructions (commands), therefore we use the LIST module again with the parameter is a view from the EQTRIV to the COMMAND module

view EQTRIV2COMMAND from EQTRIV to COMMAND {
  sort Elt -> Command ,
  op (_=_) -> (_=_)
}

The CLIST module is specified as a LIST module with actual parameter is the COMMAND.

mod* CLIST principal-sort CList {
  pr(LIST(EQTRIV2COMMAND) * {sort ListHalt -> CListHalt,
                    sort ListNormal -> CList,
                    sort ListErr -> CListErr,
                    op list-err -> clist-err,
    }
  )
}

With above specification, we can conclude that
Formalization 2.2.2  Machine Instructions $\iff$ A CList

From now, we will use a CList when we want to mention about machine instructions.

2.2.4 Compiler

In the diagram 1.2, the Compiler transforms Minila programs into Machine instructions. It means the Compiler gets a Statement as the input and then outputs a CList. Therefore, the Compiler is specified as a group of re-writing rules. The COMPILER differs with above modules such as STM or ENV because it doesn’t have own sorts. The specification of COMPILER module is as follows,

mod! COMPILER {
    pr(BASIC-NAT) pr(STM) pr(CLIST)
    op compile : Stm -> CList .
    op generator : Stm CList -> CList .
    op genForExp : Exp -> CList
    --
    eq compile(S:Stm) = generator(S,nil) @ (quit | nil) .
    -- estm
    eq generator(estm,CL:CList) = CL .
    -- V := E ;
    eq generator(V:Var := E:Exp ; S:Stm,CL:CList)
        = generator(S, CL @ genForExp(E) @ (store(V) | nil) ) .
    -- If E then S1 else S2 fi
    eq generator(if E:Exp then S1:Stm else S2:Stm fi S:Stm,CL:CList)
        = generator(S, CL @ genForExp(E)
            @ (jumpOnCond(s(s(0))) |
                jump(len(generator(S1,nil)) + s(s(0))) | nil)
            @ generator(S1,nil)
            @ (jump(len(generator(S2,nil)) + s(0)) | nil)
            @ generator(S2,nil) ) .
    -- while E do S1 od
    eq generator(while E:Exp do S1:Stm od Stm:Stm,CL:CList)
        = generator(Stm, CL @ genForExp(E)
            @ (jumpOnCond(s(s(0))) |
                jump(len(generator(S1,nil)) + s(s(0))) | nil)
            @ generator(S1,nil)
            @ (bjump(len(genForExp(E)) + len(generator(S1,nil))
                + s(s(0))) | nil)) .
    -- repeat S1 until E
    eq generator((repeat S1:Stm until E:Exp) Stm:Stm,CL:CList)
        = generator(Stm, CL @ generator(S1,nil)
The function of Compiler is defined by the `compile` operator. It gets a `Stm` as the input and outputs `CList`. The `compile` is considered as a total functions.

\[
\text{op} \quad \text{compile} : \text{Stm} \rightarrow \text{CList}.
\]

However, the `compile` creates a list of commands by calling the `generator` operator and the output obtained by the `generator` is added with the `quit` command,

\[
\text{eq} \quad \text{compile}(S:\text{Stm}) = \text{generator}(S,\text{nil}) @ (\text{quit} | \text{nil}).
\]

Depending on the syntax of each statement, there are corresponding re-writing rules for the `generator`. When evaluating a statement that contains expressions (i.e. assignment, if or while statements), the `generator` will call a `genForExp` operator. Then, depending on
the giving expression is a number, a varible or combinations of expressions, the `genForExp` will use the proper re-writing rules to transform that expression into a list of commands.

We already have the STM module (specification of Minila programs), the CLIST module (specification of Machine instructions), and the COMPILER module which defines re-writing rules for transformation from a Stm into a CList. Now we need to verify that semantics of the CList whether equal semantics of the corresponding Stm or not. In order to do that, we compare two Enviroments that are obtained after executed the Stm and CList. We will consider an Interpreter and a Virtual Machine to execute the Stm and CList, respectly.

When executing an arbitrary statement Stm, we must consider all cases which might happen. The normal case is when the execution returns a well-form Environment, a list of pairs of variables and values. Besides, there are two situations which are determined as exceptions, or called errors.

+ variables or expressions inside the given Stm have errors. As a result, this Stm makes the returned Environment become an error Environment.

+ A given Stm has an infinite loop caused by specific conditions of while or repeat statements. Therefore, when executing Stm, there are no Environment returned.

To solve the exception, called *Error Handling*, we will design new sorts that denotes the error entities.

### 2.2.5 Sort Hierarchy

In the EXP module at 2.2.1, we have a sort hierarchy for expressions,

\[
[\text{Exp} \text{ ExpErr} < \text{ExpFull}]
\]

\[
\text{op exp-err : } \rightarrow \text{ExpErr}
\]

All normal expressions are elements of the Exp sort. However, when we detect errors in expressions, all these expressions are denoted by the exp-err which belongs the ExpErr sort. The ExpErr is a dedicated sort for error expressions, but here only one constant exp-err is the denotation for every instants of error expressions. The method solves a problem that an expression belongs the sort ExpFull (it maybe belongs the ExpErr) but if it not equal exp-err, it's not an error expression.

The equation between two expressions also defines that an element of sort Exp does not equal the exp-err constant

\[
\text{eq (E:Exp = exp-err) = false .}
\]

from this, we illutrate the sort hierarchy for expressions as follows

The sort hierarchy not only specifies dedicate sorts for error cases, but also describes the relation between the normal cases and error cases. Furthermore, The Case Splitting (3.2.2) will become clear and more accurate based on the architecture of sort hierarchies.
2.3 Environment Hierarchy

Now, we consider another sort hierarchy used for the Environment, called Environment Hierarchy. It’s more complicated than sort hierarchy for the expression, and already defined in the LIST module but the sorts are renamed in the ENV module. The Environment Hierarchy is

$$[\text{EnvNormal} \quad \text{EnvErr} < \text{EnvHalt}]$$
$$[\text{EnvHalt} \quad \text{EnvNonHalt} < \text{Env}]$$

The sort relation of Environment Hierarchy is illustrated as follows
Environments belong the EnvNormal sort. However, when the execution of a Stm reaches an error, the execution is terminated and the returned Environment is determined as an error Environment. All instants of the error Environment are specified by only one constant env-err which is declared at op env-err : -> EnvErr. And, similar to the expression case, an arbitrary element of EnvNormal sort is not equal to the env-err. It's already defined in the LIST moduleat eq (L:ListNormal = list-err) = false . Because the execution of above cases are terminated and having a specific Environment, the EnvNormal and EnvErr are determined as subsorts of the EnvHalt.

There is a case that a Stm has infinite loop because of while or repeat statement inside the Stm. As a result, the execution will not terminate (non-termination), and no specific Environment is obtained for this case. However, the Environment for this Stm execution could be determined as a element that belongs of the EnvNonHalt sort. We use a constant env-non-halt to denote those Environments. Finally, the env-non-halt is different with an element of the EnvHalt sort, and already defined in the LIST module by

\[
\text{eq (L:ListHalt = list-non-halt) = false}.
\]

In general, we have three dedicated sorts: EnvNormal, EnvErr and EnvNonHalt. Each of them describes specific semantic of Minila programs. The semantics of a normal program is specified by an element of EnvNormal sort, the semantics of error programs are specified by the env-err of EnvErr sort. And when programs don’t have errors but they are not terminate, the semantics are specified by the env-non-halt of EnvNonHalt sort.

In this research, the termination is really a big problem for verifications, hence we emphasise the termination aspect in the Environment Hierachy. In this context, the error Environment, EnvErr, is specified as a termination Environment and it’s a subsort of the EnvHalt. This is contrary with the EnvNonHalt, non-termination Environment.

However, if EnvErr and EnvNonHalt are considered with error aspect, we will have another sort hierarchy for Environment. Intuitively, the infinite loop Stm is a statement without error, therefore it could be determined as a normal statement. We illustrate an alternative of sort hierarchy for Environment as follows.

This hierarchy explained another aspect of the sort hierarchy for Environment. We will have the global view about programs of Minila language, as well as deep understand for sort hierarchy of an issue. In this document, we focus on the termination program, hence we choose the first design for the Environment Hierachy.

### 2.4 Error Handling

There are some reasons for the error of expressions. Because an expression might be constructed from variables, error variables will make the expression error. However, we assume that the \(v(i)\), variable's name, does not have errors. Therefore there only have one error case, that is when a looking variable does not exist in a given Environment, which is specified in ENV module at 2.2.2

\[
\text{eq lookup(V:Var,empty) = exp-err}.
\]
eq lookup(V:Var,< V1:Var,N1:Nat > | E:EnvNormal) =
  if V1 = V then N1 else lookup(V,E) fi.

the lookup operator finds given variable V:Var from the head of an Environment. It
returns the value N1:Nat if the V already exists. However it will return exp-err when
reaching to the tail of the Environment, empty. For simple specification, we use exp-err
which for error expressions to denote the error variable. Consequently, we don’t need to
design a sort hierarchy for the variables.

2.4.1 Interpreter

Another reason which makes expressions error is the arithmetic calculations of expressions.
There are some operators which might cause errors such as the //, % with a second
input is evaluated as 0 or the "--" operator with the first input is less than the second
input. All operators for expressions are defined at INTERPRETER module, but we illustrate
with some operators as follows,

mod! INTERPRETER {
  pr(PNAT) pr(STM) pr(ENV)
  ...
  op evalExp : Nat EnvNormal -> Nat .
  op evalExp : Exp Env -> ExpFull .
  ...
  eq evalExp(V:Var, EV:EnvNormal) =
    if (lookup(V, EV) = exp-err) then exp-err else lookup(V, EV) fi .
  ...
}

27
eq evalExp(E1:Exp -- E2:Exp, EV:EnvNormal) =
    if (evalExp(E1, EV) = exp-err or evalExp(E2, EV) = exp-err)
        then exp-err
        else if (evalExp(E1, EV) < evalExp(E2, EV))
            then exp-err
            else sd(evalExp(E1, EV), evalExp(E2, EV)) fi fi .

the evalExp is a operator which evaluate the value of an expression. When the expression
is a natural number N:Nat, it is evaluated as N. We declare the evalExp for this case by

    op evalExp : Nat ExpNormal --> Nat .

When the expression is a variable V:Var, there are two cases. If the V is not be found
(lookup(V, EV) = exp-err) then that expression is evaluated as exp-err, otherwise it is
evaluated by the value of lookup(V, EV).

When the expression is (E1:Exp -- E2:Exp), it's easy to conclude that when E1 or
E2 are exp-err then that expression is exp-err. Moreover, when the minuend E1 is less
than the subtrachend E2, (evalExp(E1, EV) < evalExp(E2, EV)), it means the expression
is exp-err. The < operator of natural number is required, therefore the INTERPRETER
module must import the PNAT than the BASIC-NAT.

In both variables and combination of expressions, the output value of expressions might
be a normal value (E:Exp) or an error value (exp-err:ExpErr). Therefore, we must
declare the evalExp for this meaning by

    op evalExp : Exp Env --> ExpFull

the sort ExpFull is declared to cover Exp and ExpErr subcases (Fig. 2.1) With different
Nat or Exp input sort, the output sort of the evalExp is different, Nat or ExpFull. The
correspondence between the input and output sort is a technique of the Error Handling.

The INTERPRETER uses the interpret operator to specify the semantics of a Stm
statement. The interpret will call an immediate operator, eval, to do its work. Depending on
a specific structure of the Stm, the eval does different tasks, doing nothing for estm
statement or calling evalAssign, evalIf, evalWhile, evalFor, evalRepeat for assign,
if, while, for, repeat, respectively. We declare these operators as follow,

    op interpret : Stm Env --> Env
    op eval : Stm Env --> Env
    op evalAssign : Var Nat EnvNormal --> EnvNormal
    op evalAssign : Var ExpFull Env --> Env
    op evalIf : Exp Stm Stm Env --> Env
    op evalWhile : Exp Stm Env --> Env
    op evalRepeat : Stm Exp Env --> Env
    op evalFor : Var Exp Stm Env --> Env .

With only the evalAssign, the output Environment could be determined as EnvNormal
with the Nat, EnvNormal input sorts. The output Environment of other operators could
not determined clearly as EnvNormal, EnvErr or EnvNonHalt because it depends on the specific input Stm and Env.

The output Environment will be the env-err if a input Stm contains an expression which is evaluated as exp-err or a input Environment is the env-err. The output Environment will be the env-non-halt if a input Stm contains infinite loop While or Repeat statements or a input Environment is the env-non-halt. We take the specification of While statement as an example,

\[
\text{ceq eval}(S:Stm, EV:Env) = EV \text{ if } (EV = \text{env-non-halt} \text{ or } EV = \text{env-err}) .
\]

\[
\text{eq eval(while } E:Exp \text{ do } S1:Stm \text{ od } S:Stm, EV:EnvNormal) =
\]
\[
\text{eval}(S, \text{evalWhile}(E, S1, EV)) .
\]

\[
\text{eq evalWhile}(E:Exp, S1:Stm, EV:Env)
\]
\[
= \text{if } (EV = \text{env-non-halt})
\]
\[
\text{then env-non-halt}
\]
\[
\text{else if } (EV = \text{env-err} \text{ or } \text{evalExp}(E, EV) = \text{exp-err})
\]
\[
\text{then env-err}
\]
\[
\text{else if } \not{\text{evalExp}(E, EV) = 0}
\]
\[
\text{then evalWhile}(E, S1, \text{eval}(S1, EV))
\]
\[
\text{else } \text{eval}(estm, EV) \quad \text{fi fi fi} .
\]

The first eval equation is a conditional equation which check the input Environment is env-non-halt or env-err with a general Stm input. If the condition happens, the execution will be stopped here and the output Environment will be the input Environment. After this conditional equation, other eval equations are specified with the EV:EnvNormal input like the second equation. The input statement of the second eval is a sequential statement which contain a While statement and a general S:Stm following. Therefore, the eval operator will be repeated for the general S but the input Environment is a result Environment of execution for the While by calling the evalWhile.

If looking at the second equation, the EV:EnvNormal is directly transfered into the evalWhile of the third equation. However, the input Environment of evalWhile operator is not a EV:EnvNormal. Our experiments show that the input must be a general Environment, EV:Env. Therefore, the evalWhile checks the input EV:Env for env-non-halt or env-err first. It also checks that evaluation of the E:Exp with respect into EV is exp-err or not. The input E expression of the While always is element of a normal Exp sort. We only know this E is an error expression or not after evaluated by the evalExp operator. If the expression in While statement is an error expression, the output Environment for this statement is env-err.

In summary, the input Environment of the eval or evalWhile operators must be considered as a general EV:Env even though it's transfered with a EV:EnvNormal from previous operators. The EV must be checked to be env-non-halt or env-err in specifications of every operators. This is our experiments for Error Handling.
2.4.2 Virtual Machine

Similar to the Interpreter, a Virtual Machine is a function which describes algebraic semantics of the Machine language. After executed a CList, the specification of an arbitrary Machine language, the semantics of this CList is specified by an output Environment. However, for doing the CList’s execution, a new structure called a stack is required. The stack is implemented as a list of natural numbers. Therefore, a view from the EQTRIV module to the PNAT

view EQTRIV2PNAT from EQTRIV to PNAT {
    sort Elt -> Nat,
    op (_,=_) -> (_,=_),
}

is used as a actual parameter of the LIST.

A STACK module for the stack is specified as follows,

mod* STACK {
    pr(EXPR)
    inc(LIST(EQTRIV2PNAT) * {
        sort ListHalt -> Stack,
        sort ListNormal -> StackNormal,
        sort ListErr -> StackErr,
        op nil -> empstk,
        op list-err -> stkErr
    })
    -- define for StackErr cases
    op _|_| : ExpFull Stack -> StackErr .
    --
    ceq (E:ExpFull | Stk:Stack) = stkErr if (E = exp-err) .
    eq (V:Var | Stk:Stack) = stkErr .
}

The Virtual Machine uses a vm operator to specify the specifications of a CList with respect into an Env. The vm transfers its input into a exec operator as well as adding Nat and Stack sorts with initial values are 0 and empstk, respectly. The exec calls a exec2 operator to do it’s tasks. These operators are declared as follows,

mod! VM {
    pr(CLIST) pr(ENV) pr(EXPR) pr(STACK)
    --
    op vm : CList Env -> Env .
    op exec : CList Nat Stack Env -> Env
    op exec2 : Command CList Nat Stack Env -> Env
}

The definition of these operators could be referened for more details at the source codes.
2.4.3 3-steps Detection of Error Handling

A method for Error Handling could be summarized as a 3-steps detection. We use above specification for expressions as an example to illustrate each step of this method.

- **Step 1: Sort hierarchy and error constants.**
  At this first step, we design a sort hierarchy for that error situations are denoted by a sort. In the sort hierarchy of expressions, we use the $\text{ExpErr}$ sort to denote error expressions. Moreover, a constant is declared as a element of this sort (e.g. the $\text{exp-err}$ is a element of the $\text{ExpErr}$). There is only one constant for each error sort. When error situations are detected, all of them are specified by this constant (e.g. the $\text{exp-err}$). The sort hierarchy must show that all elements of a normal sort are not equal to the error constant. In this example, all elements of $\text{Exp}$ sort are not equal to the $\text{exp-err}$.

- **Step 2: Sort difference for operators.**
  Based on the sort hierarchy, operators which used for execution are declared. For the expressions, the $\text{evalExp}$ is declared,
  
  \[
  \text{op evalExp} : \text{Nat ExpNormal} \rightarrow \text{Nat} .
  \]
  
  \[
  \text{op evalExp} : \text{Exp Env} \rightarrow \text{ExpFull} .
  \]
  
  Depending on the input sorts $\text{Nat}$ or $\text{Exp}$, the output sort will be $\text{Nat}$ or $\text{ExpFull}$, respectly. This is called the *sort difference*. At this step, we have just detected the output sort but we don’t obtain a specific output value. It’ll be done at the next step.

- **Step 3: Re-writing rules for run-time executions.**
  
  \[
  \text{eq evalExp}(V:\text{Var},EV:\text{EnvNormal}) =
  \begin{cases} 
  \text{exp-err} & \text{if } (\text{lookup}(V,EV) = \text{exp-err}) \\
  \text{lookup}(V,EV) & \text{else}
  \end{cases}
  \]
  
  The "run-time" means that executions will be done with specific input value. The equation shows that with a specific $V:\text{Var}$, the run-time execution determines that the output will be a normal value or an error expression, denoted by $\text{exp-err}$ is detected. The error detection is defined by re-writing rules inside specifications.

In the 3-steps detection, we could not find the sort difference for some operators such as the $\text{interpret}$, $\text{eval}$ or $\text{evalIf}$ operators of Interpreter. Therefore the step 2 could not be applied. However, another way of the sort difference will be done at the step 3. As a result, the step 1 and step 3 are enough for Error Handling at that situation.
Chapter 3

Compiler Verification

In this chapter, we will verify the correctness property of the Compiler for Minila language. First of all, the correctness of the Compiler must be defined into a formalization, called theorem. After that, proof scores will be conducted for this theorem. Finally, the proof assistant of CafeOBJ executes these proof scores. When all proof scores are passed (return true), it means that the correctness property of the Compiler is satisfied.

Now, we consider the formalization that describes the correctness of Compiler for Minila language.

3.1 Correctness of Minila’s Compiler

3.1.1 Previous Formalization

In previous works, we have already verified the correctness of Compiler for expressions. In the previous context, the correctness property could be defined intuitively as follows,

Definition 3.1.1 If the Interpreter returns a natural number as the result of executing an expression, then the Compiler generates an instruction sequence for the expression and the Virtual Machine returns for the instruction sequence the same natural number as the result returned by the Interpreter.

From specifications, we obviously realize that the Interpreter describes algebraic semantics of Minila programs. And the Virtual Machine describes semantics of Machine language. The Compiler transforms Minila programs into sequences of Machine instructions, and this’s one way direction. Therefore, in the definition of correctness property, the value returned by the Interpreter is determined as an assumption for comparing with the value from Virtual Machine. In other words, the Interpreter is considered as an oracle in that definition.

In previous works, semantics of expressions are evaluated with no respect to the Environment. Therefore, specifications of the Interpreter, Compiler, and Virtual Machine are quite different with the specifications mentioned in the chapter 2.

op interpret-exp : Exp -> Nat


\[
\text{op compile-exp : Exp -> CList}
\]

\[
\text{op vm-exp : CList -> Nat}
\]

Input sort of the `interpret-exp` is only the `Exp`. The `interpret-exp` doesn’t have `Env` input sort like the `interpret` at 2.4.1, and the `vm-exp` is too. The formalization of the correctness for Minila’s Compiler in previous works is as follows:

**Formalization 3.1.1**

\[
\text{th-exp(E:Exp,N:Nat) =}
\]
\[
( \text{interpret-exp(E) = N implies vm-exp(compile-exp(E)) = N} )
\]

Because the Interpreter is as an oracle, we select the `implies`, a CafeOBJ keyword, to specify this meaning. The `implies` indicates that if the Interpreter could not return a number, the `th-exp` theorem is trivial.

In fact, previous specifications and verifications could not be used again in this research. However, they’re really meaningful works, and ideas about the correctness property of Compiler is still remained in this document.

### 3.1.2 Formalization for Correctness of The Compiler

At 3.1.2, we have the formal definition of the correctness of Compiler for Minila language. However, we already know that a Minila program is specified as a \( S : \text{Stm} \) statement. Therefore, the correctness of the Compiler for a statement \( S \) is defined as follows.

**Definition 3.1.2** The Environment returned after executing an arbitrary \( S \) statement by the Interpreter must have semantic equivalence of the Environment returned by the Virtual Machine from the execution of instruction sequence which was generated from \( S \) by the Compiler.

In the chapter 2, the Interpreter, Compiler, and Virtual Machine have already been specified. The summary is showed as follows,

\[
\text{op interpret : Stm Env -> Env} .
\]
\[
\text{op compile : Stm -> CList} .
\]
\[
\text{op vm : CList Env -> Env} .
\]

In constrast to `interpret-exp`, an `Env` is required for the `interpret`. The same thing happens with the `vm`. Moreover, a `Stm` which denotes a Minila statement is considered replaced for the `Exp` in previous works. Extentions from the `Exp` into the `Stm` and adding the information of `Env` make the verifications more complicated. However, the ideas about the correctness of Compiler will not change, there is an equivalence between two Environments returned by the Interpreter and Virtual Machine of one statement.

Concerning with the return of Environments, there is a problem that the executions of some statement might not return Environements both by the Interpreter and Virtual Machine. This occurs with iterative statements including the `while` and `repeat` that might
be infinite loop. When a statement is infinite loop, called non-termination situation, it contradicts with a finite statement, termination. The non-termination causes many problems for verification of the correctness for Minila statement, and the details will be showed in next sections.

The correctness of the Compiler for Minila language could be formed as follows,

**Formalization 3.1.2**

$$(\text{interpret}(\text{Stm},\text{Env1}) = \text{Env2}) \iff (\text{vm}(\text{compile}(\text{Stm}),\text{Env1}) = \text{Env2})$$

In practice, the \text{implies} relation is enough for definition of the correctness like that defined in previous works. However, we want to emphasize that the correctness of Compiler for Minila statements is defined as the semantic equivalence between two returned Environments. In this formalization, the "semantic equivalence" is expressed by the \text{iff}, the CafeOBJ keyword. The \text{iff} has "if and only if" meaning, it’s stronger than the \text{implies}.

With the non-termination situations, the comparation of semantics could not directly done because there is no Environment returned. For this reason, results of the interpret and \text{vm} are compared through \text{Env2}, an intermediate Environment. When the \text{Stm} is a infinite loop statement, the $$(\text{interpret}(\text{Stm},\text{Env1}) = \text{Env2})$$ equation becomes false and the same thing for the $$(\text{vm}(\text{compile}(\text{Stm}),\text{Env1}) = \text{Env2})$$.

The correctness property is formalized by a theorem called \text{th} and declared as an operator in a \text{THEOREM} module. We should have some background about using CafeOBJ for verifications before considering the \text{th} in details in next sections.

### 3.2 Verification by Using CafeOBJ

In CafeOBJ language, a formalization is verified by executing \text{proof scores}. If all proof scores for a formalization are passed (returned \text{true}), it means that this formalization is validated or succeeded. In other words, the property which specified by this formalization will be satisfied. Therefore, conducting proof scores is very important for verifications.

#### 3.2.1 Proof Scores

**Example 3.2.1.1** This is an example of a proof score

```
open THEOREM
-- arbitrary values
  op var1 : -> Var .
-- assumptions
  eq stm = var := var1 ; .
  eq lookup(var1,env-normal) = nat . -- var1 was already stored in env-normal
-- check
  red th(stm,env-normal,env-halt) .
close
```
The proof score is conducted for the \( \text{th} \) which stored in the \textsc{Theorem} module, hence it begins with "\text{open} \ \textsc{Theorem}" and ending with "\text{close}" keyword. A sentence with "--" keyword at first is a comment which will be not executed by CafeOBJ.

A proof score is usually divided into three segments. An essential segment, called "check", contains a sentence with the "red" (or "reduce") keyword at the head. When a proof score is executed by CafeOBJ, the red will apply re-writing rules in specifications to reduce the following theorem. In the example, the \( \text{th} \) is reduced with arbitrary inputs, \text{stm}, \text{env-normal} and \text{env-halt}.

The arbitrary statement \text{stm} is used many times in proof scores, hence it should be declared in the \textsc{Theorem} module. In this proof score, we consider a particular case of the assignment statement, hence "\text{stm} = \text{var} := \text{var1} ;" is an assumption. The arbitrary variable \text{var1} is declared in the "arbitrary values" segment. In this case, we also assume that the value of \text{var1} is an arbitrary number which already exists in the \text{env-normal}, hence an assumption for "\text{lookup(var1,env-normal) = nat}" is added. All assumptions are put into the "assumptions" segment.

After executed by CafeOBJ, the "\text{red th(stm,env-normal,env-halt)}" returns \text{true}. It means that the \text{th} is satisfied with the variable existing in the Environment. However, there is one more case for the \text{var1}, when the \text{var1} does not exist in the \text{env-normal}. Therefore, for that case, we need an another proof score which has an assumption like "\text{lookup(var1,env-normal) = exp-err}". In general, whether an arbitrary variable exists in the Environment or not, we split into two cases which have corresponding proof scores. This method is called \text{Case Splitting}.

### 3.2.2 Case Splitting

A proof score usually requires some assumptions to be able to reduce into \text{true}. However, when adding an assumption for a proof score, we have to conduct another that is a copy of the proof score but adding contradiction of this assumption. At example 3.2.1.1, when the "\text{lookup(var1,env-normal) = nat}" assumption is adding, we must conduct a following proof score,

\begin{verbatim}
open \textsc{Theorem}
-- arbitrary values
  op \text{var1} : \rightarrow \text{Var} .
-- assumptions
  eq \text{stm} = \text{var} := \text{var1} ; .
-- (\text{lookup(var1,env-normal) = nat}) = \text{false} .
-- \text{var1} was not stored in \text{env-normal}
  eq (\text{lookup(var1,env-normal) = exp-err}) = \text{true} .
-- check
  red \text{th(stm,env-normal,env-halt)} .
close
\end{verbatim}

Intuitively, the "(\text{lookup(var1,env-normal) = exp-err}) = \text{true}" and the "\text{lookup(var1,env-normal) = nat}" are contradicted. The \text{lookup} of \text{var1} returns a
value \texttt{nat} is the opposite to that no value is returned which determined as \texttt{exp-err}, an error situation. There are many form of two assumptions which are contradicted. We must sure that all cases generated by Case Splitting must be contradicted or complemented.

The Case Splitting is also applied based on the structure of the value sort. For example, when evaluating the semantic of expression \texttt{exp} with respect to \texttt{env-normal}, the result might be a natural number or \texttt{exp-err} of the error case. However, a number might be equal 0 or \texttt{s(nat)}, a non zero number. Therefore, we have three cases as follows,

\begin{align*}
\text{eq evalExp(exp,env-normal)} &= 0 . \\
\text{eq evalExp(exp,env-normal)} &= \text{s(nat)} . \\
\text{eq evalExp(exp,env-normal)} &= \text{exp-err} .
\end{align*}

Three cases cover possible values that might be returned. Proof scores for three assumptions make the Case Splitting complemented.

There are many forms of Case Splitting for one property. The above example shows that we need three cases, however if needed, two contradicted assumptions are enough. We could split as follows,

\begin{align*}
\text{eq evalExp(exp,env-normal)} &= 0 . \\
\text{eq (evalExp(exp,env-normal) = 0) = false} .
\end{align*}

When evaluating a statement, there are three cases: the statement terminates in normal, returns error values, or is trapped in infinite loops. The Environment Hierachy designed at 2.2 specifies these cases with the sort \texttt{EnvNormal, EnvErr, EnvNonHalt}, respectively. The Case Splitting for evaluation of \texttt{stm} statement with respect to \texttt{env-normal} by the \texttt{eval} operator of Interpreter is showed as follows,

\begin{align*}
\text{eq eval(stm,env-normal)} &= \text{env1-normal} . \\
\text{eq eval(stm,env-normal)} &= \text{env-err} . \\
\text{eq eval(stm,env-normal)} &= \text{env-non-halt} .
\end{align*}

with \texttt{env-err} and \texttt{env-non-halt} are constants of \texttt{Env} sort. However, the \texttt{env1-normal} is different, it’s just a declaration for an arbitrary element of \texttt{EnvNormal} sort.

When adding \texttt{eval(stm,env-normal) = env-non-halt}, it means that we assume the \texttt{stm} is an infinite statement that not return a specific Environment. We only recognize the property of \texttt{stm} through \texttt{env-non-halt} denotation. It’s really useful for verifications of the non termination situations.

If proof scores for all cases return \textit{true}, Case Splitting will be stopped. If not, other assumptions will be added and we continue applying Case Splitting. In case that a proof score returns \textit{false}, adding assumptions are not effected, we have to find some lemmas to proof this case. We will talk about a technique, called ”Finding Lemmas”, in detail at section 3.3.2. Now, we will explain in detail the theorem \texttt{th}.
3.3 Correctness’s Verification

3.3.1 Theorem \( \text{th} \)

The theorem \( \text{th} \) is an implementation of definition about the correctness of Compiler for Minila statements (programs). As a result, verifying the correctness of Compiler for Minila language is considered as checking proof scores for the \( \text{th} \). Theorem \( \text{th} \) is the detail of formalization 3.1.2.

The \( \text{th} \) is specified in \text{THEOREM} module as follows,

\[
\text{op th : Stm EnvHalt EnvHalt -> Bool}.
\]

\[
\text{th(Stm:Stm,Env1:EnvHalt,Env2:EnvHalt) =}
\]

\[
(\text{interpret(Stm,Env1) = Env2}) \ implies \ (\text{vm(compile(Stm),Env1) = Env2})
\]

The \( \text{th} \) have three input sorts, \( \text{Stm:Stm} \) for an arbitrary statement, \( \text{Env1:EnvHalt} \) for an input Environment, and \( \text{Env2:EnvHalt} \). The input Environment might be error, hence \( \text{Env1} \) is an element of \( \text{EnvHalt} \) sort. The \( \text{Env2} \) is used to compare with output Environments which are generated by the \text{interpret} or \text{vm}. The output of \( \text{th} \) determined as a \text{Bool} value, it means that if the \( \text{th} \) returns \text{true} for all \( \text{Stms} \) then the Compiler will satisfy the correctness property.

Relation between \text{interpret} and \text{vm} is determined as the \text{implies}, it’s not the \text{iff} from formalization 3.1.2. In fact, when conducting proof scores, \text{iff} relation is only satisfied with terminated statements, but the \text{implies} is satisfied with all kinds of statements. Therefore, we must determine the \text{implies} for verification in general. However, in previous works, the \text{implies} is enough for the correctness of Compiler. And the spirit of definition of correctness property is still remained in the theorem \( \text{th4} \), which will be explained in next sections.

In order to verify the theorem \( \text{th} \) for an arbitrary statement \( \text{Stm} \), we apply induction on the \( \text{Stm} \) with

- Basic Cases: The \( \text{th} \) will be verified with input \( \text{Stm} \) is a \text{basic} statement including the \text{empty}, \text{assignment}, \text{if}, \text{while}, \text{for} and \text{repeat} statement.

- Induction Cases: The \( \text{th} \) will be verified with input \( \text{Stm} \) is a sequential statement. The sequential statement is constructed with a basic statement including the \text{assignment}, \text{if}, \text{while}, \text{for}, \text{repeat} statement at first and the following is an arbitrary statement.

The Case Splitting technique is applied to make sure that all cases are specified into proof scores. An arbitrary statement is splitted into many cases for induction as follows,

- Basic Cases: Basic statement
  
  \[
  \begin{align*}
  + \text{stm} & = \text{estm} \quad (\text{Empty statement}) \\
  + \text{stm} & = \text{var} := \text{nat} \quad (\text{Assignment statement}) \\
  + \text{stm} & = \text{var} := \text{vart} \quad (\text{Assignment statement})
  \end{align*}
  \]
+ \texttt{stm} = \texttt{var} := \texttt{exp} ; \quad \text{(Assignment statement)} \quad \textcircled{4} \\
+ \texttt{stm} = \texttt{if} \ \texttt{exp} \ \texttt{then} \ \texttt{stm1} \ \texttt{else} \ \texttt{stm2} \ \texttt{if} \quad \textcircled{5} \\
+ \texttt{stm} = \texttt{while} \ \texttt{exp} \ \texttt{do} \ \texttt{stm1} \ \texttt{od} \quad \textcircled{6} \\
+ \texttt{stm} = \texttt{for} \ \texttt{var} \ \texttt{from} \ e1 \ \texttt{to} \ e2 \ \texttt{do} \ \texttt{stm1} \ \texttt{od} \quad \textcircled{7} \\
+ \texttt{stm} = \texttt{repeat} \ \texttt{stm1} \ \texttt{until} \ \texttt{exp} . \quad \textcircled{8} \\

- Induction Cases: Sequential statement beginning with a basic statement

  + \texttt{stm} = \texttt{var} := \texttt{nat} ; \ \texttt{stm1} \quad \text{(with an assignment statement)} \quad \textcircled{9} \\
  + \texttt{stm} = \texttt{var} := \texttt{var1} ; \ \texttt{stm1} \quad \text{(with an assignment statement)} \quad \textcircled{10} \\
  + \texttt{stm} = \texttt{if} \ \texttt{exp} \ \texttt{then} \ \texttt{stm1} \ \texttt{else} \ \texttt{stm2} \ \texttt{fi} \ \texttt{stm1} \quad \text{(with an If statement)} \quad \textcircled{11} \\
  + \texttt{stm} = \texttt{while} \ \texttt{exp} \ \texttt{do} \ \texttt{stm1} \ \texttt{od} \ \texttt{stm1} \quad \text{(with a while statement)} \quad \textcircled{12} \\
  + \texttt{stm} = \texttt{for} \ \texttt{var} \ \texttt{from} \ e1 \ \texttt{to} \ e2 \ \texttt{do} \ \texttt{stm1} \ \texttt{od} \ \texttt{stm1} \quad \text{(with a for statement)} \quad \textcircled{13} \\
  + \texttt{stm} = \texttt{repeat} \ \texttt{stm1} \ \texttt{until} \ \texttt{exp} \ \texttt{stm1} \quad \text{(with a repeat statement)} \quad \textcircled{14} \\

The first proof score will be conducted is for a case that the input Environment is an error Environment.

open THEOREM
-- check
   red th(stm,env-err,env-halt) .
close

This case shows that when we consider an arbitrary statement with respect into an error Environment, the correctness of Compiler is satisfied without any assumptions. From now, we only do verifications for an arbitrary normal Environment (env-normal:EnvNormal).

A proof score for the empty statement also returns \textit{true} without any assumptions.

open THEOREM
-- assumptions
   eq stm = estm .
-- check
   red th(stm,env-normal,env-halt) .
close

Similarly, the case \textcircled{2} returns \textit{true} without any assumptions.

open THEOREM
-- assumptions
   eq stm = var := nat ; .
-- check
   red th(stm,env-normal,env-halt) .
close
In the case 3, after applied Case Splitting, the proof scores all return true (view example 3.2.1.1).

However, for the case 4, after applied Case Splitting many times, all proof scores could not return true. There are proof scores which return false. Therefore, some lemmas are required to validate those proof scores.

### 3.3.2 Finding Lemmas

Conducting lemmas is the most difficult part of verifications. It requires clear understanding of the specifications and the meaning of assumptions that added into proof scores. Occasionally, some candidates are conducted but proof scores could not validated, it means those are not lemmas. As a result, the experience in verifications is rather useful for finding proper lemmas. We will show one technique for conducting lemmas as follows.

#### Building Formalizations

When adding an assumption makes the proof score return false, the contradiction of this assumption could be used to predict the missing lemmas. However, we have to understand the meaning of it, adding with meaning of specifications which related to executions of the current theorem. A formalization is proper if it could be proved by proof scores and at that time, it’s considered as a lemma.

We illustrate this technique by considering the provement of the 4. Let’s take a proof score as follows,

```plaintext
open THEOREM
-- assumptions
  eq stm = var := exp ; .
  eq evalExp(exp,env-normal) = nat .
-- inv1
  eq nth((genForExp(exp) @ (store(var) | (quit | nil))),
         s(len(genForExp(exp)))) = quit .
-- th2
  eq (exec2(nth((genForExp(exp) @ (store(var) | (quit | nil))),0),
                        (genForExp(exp) @ (store(var) | (quit | nil))),0,empstk,env-normal)
                         = ((< var , nat >) | env-normal)) = false .
-- check
  red th(stm,env-normal,env-halt) .
close
```

When adding the 4th assumption, the proof score returns false. Looking at the 4th assumption, we know that the execution of a list of commands, genForExp(exp) @ (store(var) | (quit | nil)), is not equal with the (< var, nat > | env-normal). Based on the 2nd assumption, the value of exp is assumed as the nat. Therefore, the semantic of genForExp(exp) @ (store(var) | (quit | nil)) with respect into
env-normal is described by (((< var , nat >) | env-normal). From the analysis, if a formalization that be reduced into contradiction of the 4th assumption,

\[
eq \text{exec2}(\text{nth}((\text{genForExp}(\text{exp}) \oplus (\text{store}(\text{var}) \mid (\text{quit} \mid \text{nil}))),0), \\
(\text{genForExp}(\text{exp}) \oplus (\text{store}(\text{var}) \mid (\text{quit} \mid \text{nil}))),0,\text{empstk},\text{env-normal}) \\
= (((< \text{var} , \text{nat} >) | \text{env-normal})
\]

is a lemma, called th2, then the th2 will become false by the 4th assumption. It means the lemma th2 could be used to make this proof score return true.

However, the above equation is just a specific case of the lemma th2, it's not a proper formalization. Based on the specification of Virtual Machine, we recognize that (((< var , nat >) | env-normal) is reduced from the execution of a list of commands, (store(var) | (quit | nil)), with the (nat | empstk) stack. The form is as follows,

\[
\text{exec2}(\text{nth}((\text{store}(\text{var}) \mid (\text{quit} \mid \text{nil}))),0), \\
(\text{store}(\text{var}) \mid (\text{quit} \mid \text{nil})),0,(\text{nat} | \text{empstk}),\text{env-normal})
\]

We realize that the value of exp expression returned from evalExp(exp,env-normal) is put into the empstk stack after executed genForExp(exp) command. Intuitively, when executing expressions, the value returned by the Interpreter equals with the value done by the Virtual Machine. This's quite similar to the correctness of Compiler for expressions which already verified in previous works. With more analyses, we finally conduct the formalization of lemma th2 as follows,

\[
\text{eq th2}(\text{Exp:Exp},\text{V:Var},\text{Stk:StackNormal},\text{Env:EnvNormal}) = \\
\text{exec2}(\text{nth}((\text{genForExp}(\text{Exp}) \oplus (\text{store}(\text{V}) \mid \text{quit} \mid \text{nil}))),0), \\
(\text{genForExp}(\text{Exp}) \oplus (\text{store}(\text{V}) \mid \text{quit} \mid \text{nil})),0,\text{Stk},\text{Env}) \\
= \text{exec2}(\text{nth}((\text{store}(\text{V}) \mid \text{quit} \mid \text{nil})),0), \\
(\text{genForExp}(\text{Exp}) \oplus (\text{store}(\text{V}) \mid \text{quit} \mid \text{nil})),\text{len}(\text{genForExp}(\text{Exp})), \\
(\text{evalExp}(\text{Exp},\text{Env}) \mid \text{Stk}),\text{Env})) .
\]

The th2 shows that an Exp:Exp expression with respect into Env:EnvNormal is executed by the Virtual Machine (genForExp(Exp)) and by the Interpreter (evalExp(Exp,Env)) has the same value. However, the value is put into the Stk stack after generated by the exec2 of Virtual Machine.

Now, we use a prefix th_ (theorem) with a number to name lemmas which related with the Interpreter and Virtual Machine, and other lemmas will be named different. And we also conducted some other lemmas, called invariants, which are used frequently to prove theorems.

\[
\text{eq inv1}(\text{CL:CList},\text{CL1:CList},\text{PC:Nat}) = \\
\text{nth}(\text{CL} \oplus \text{CL1},\text{len}((\text{CL}) + \text{PC}) = \text{nth}(\text{CL1},\text{PC}) ) .
\]
\[
\text{eq inv2}(\text{Stm:Stm},\text{CL:CList}) = \\
\text{generator}(\text{Stm},\text{CL}) = (\text{CL} \oplus \text{generator}(\text{Stm},\text{nil})) .
\]
\[
\text{eq sd-lemma}(\text{N1:Nat},\text{N2:Nat}) = (\text{sd}((\text{N1} + \text{N2}),\text{N1}) = \text{N2}) .
\]
The \( \text{nth}(CL: \text{CList}, PC: \text{Nat}) \) returns a command at position \( PC \) of the \( CL \) list. The \text{inv1} lemma shows that a command at \( PC \) of the \( CL1 \) is same as a command at position \((\text{len}(CL) + PC)\) of the \((CL \circ CL1)\). The \text{inv2} shows the meaning of the \textit{generator} function for an abstract statement \( \text{Stm} \). And the \text{sd-lemma} means that \(((N1 + N2) - N1) = N2\) with \( N1, N2 \) are natural numbers.

The \text{th2} lemma is considered for a specific list of commands, \((\text{genForExp}(\text{Exp}) \circ (\text{store}(V) | \text{quit} | \text{nil}))\), We should conduct a theorem for an arbitrary list of commands.

**Generalizing Lemmas**

In the \text{th2} theorem, the \text{genForExp}(\text{Exp}) is at first of a certain list. Therefore, the lemma \text{th2} should be generalized in which the position of \text{genForExp}(\text{Exp}) is arbitrary. Finally, we conduct a theorem \text{th3}, a general form than \text{th2}, which is specified as follows,

\[
\text{eq}\ \text{th3}(\text{Exp}: \text{Exp}, CL1: \text{CList}, CL: \text{CList}, \text{Stk}: \text{StackNormal}, \text{Env}: \text{EnvNormal}) = \left(\begin{array}{l}
\text{exec2}(\text{nth}((\text{genForExp}(\text{Exp}) \circ CL), 0), \\
(\text{CL1} \circ \text{genForExp}(\text{Exp}) \circ CL), \text{len}(CL1), \text{Stk}, \text{Env}) = \text{exec2}(\text{nth}(CL, 0), \\
(\text{CL1} \circ \text{genForExp}(\text{Exp}) \circ CL), (\text{len}(CL1) + \text{len}(\text{genForExp}(\text{Exp}))), \\
(\text{evalExp}(\text{Exp}, \text{Env}) | \text{Stk}), \text{Env})
\end{array}\right).
\]

The input list, \((\text{CL1} \circ \text{genForExp}(\text{Exp}) \circ CL)\), is general. The \text{genForExp}(\text{Exp}) could be at everywhere inside the list, and its position is depended on the input \text{CL1} and \text{CL}. When the \text{th3} is validated, the \text{th2} will be straightforward by replacing \text{CL1} is \text{nil} and \text{CL} is \((\text{store}(V) | \text{quit} | \text{nil})).

In the \text{th3}, we have the semantic equivalence of expressions between two executions done by the Interpreter and Virtual Machine. The \text{evalExp}, an operator of the Interpreter, specifies the semantic of expression \text{Exp} with respect to \text{Env}. The \text{genForExp}, an operator of the Compiler, transforms \text{Exp} into a list of commands. This list then be specified by the \text{exec2}, an operator of the Virtual Machine. It’s clear that the \text{th3} theorem describes the correctness of Compiler for Minila expressions.

The verification of \text{th3} will be done by checking proof scores for each cases of expression by applied Case Splitting. The \text{Exp} might be \{nat, var, e1 ++ e2, e1 -- e2, e1 ** e2, e1 // e2, e1 % e2, e1 === e2, e1 !== e2, e1 >> e2, e1 << e2, e1 && e2, e1 || e2\}, with \text{e1} and \text{e2} are expressions.

Proof scores must be conducted for error cases of expressions, an example of those proof scores is as follows

\[
\text{open\ THEOREM}\ \\
\text{-- arbitrary values\ }
\text{ops e1 e2 : -> Exp .}\ \\
\text{op n1 : -> Nat .}\ \\
\text{-- assumptions}\ \\
\text{eq evalExp(e1, env-normal) = n1 .}
\]

\[
41
\]
eq evalExp(e2, env-normal) = 0.
....
-- check
red th3(e1 // e2, c1, c1, stk, env-normal).
close

When value of e2 is 0, the e1 // e2 becomes an error expression.

The th3 should cover the situation that the Exp:Exp with respect to Env:EnvNormal becomes an error expression. We have that when the Exp is an error expression, evalExp(Exp, Env) returns exp-err, then the left hand side of th3 equation returns env-err. Therefore, the (evalExp(Exp, Env) | Stk) becomes an error stack. This is the reason for definition of error stacks in STACK module,

\[
\text{op \_|\_ : ExpFull Stack -> StackErr.}
\]
\[
\text{ceq (E:ExpFull | Stk:Stack) = stkErr if (E = exp-err).}
\]
\[
\text{eq (V:Var | Stk:Stack) = stkErr.}
\]

Because the th3 is one of two base theorems for verifying the th, the formalization about correctness of Compiler, covering the error case of expression in th3 theorem is very meaningful. All proof scores for th3 need only the inv1 as hypothesis, no other theorems are required.

With the th3, proof scores for the assignment statement of theorem th is trivial. The th3 is also used for proving of other statements. However, for the conditional and iterative statements, we need more lemmas for verifications. When verifying iterative statements, one statement could be executed many times depending on its condition. And the input Environment will be changed after the statement loops again. We need some specifications that denotes the value of input Environment after many looping.

### 3.3.3 More Specifications

Now, we consider the input of theorem th is a while statement as the \(\circ\). The input \(\text{stm}\) is as follows,

\[
\text{stm = while exp do stm1 od.}
\]

with respect to the env-normal, the input Environment. For the case that the value of exp with env-normal equals 0,

\[
\text{eq evalExp(exp, env-normal) = 0.}
\]

execution of the \(\text{stm}\) will terminate and the th4 is trivial for this case.

For the case that the value of exp with env-normal is not equal 0, the execution executes a local \(\text{stm1}\), then the \(\text{stm}\) is looped again. The local \(\text{stm1}\) could be applied as the induction hypothesis and the eval(stm1, env-normal) might be used for semantics of \(\text{stm1}\) with respect to env-normal. The input Environment for the next execution of \(\text{stm}\) becomes eval(stm1, env-normal). In general, after many looping times, the Environment has a form as follows,
\[
\text{eval(stm1,eval(stm1,...(eval(stm1,env-normal))))).
\]

The general form of input Environment after \(N\) looping times will be specified by \texttt{create-env} operator in module \texttt{THEOREM-CONDITION} as follows,

\[
\text{mod THEOREM-CONDITION \{}
\]
\[
\text{...}
\]
\[
\text{op create-env : Stm Env Nat \to Env}
\]
\[
\text{eq create-env(Stm:Stm,EV:EnvNormal,0) \=} \text{EV}.
\]
\[
\text{eq create-env(Stm:Stm,EV:EnvNormal,s(N:Nat))}
\]
\[
\text{\quad \=} \text{eval(Stm,create-env(Stm,EV,N))}.
\]
\[
\text{...}
\]
\[
\text{\}.
\]

The \texttt{create-env(stm1,env-normal,N)} is a returned Environment after doing the \texttt{stm1} \(N\) times with respect to \texttt{env-normal}. It’s clear that the \texttt{create-env(stm1,env-normal,N)} might be an error or non-termination Environment, respectively \texttt{env-err} or \texttt{env-non-halt}.

Besides with the \texttt{create-env}, we define other functions that indicates the condition of expressions, exactly the evaluation of \texttt{exp} with respect to \texttt{env-normal} after executed the \texttt{Stm} \(N\) times. The specifications are also in the \texttt{THEOREM-CONDITION} module as follows,

\[
\text{op notZero : Exp Stm EnvNormal Nat \to Bool}.
\]
\[
\text{eq notZero(Exp:Exp,Stm:Stm,EV:EnvNormal,0) \=} \text{not (evalExp(Exp,EV) = 0 or evalExp(Exp,EV) = exp-err)}.
\]
\[
\text{eq notZero(Exp:Exp,Stm:Stm,EV:EnvNormal,s(N:Nat)) \=} 
\]
\[
\text{not (create-env(Stm,EV,s(N)) = env-non-halt or create-env(Stm,EV,s(N)) = env-err)}
\]
\[
\text{and (if (evalExp(Exp,create-env(Stm,EV,s(N)))) = 0 or evalExp(Exp,create-env(Stm,EV,s(N))) = exp-err) then false else notZero(Exp,Stm,EV,N) fi)}.
\]
\[
\text{--}
\]
\[
\text{op notZeroEqZero : Exp Stm EnvNormal Nat \to Bool}.
\]
\[
\text{eq notZeroEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,0) \=} \text{(evalExp(Exp,EV) = 0)}.
\]
\[
\text{eq notZeroEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,s(N:Nat)) \=} 
\]
\[
\text{not (create-env(Stm,EV,s(N)) = env-non-halt or create-env(Stm,EV,s(N)) = env-err)}
\]
\[
\text{and (notZero(Exp,Stm,EV,N) and evalExp(Exp,create-env(Stm,EV,s(N))) = 0)}.
\]
\[
\text{--}
\]
\[
\text{op existEqZero : Exp Stm EnvNormal Nat \to Bool}.
\]
\[
\text{eq existEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,N:Nat) \=} 
\]
\[
\text{not (create-env(Stm,EV,N) = env-non-halt or create-env(Stm,EV,N) = env-err)}
\]
\[
\text{and (evalExp(Exp,create-env(Stm,EV,N)) = 0)}.
\]
In mathematics, the above functions means that

- **notZero(Exp:Exp,Stm:Stm,EV:EnvNormal,N:Nat)**:
  \[ \forall k \in \text{Nat}, \ k \leq N, \ (\text{evalExp}(\text{create-env}(\text{Stm},\text{EV},k))) > 0 \]

- **notZeroEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,N:Nat)**:
  \[ \forall k \in \text{Nat}, \ k < N, \ (\text{evalExp}(\text{create-env}(\text{Stm},\text{EV},k))) > 0 \text{ and } (\text{evalExp}(\text{create-env}(\text{Stm},\text{EV},N)) = 0) \]

- **existEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,k:Nat)**:
  \[ \exists k \in \text{Nat}, \ (\text{evalExp}(\text{create-env}(\text{Stm},\text{EV},k))) = 0 \]

Intuitively, if **notZero(Exp,Stm,EV,N)** is true, the Stm will be loop N times, but we don’t know it terminates or loops again at the N + 1 times. If **notZeroEqZero(Exp,Stm,EV,N)** is true, the Stm certainly terminates after looping N - 1 times. The **existEqZero** is quite different with others, the counter k has ”exist” than ”for all” property. In general, if **existEqZero(Exp,Stm,EV,N)** is true, the Stm definitely terminates, however we don’t know exactly when the termination happens (at least N times). Specifications of these functions are too complicated because we must limited **exp-err** value of the **evalExp** function, and env-non-halt, env-err of the **create-env**. The specifications must be sured that all error or non-termination situations are not occured.

We have a lemmas about the relation between the **notZeroEqZero** and **existEqZero** as follows,

\[ N \in \text{Nat}, \ \exists M \leq N, \ \text{existEqZero}(\text{Exp},\text{Stm},\text{EV},\text{N}) \rightarrow \text{notZeroEqZero}(\text{Exp},\text{Stm},\text{EV},\text{M}) \]

  \[ \text{existEqZero}(\text{Exp},\text{Stm},\text{EV},\text{N}) \text{ implies } (M < s(N) \text{ implies notZeroEqZero}(\text{Exp},\text{Stm},\text{EV},\text{M})) \] .

Because the **notZeroEqZero-lemma** requires the existing of value M, we apply ”witness” technique for provement. This technique claims that we just find a counterexample of value M and adding into proof scores. If proof scores are success, then the lemma is satisfied. Here is an example of proof scores using the witness method.

\[ \rightarrow 1.i) \]
open THEOREM
  op m : \rightarrow \text{Nat} .
-- assumptions
-- this is a witness .
  eq m = 0 .
-- check
  red notZeroEqZero-lemma(exp,stm,env-normal,0,m) .
close
In Base Case of the notZeroEqZero-lemma, $N = 0$, the existEqZero($Exp, Stm, EV, N$) becomes evalExp($Exp, EV$) = 0. Because of the existing of $M$, if we choose $m = 0$, the ($M < s(N)$) becomes true, and the notZeroEqZero($Exp, Stm, EV, M$) becomes evalExp($Exp, EV$) = 0. The proof score is success and $m = 0$ is considered as a witness.

The proof scores for notZeroEqZero-lemma require another lemma, called notZero-lemma. It’s defined as follows,

$$N \in \text{Nat}, \exists M \leq N, \quad \neg \text{notZero}(Exp,Stm,Ev,N) \iff \text{evalExp}(Exp, create-env(Stm,EV,M)) = 0.$$  

And this lemma is also proved by applied the witness technique.

Intuitively, execution of a while statement will terminate if the existEqZero becomes true with existing $N$. This is contrary to the non-termination situation when the while statement loops for ever. On other words, if the notZero is true for an arbitrary number $N$, the non-termination situation happens. There is a contradiction between notZero and existEqZero which is specified as follows

- $\exists N \in \text{Nat}, \text{existEqZero}(Exp,Stm,Ev,N) = true$
- $\forall N \in \text{Nat}, \text{notZero}(Exp,Stm,Ev,N) = true$

The contradiction tells that a while statement could be splitted into two cases based on the existEqZero and notZero. We see in detail at the next section.

When the notZero is true for an arbitrary number $N$, the while statement is infinite loops. This could be defined as follows,

$$\text{eq definition}(Exp:Exp,Stm:Stm,Ev:EnvNormal,N:Nat) =$$
$$\text{notZero}(Exp,Stm,Ev,N) \implies \text{interpret(while Exp do Stm od,Ev) = env-non-halt}.$$  

We didn’t specify the env-non-halt in specifications of the Interpreter or Virtual Machine. As a result, the proof scores for non-termination situations might not be proved because the missing information of env-non-halt. However, with this definition, the env-non-halt state could be known. We notice that the definition is just for the Interpreter execution because the notZero condition could be determined. For the Virtual Machine, the execution for sequence of commands only terminate when it meets the quit command which usually at the end of the sequence. The sequence of commands without the quit might terminate or not. Therefore, non-termination execution could not be determined for a sequence of commands. On other words, the condition of non-termination
of the Minila statement can not be preserved on corresponding sequence of commands of the Machine language.

Even though having more specifications of the `existEqZero` or `notZero`, proof scores of the th for conditional and iterative statements could not be success. We need another lemma that describes a theorem for a general statement.

### 3.3.4 Theorem th4

Similar to theorem th3, the th4 considers list of commands for a general statement, \( Stm:Stm \). Based on the ideas of th3, we finally form a formalization of the th4 as follows,

\[
eq th4(Stm:Stm, CL1:CList, CL:CList, Stk:StackNormal, EV:EnvNormal) = \\
\text{not } (\text{eval}(Stm,EV) = \text{env-non-halt}) \implies \\
(\text{exec2}(\text{nth}((\text{generator}(Stm,nil) \circ CL),0), \\
(\text{CL1} \circ \text{generator}(Stm,nil) \circ CL),\text{len}(CL1),Stk,EV) \\
= \text{exec2}(\text{nth}(CL,0), \\
(\text{CL1} \circ \text{generator}(Stm,nil) \circ CL), \\
\text{len}(\text{generator}(Stm,nil)) + \text{len}(CL1),Stk,\text{eval}(Stm,EV))) .
\]

The structure of th4 is quite similar to the th3. The th4 considers a general list, \( (\text{CL1} \circ \text{generator}(Stm,nil) \circ CL) \), in which the position of \( \text{generator}(Stm,nil) \) is arbitrary. In the th4, the Interpreter specifies semantics of statement \( Stm \) with respect to \( EV \) by executing \( \text{eval}(Stm,EV) \). And the Virtual Machine executes the list of commands that generated by \( \text{generator}(Stm,nil) \) of the Compiler. This execution is done by the \text{exec2} operator.

However, for an infinite loop statement, the execution will not terminate, it means that the left hand side of th4 equation will not return an Environment even if applied Case Splitting many times. Similarly, the \( \text{eval}(Stm,EV) \) could not return a specific Environment, and the right hand side of th4 equation might be not determined. In this situation, comparing the value of both sides is impossible. Therefore, the th4 should not cover infinite loop statements, or non-termination situations. These situations are denoted by the \text{env-non-halt} of \text{EnvNonHalt} sort in the Environment Hierarchy at 2.2. And in theorem th4, the exception for infinite loop statements is expressed by following condition,

\[
\text{not } (\text{eval}(Stm,EV) = \text{env-non-halt})
\]

it means when \( Stm \) is an infinite loop statement, this condition becomes \text{false} and the th4 is trivial.

When building the theorem th4, we tried not to include the above condition, but the prove ment was not success. When applied Case Splitting for all cases of a while statement, the execution of the right hand side of th4 equation could be determined as \text{env-non-halt} for infinite statements. However, we realized that the execution of left hand side will not stop if continuing apply Case Splitting. Finally, the th4 equation could not be verified. We already know that the \text{exec2}, is execution of the Virtual Machine
for sequence of commands. Moreover, we could not determined when the sequence of
commands corresponding to a while statement terminate. This might be an explanation
for requirement of the condition of theorem \textbf{th4}.

In the \textbf{th4}, the value of \texttt{eval(Stm,EV)} is replaced for the value is generated by the
\texttt{exec2} of Virtual Machine. It shows that semantics of a statement \texttt{Stm} that are generated
by the Interpreter and the Virtual Machine are equivalent. However, the \textbf{th4} also
confirms that the input \texttt{Stm} is limited with finite loop statements by the termination
condition. The correctness defined by the \textbf{th4} is the semantic equivalence between two
Environments returned by the Interpreter and Virtual Machine. This show that the basis
of definition 3.1.2 is still remained.

Formalizing the \textbf{th4} is an important step for proving the \textbf{th}, the main object of this
research. Intuitively, the \textbf{th4} is a specific case of the \textbf{th2}. The accepted statements of
\textbf{th4} are just finite statements, that is a subset of Minila statement. The condition of
\textbf{th4} is one reason of that the \texttt{iff} relation is not satisfied for the \textbf{th}. By applied Case
Splitting for non termination situations, only (\texttt{interpret(Stm,Env1) = Env2}) becomes
\texttt{false} and the \textbf{th} is trivial with the \texttt{implies} relation.

We use induction on the input statement to prove the theorem \textbf{th4}. The \texttt{Stm:Stm} for
Basic Cases and Induction Cases applies by Case Splitting like doing with the \textbf{th} at 3.3.1.
Proof scores for empty, assignment and if statements are success without many problems.

\textbf{The significance of Environment Hiearchy}

In practice, a statement could not be determined as an infinite loop statement at begin-
ing. When doing verifications, even though a statement is infinite loop, we must conduct
proof scores to make sure that the correctness property is statisfied with this statement.
Obviously, a general statement might belong the finite group or infinite group. On other
words, there’re only two cases, termination and non-termination. Applied Case Split-
ting for the statement object requires hierachy of Environment which is desiged like the
Environment Hiearachy at 2.2. With the Environment Hiearachy, a statement is finite or
infinite loop statement is not important. The termination and non-termination situations
are just cases which will be verified seperately by proof scores. When we add assumptions
about the semantic of a general statement, proof scores are required for three cases based
on the Environment Hiearachy,

\begin{align*}
\text{eq eval(stm,env-normal) = env1-normal} \\
\text{eq eval(stm,env-normal) = env-err} \\
\text{eq eval(stm,env-normal) = env-non-halt}
\end{align*}

When proof scores for three cases are all satisfied, it means that we covered all statement
of Minila language.

Among the Minila statements, some kinds of statements is conducted form sub-statements
such as the if statement,

\begin{align*}
\text{stm = if exp then stm1 else stm2 fi}
\end{align*}
The **stm** is based on the **stm1** and **stm2**. For execution of the **stm**, the returned Environment from the **stm1** or **stm2** is necessary. However, the **stm1** or **stm2** are just the general statements which maybe finite or infinite statements. With the Environment Hierachy, three values of Case Splitting could range all statements that the **stm1** or **stm2** might be. These cases are used as induction hypotheses for provement of the **stm**. In general, the Environment Hierachy is very important for verifications not only of the **th** but also of the **th4**, and other theorems.

**Verification of the While statement**

Verifications of the While (iterative) statement is more difficult than other statements. Applying Case Splitting for conducting proof scores of the **th4** is more complicate than for other theorems. The finite or infinite of While statement are splitted into two cases which based on the assumptions of **existEqZero** and **notZero**. From the source code of proof scores we see that there are three assumptions for two splitted cases,

\[
\begin{align*}
\text{eq } & \text{existEqZero(exp,stm1,env-normal,0) = true} . \\
\text{eq } & \text{existEqZero(exp,stm1,env-normal,s(nat)) = true} . \\
\text{eq } & \text{notZero(exp,stm1,env-normal,k) = true} . \\
\end{align*}
\]

The **existEqZero** should be considered with zero and non-zero values. For zero case of the **existEqZero**, the equivalent assumptions,

\[
\text{eq evalExp(exp,env-normal) = 0} .
\]

is replaced. And we might replace an assumption by group of equivalent assumptions, such as

\[
\text{eq notZeroEqZero(exp,stm1,env-normal,s(nat)) = true} .
\]

is replaced by three assumptions

\[
\begin{align*}
\text{eq } & \text{eval(stm1,create-env(stm1,env-normal,nat)) = env1-normal} . \\
\text{eq } & \text{notZero(exp,stm1,env-normal,nat) = true} . \\
\text{eq } & \text{evalExp(exp,env1-normal) = 0} . \\
\end{align*}
\]

Finally, for the infinite case of the While statement, described by the **notZero** assumption, the definition of **env-non-halt**, 

\[
\text{red definition(exp,stm1,env-normal,k) implies th4(stm,c11,cl,stk,env-normal)} .
\]

is used for provement of the **th4**. This definition is needed because it’s a unique information about specification of the **env-non-halt**.
The relation \( \text{iff} \) for theorem \( \text{th1} \), but \( \text{implies} \) for the \( \text{th} \)

We could realize that the \( \text{th4} \) is a general form of the theorem \( \text{th1} \) as follows,

\[
eq \text{th1}(\text{Stm}:\text{Stm}, \text{EV}:\text{EnvNormal}, \text{EV2}:\text{EnvHalt}) = \\
\neg (\text{interpret}(\text{Stm}, \text{EV}) = \text{env-non-halt}) \\
\quad \text{implies} \quad (\text{interpret}(\text{Stm}, \text{EV}) = \text{EV2} \text{ iff } \text{vm}(\text{compile}(\text{Stm}), \text{EV}) = \text{EV2}) .
\]

it’s exactly a specific case of the theorem \( \text{th} \) with the condition accepts only finite loop statements, \( \neg (\text{interpret}(\text{Stm}, \text{EV}) = \text{env-non-halt}) \).

We could see that the \( \text{th4} \) (or \( \text{th1} \)) satisfies \( \text{iff} \) relation, however the \( \text{iff} \) could not validated in theorem \( \text{th} \) when the non-termination condition is removed. The \( \text{th} \) only satisfies the weaker \( \text{implies} \) relation.

\[
eq \text{th}(\text{Stm}:\text{Stm}, \text{Env1}:\text{EnvHalt}, \text{Env2}:\text{EnvHalt}) = \\
(\text{interpret}(\text{Stm}, \text{Env1}) = \text{Env2}) \text{ implies } (\text{vm}(\text{compile}(\text{Stm}), \text{Env1}) = \text{Env2}) .
\]

Looking at inside the proof scores of the \( \text{th4} \) and the \( \text{th} \) for find the reason, we recognize that when applied Case Splitting into \text{existEqZero} and \text{notZero} cases of the while statement, the \text{notZero} indicate the non-termination situations. Based on specifications, with \text{notZero} assumption, we could not determined \text{interpret}(\text{Stm}, \text{EV}) and \text{vm}(\text{compile}(\text{Stm}), \text{EV}) are returned Environment or not because the Case Splitting will repeat forever. That why we need the definition,

\[
eq \text{definition}(\text{Exp}:\text{Exp}, \text{Stm}:\text{Stm}, \text{EV}:\text{EnvNormal}, \text{N}:\text{Nat}) = \\
\text{notZero}(\text{Exp}, \text{Stm}, \text{EV}, \text{N}) \\
\quad \text{implies} \quad (\text{interpret}(\text{while Exp do Stm od}, \text{EV}) = \text{env-non-halt}) .
\]

mentioned in above section. This is a definition for the meaning of while statement. With this definition as hypothesis, the \( \text{th1} \) (or \( \text{th4} \)) is validated. However the \( \text{th} \) could not validate with the \( \text{iff} \) relation because the rightside, \( \text{vm}(\text{compile}(\text{Stm}), \text{Env1}) = \text{Env2} \), of \( \text{th} \) could become \text{false}. There is no definition for the Virtual Machine in this case. It means the \( \text{th} \) only satisfied with the \( \text{implies} \) relation. In our opinion, the definition for while statement could not preserved when it’s transformed by Compiler, and this might be reasonable answer for the different relation between the \( \text{th1} \) and \( \text{th} \).

When proof scores of \( \text{th4} \) are all validated, the \( \text{th4} \) could be used to verify other lemmas. With theorem \( \text{th4} \), the proof scores of the \( \text{th} \) are trivial. Besides with the \( \text{th2} \), \( \text{th3} \), and \( \text{th4} \) there are other theorems for verification of the \( \text{th} \) that is in detail at the source code.

The complete verifications of the \( \text{th} \) show that the correctness of Compiler for Minila language is validated. It is the main object of this research. In summary, other theorems for verifications of the correctness in this document could be found at the source code.

The summary diagram that shows relation between theorems and lemmas used in this document is as follows,
3.3.5 The source code

In this research, specifications is all stored in "minila.mod" file, proof scores for each theorem are all put into a separate file. For an example, all proof scores for theorem th is in file "th.mod". The source code (specifications and proof scores) is compressed in zip file, "minila.zip".

The CafeOBJ version 2009 Feb 23 is used for this research. When execute "in minila.mod" to run the specification, there output some warnings. We already realized those warnings, however it's not effect on the verification of proof scores. We let an warning

```
[Warning]: axiom : exec2(mod,CL:CList,PC:Nat,(N2:Nat | (N1:Nat | Stk:StackNormal)),EV:EnvNormal) = (if (not (N2 = 0)) then
  exec(CL,(PC + s(0)),((N1 rem N2) | Stk),EV) else env-err fi)
contains error operators............* done.
```

In specifications, the rem operation is defined with N2 belongs NzNat. However, in above equation, we could see that the not (N2 = 0) was already added as a condition before doing the rem. Therefore, if N2 equals 0, the rem could not be run. The CafeOBJ language could not detected this meaning in the equation, hence it alerts a warning. The same things happen with other equations in the specifications. After run "in minila.mod" command, we could do verification by executing proof scores for each theorem. For an example, "in th.mod" for verifying the main theorem th. All proof scores of all theorems return true means that the correctness of Compiler for Minila statements is verified.
Chapter 4

Conclusion

4.1 Summary

In this research, we completely specified the Minila language, the Compiler and other related components such as the Environment, Interpreter or Virtual Machine. We also considered problems of specification such as the Error Handling and the sort hierarchy, especially the Environment Hierarchy. Furthermore, the correctness of Compiler for Minila language were defined and formalized as the theorem $th$, it’s the extension and improvement of the definition in previous works. The proof scores for theorem $th$ and the others were all success. It means the correctness of Compiler for Minila language were already verified by using CafeOBJ language. The success verification for Minila language is an important achievement for verifying the correctness of compiler for an imperative programming language in general.

4.1.1 Completed Specifications

Specifications of the Minila and others are all stored in the ”minila.mod” file. In total, there are about 850 lines for specifications. The specifications also contains the Error Handling and Environment Hierarchy.

Error Handling

When defining a programming language, exception (error) situations should be considered. Therefore, the Error Handling is a important part of specifications. In this document, we already introduced 3-step error detection method, which might be a technique for solving error problems. This technique could be used when specify other programming language in general.

Environment Hierarchy

Environment Hierarchy is one possible method when specify and verify the statement of Minila language, especially for the infinite looping statement. With this hierarchy, the finite
and infinite statements are determined as two cases of Case Splitting that corresponding with the terminated Environment and non-termination Environment. Proof scores are just conducted for each cases like normal situations of Case Splitting. And in this document, Environment Hierachy is very useful for verification of the correctness of Compiler.

4.1.2 Completed Verfications

Verifying the correctness of Compiler for Minila language is expressed in proof scores. In this documents, proof scores were already verified. We have more than 10,000 lines of proof scores in total. The th3 and th4 are two theorems that should be noticed. Roughly, the th3 covers the correctness of Compiler for expressions that were verified in previous works. And the th4 showed the semantic equivalence when executed by the Interpreter and Virtual Machine of finite statements. The th4 specifies the correctness of Compiler for termination situations. Even though the th only satisfied the "implies" than "iff" relation, the completed verification of th claims that the Compiler of Minila language satisfies the correctness property. On other words, the correctness of Compiler for an imperative programming language is validated in this research with Minila as an given imperative language.

4.2 Related Works

The correctness of compilers was first considered in [11] but it focused only on the arithmeic expressions of programming language. Thereafter, there are many efforts for verifying the correctness of compilers for whole programming languages. Many potential approaches are proposed such as denotational semantics [13][18], refinement calculus [14], structural operational semantics [5].

Our research is quite close to the "An Algebraic Approach to Compiler Design" [1][2]. In that document, a reasoning (the source) language and a compiler were defined by using the algebraic approach. The source language also considered complex statement structures such as the Arbort, Miracle, Nondeterminism etc. The correctness property was verified by the OBJ3, an implementation of the OBJ specification language [10].

Many projects demonstrated on a concrete implementation of the source and target language. They focused on the realistic aspect of programming language and the Compiler closed to the implementation for concrete language.

Leroy [19] verified a compiler from Cminor (a C-like imperative language) to Power PC assembly code, using the Cop proof assistant both for programming the compiler and for proving its correctness.


In the context of the German Verisoft initiative, Leinenbach [4] and Strecker [16] formally verified a compiler for a C-like language called C0 down to DLX assembly code.
using the Isabelle/HOL proof assistant. This compiler appears to work in a single pass and to generate unoptimized code.

The Verifix project [7] proposed the constructions of mathematically correct compilers. The only part that led to a machine-checked proof was the formal verification in PVS of a compiler for a subset of Common Lisp to Transputer code.

4.3 Future Research

Because of the short time of Master’s research, this document only considered the Minila language with essential, simple features. In future, we shall concentrate on complex structures of imperative programming language such as procedure calls or pointer. Moreover, we study about the code optimization and more on mechanisation of the algebraic denotational semantics.
Bibliography


