Title	Verifying the Correctness of Compiler for an Imperative Programming Language
Author(s)	Trinh, Bao Ngoc Quoc
Citation	
Issue Date	2011-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/9639
Rights	
Description	Supervisor: Professor Futatsugi Kokichi, School of Information, Master Course

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 exress my deep sense of gratitude to my suppervisor, 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 cource.

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

List of Figures

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 languague 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 $\quad \_\,;\,\_ \;:\; Store\;\; Program \;\rightarrow\; Store$

The ":" notation separates the syntax of operation which declared by "_ ; _" and the rest of fomulas. In the form of the "_ ; _" operation, the underbars are place holders that indicate where input entities of algebra *Store* and *Program* go, respectively. The "→" 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 languague.

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.

```
mod! BASIC-NAT {
```

```
  [Zero NzNat < Nat]
  op 0 : -> Zero       {constr}
  op s : Nat -> NzNat  {constr}
  ...
```

In this example, `Zero` and `NzNat` are sorts which denote zero number and non zero number, respectly. Intuitively, zero or non zero numbers are all considered as general natural numbers, therefore `Zero` and `NzNat` are subsorts of the `Nat`, generall natural numbers, and the **subsort** relation is denoted by the "<" symbol. This is the **order sorted signatures** feature of CafeOBJ language.

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

```
    op s : Nat -> NzNat  {constr}
```

where CafeOBJ keyword "`op`" indicates that the syntax of an operation is being declared, with the output sort after the "→" and the input sorts listed between the ":" and the "→". The `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,

```
 op _=_ : Nat Nat -> Bool
 op if_then_else_fi : Exp Stm Stm -> Stm
```

We will explain in detail these operators in next sections.

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

Example 1.1.3.2 A basic natural number only has data sorts and comparation operations. It's specified by a module as follows,

```
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 .
```

```
  eq 0 < s(N:Nat) = true .
  eq s(M:Nat) < 0 = false .
  eq s(M:Nat) < s(N:Nat) = M < N .
}
```

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

```
  eq (0 = s(N:Nat)) = false .
  eq (N:Nat = N) = true .
  eq (s(M:Nat) = s(N:Nat)) = (M = N) .
```

specify the semantics of equations between two numbers.

Using CafeOBJ, an imperative language is constructed from many components, each component is a 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 appoach 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 ψ

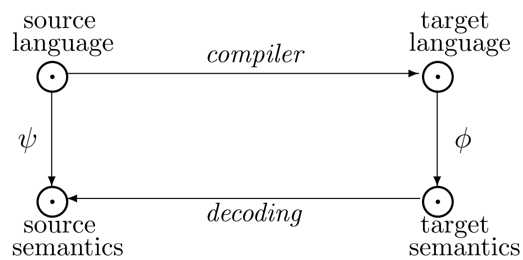

Figure 1.1: The Morris Correctness Diagram

and ϕ used to specify the algebraic semantics of the source and target language, respectly 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 an 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 ψ and ϕ 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 ψ and ϕ 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

+ $estm$ (empty statement)

+ $v(i) := E\ ;$ (assignment statement)

+ $if\ E\ then\ S_a\ else\ S_b\ fi$ (conditional statement)

+ $while\ E\ do\ S\ od$ (iterative statement)

+ $for\ v(i)\ from\ E_a\ to\ E_b\ do\ S\ od$ (iterative statement)

+ $repeat\ S\ until\ E$ (iterative statement)

In above structures, $v(i)$ is a varible 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

```
v(0) := 0 ; v(s(0)) := s(0) ; while v(0) << s(s(s(0))) do
                                     v(0) := v(0) ++ s(0) ;
                                     v(s(0)) := v(s(0)) ** v(0) ;
                                  od .
```

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(s(0)) , s(s(s(s(s(s(0)))))) >) | ((< v(0) , s(s(s(0))) >) |
((< v(s(0)) , s(s(0)) >) | ((< v(0) , s(s(0)) >) |
((< v(s(0)) , s(0) >) | ((< v(0) , s(0) >) |
((< v(s(0)) , s(0) >) | ((< v(0) , 0 >) | 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,

```
(push(0) | (store(v(0)) | (push(s(0)) | (store(v(s(0))) | (load(v(0)) |
(push(s(s(s(0)))) | (lessThan | (jumpOnCond(s(s(0))) |
(jump(s(s(s(s(s(s(s(s(s(s(0))))))))))) | (load(v(0)) | (push(s(0)) | (add |
(store(v(0)) | (load(v(s(0))) | (load(v(0)) | (multiply | (store(v(s(0))) |
(bjump(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))) |
(quit | nil)))))))))))))))))))
```

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,

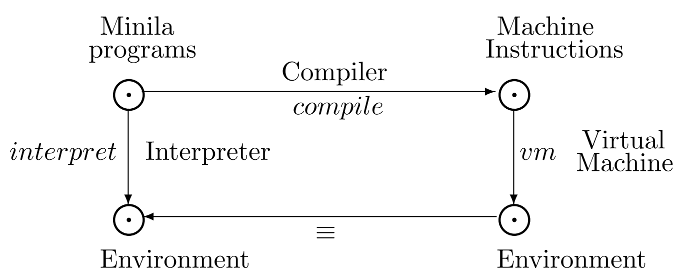

Figure 1.2: The Correctness of Minila's Compiler Diagram

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 signification role in the computer science. Compiler verification is big challenge of high-assurance software. In this document, we attempt to verify the correctness of 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 successfull 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,

```
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:Nat) * N:Nat = N + (M * N) .
-- sd
  eq sd(0,N:Nat) = N .
  eq sd(s(M:Nat),0) = s(M) .
  eq sd(s(M:Nat),s(N:Nat)) = sd(M,N) .
-- quo
  eq M:Nat quo s(N:Nat) = (if M < s(N) then 0 else s(sd(M,s(N)) quo s(N)) fi) .
-- rem
  eq M:Nat rem s(N:Nat) = (if M < s(N) then M else sd(M,s(N)) rem s(N) 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 symmentric 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 exeptions 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
  --
  op _=_ : ExpFull ExpFull -> Bool {comm} .
  eq (E:ExpFull = E) = true .
  eq (E:Exp = exp-err) = false .
```

```
-- _++_
  op _++_ : Exp Exp -> Exp           {l-assoc prec: 30}
-- _--_
  op _--_ : Exp Exp -> Exp           {l-assoc prec: 30}
-- _**_
  op _**_ : Exp Exp -> Exp          {l-assoc prec: 29}
-- _//_
  op _//_ : Exp Exp -> Exp           {l-assoc prec: 29}
-- _%%_
  op _%%_ : Exp Exp -> Exp           {l-assoc prec: 29}
-- _===_
  op _===_ : Exp Exp -> Exp           {l-assoc prec: 31}
-- _!==_
  op _!==_ : Exp Exp -> Exp           {l-assoc prec: 31}
-- _<<_
  op _<<_ : Exp Exp -> Exp            {l-assoc prec: 31}
-- _>>_
  op _>>_ : Exp Exp -> Exp            {l-assoc prec: 31}
-- _&&_
  op _&&_ : Exp Exp -> Exp            {l-assoc prec: 32}
-- _||_
  op _||_ : Exp Exp -> Exp            {l-assoc prec: 32}
}
```

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 formilized 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(EXP)
  [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 .
  eq ((X:Elt.M | L:ListNormal) = (Y:Elt.M | L1:ListNormal)) =
        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 .
-- _@_
  eq 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,

```
mod* EQTRIV {
  [Elt]
  op _=_ : Elt Elt -> Bool .
}
```

The sort hierachy of `LIST` is quite complicated

```
    [Nil NnList < ListNormal]
    [ListNormal ListErr < ListHalt]
    [ListHalt ListNonHalt < List]
```

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

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

The general `LIST` supports some operations, the `=` for equations, the `@` for combinations of lists, the `nth` outputs the first element of a given list, and the `len` calculates the number elements of a given list.

A view is a mapping from the `EQTRIV` into the `ENTRY` module

```
  view EQTRIV2ENTRY from EQTRIV to ENTRY {
    sort Elt     -> Entry,
    op (_=_)  -> (_=_)
  }
```

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

```
mod! ENV principal-sort Env {
  pr(PNAT)  pr(EXP)
  pr(LIST(EQTRIV2ENTRY) * {
                  sort Nil     -> Empty,
                  sort List    -> Env,
                  sort ListNonHalt -> EnvNonHalt,
                  sort ListHalt -> EnvHalt,
                  sort ListNormal -> EnvNormal,
                  sort ListErr -> EnvErr,
                  op nil       -> empty,
                  op list-err   -> env-err,
                  op list-non-halt -> 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:Exp,env-err) = env-err .
  eq update(V:Var,Exp:Exp,env-non-halt) = env-non-halt .
  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 $\Longleftrightarrow$ 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)
```

```
                        @ genForExp(E)
                        @ (jumpOnCond(s(s(0))) |
                           bjump(len(genForExp(E)) + len(generator(S1,nil)) + s(0))
                                  | nil)) .

  -- for V from E1 to E2 do S1 od
  eq generator(for V:Var from E1:Exp to E2:Exp do S1:Stm od Stm:Stm,CL:CList)
     = generator(Stm, CL @ genForExp(E1)
                        @ (store(V) | load(V) | nil)
                        @ genForExp(E2)
                        @ (greaterThan |
                           jumpOnCond(len(generator(S1,nil)) + s(s(s(s(s(s(0)))))) | nil
                        @ generator(S1,nil)
                        @ (load(V) | push(s(0)) | add | store(V) |
                           bjump(len(genForExp(E2)) +
                               len(generator(S1,nil)) + s(s(s(s(s(s(s(0))))))) | nil)
  --
  eq genForExp(V:Var) = load(V) | nil .
  eq genForExp(N:Nat) = push(N) | nil .
  eq genForExp(E1:Exp ** E2:Exp) = genForExp(E1) @ genForExp(E2) @ (multiply | nil) .
  eq genForExp(E1:Exp // E2:Exp) = genForExp(E1) @ genForExp(E2) @ (divide | nil) .
  eq genForExp(E1:Exp %% E2:Exp) = genForExp(E1) @ genForExp(E2) @ (mod | nil) .
  eq genForExp(E1:Exp ++ E2:Exp) = genForExp(E1) @ genForExp(E2) @ (add | nil) .
  eq genForExp(E1:Exp -- E2:Exp) = genForExp(E1) @ genForExp(E2) @ (minus | nil) .
  eq genForExp(E1:Exp << E2:Exp) = genForExp(E1) @ genForExp(E2) @ (lessThan | nil) .
  eq genForExp(E1:Exp >> E2:Exp) = genForExp(E1) @ genForExp(E2) @ (greaterThan | nil) .
  eq genForExp(E1:Exp === E2:Exp) = genForExp(E1) @ genForExp(E2) @ (equal | nil) .
  eq genForExp(E1:Exp !== E2:Exp) = genForExp(E1) @ genForExp(E2) @ (notEqual | nil) .
  eq genForExp(E1:Exp && E2:Exp) = genForExp(E1) @ genForExp(E2) @ (and | nil) .
  eq genForExp(E1:Exp || E2:Exp) = genForExp(E1) @ genForExp(E2) @ (or | 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.

```
op compile : Stm -> 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,

```
eq compile(S:Stm) = generator(S,nil) @ (quit | 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 tranform 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 Hierachy

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

```
[Exp ExpErr < ExpFull]
op exp-err : -> 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

```
eq (E:Exp = exp-err) = false .
```

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

The sort hierachy 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 hierachies.

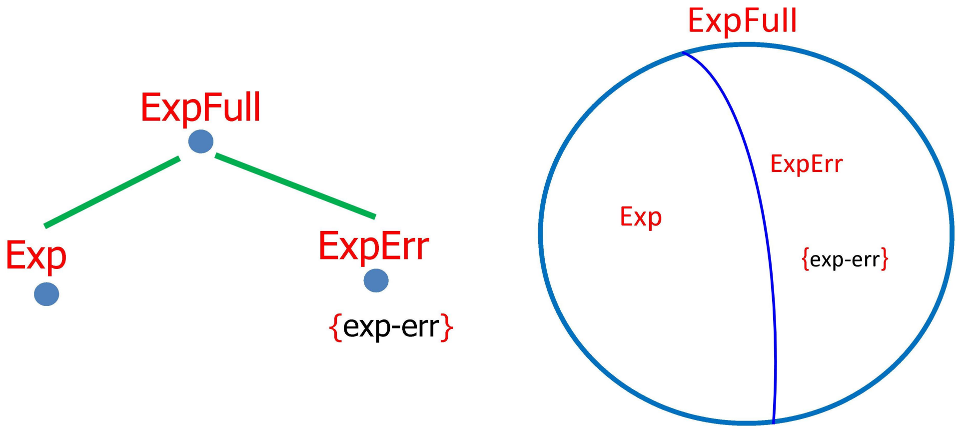

Figure 2.1: Sort hierachy for the Expression

2.3 Environment Hierachy

Now, we consider another sort hierachy used for the Environment, called *Environment Hierachy*. It's more complicated than sort hierachy for the expression, and already defined in the `LIST` module but the sorts are renamed in the `ENV` module. The Environment Hierachy is

```
[EnvNormal EnvErr < EnvHalt]
[EnvHalt EnvNonHalt < Env]
```

The sort relation of Environment Hierachy is illustrated as follows

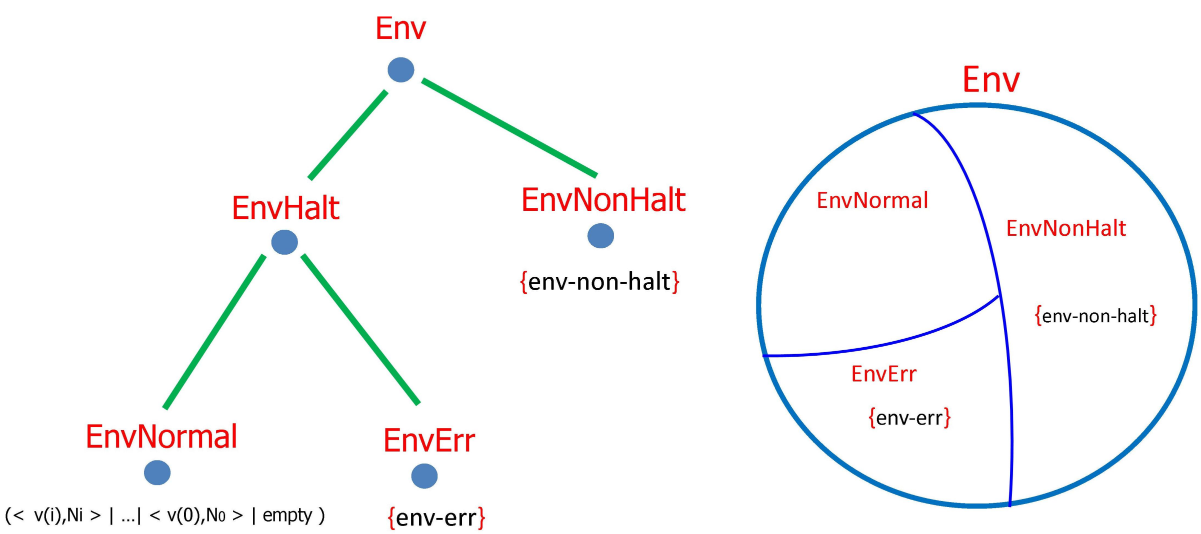

Figure 2.2: The Environment Hierachy

When the execution of a `Stm` terminates without errors, the returned Environment is a list of pairs of variables and values, which has a well-form oganization. All well-form

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 excution 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 aldready defined in the `LIST` module by

```
eq (L:ListHalt = list-non-halt) = false .
```

In general, we have three dedicated sorts: `EnvNormal`, `EnvErr` and `EnvNonHalt`. Each of them describles 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 emphazise 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 hierachy 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 hierachy for Environment as follows.

This hierachy explained another aspect of the sort hierachy for Environment. We will have the global view about programs of Minila language, as well as deep understand for sort hierachy 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 varibles 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 Environement, which is specified in `ENV` module at 2.2.2

```
eq lookup(V:Var,empty) = exp-err .
```

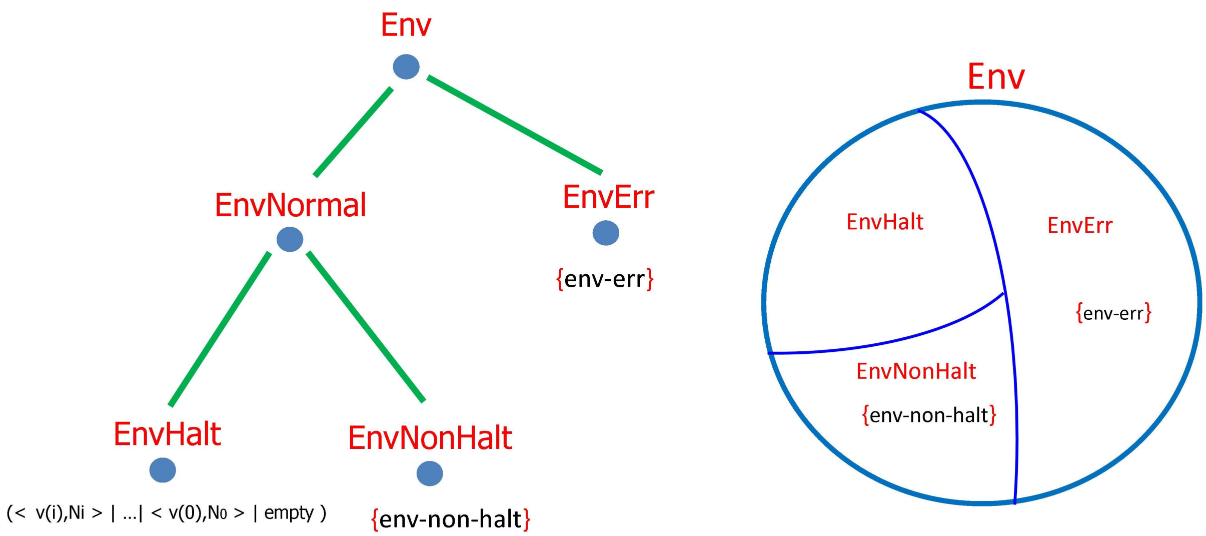

Figure 2.3: The alternative sort hierachy for Environment

```
  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 hierachy 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(N:Nat,EV:EnvNormal) = N .
  eq evalExp(V:Var,EV:EnvNormal) =
       if (lookup(V,EV) = exp-err) then exp-err else lookup(V,EV) fi .
  ...
```

```
  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 varible `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*, respectly. 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,

```
ceq eval(S:Stm,EV:Env) = EV if (EV = env-non-halt or EV = env-err) .
--
eq eval(while E:Exp do S1:Stm od S:Stm,EV:EnvNormal) =
      eval(S,evalWhile(E,S1,EV)) .
--
eq evalWhile(E:Exp,S1:Stm,EV:Env)
   = if (EV = env-non-halt)
        then env-non-halt
        else if (EV = env-err or evalExp(E,EV) = exp-err)
               then env-err
               else if not(evalExp(E,EV) = 0)
                       then evalWhile(E,S1,eval(S1,EV))
                       else eval(estm,EV)   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 executtion 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(EXP)
  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(EXP)  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 hierachy and error constants.
  At this first step, we design a sort hierachy for that error situations are denoted by a sort. In the sort hierachy of expressions, we use the `ExpErr` sort to denote error expressions. Moreover, a constant is declared as a element of this sort (e.g. the `exp-err` is a element of the `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 `exp-err`). The sort hierachy must show that all elements of a normal sort are not equal to the error constant. In this example, all elements of `Exp` sort are not equal to the `exp-err`.

- Step 2: Sort difference for operators.

  Based on the sort hierachy, operators which used for execution are declared. For the expressions, the `evalExp` is declared,
  ```
  op evalExp :  Nat ExpNormal -> Nat .
  op evalExp :  Exp Env -> ExpFull .
  ```
  Depending on the input sorts `Nat` or `Exp`, the output sort will be `Nat` or `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.
  ```
  eq evalExp(V:Var,EV:EnvNormal) =
        if (lookup(V,EV) = exp-err) then exp-err else lookup(V,EV) fi .
  ```
  The "run-time" means that executions will be done with specific input value. The equation shows that with a specific `V:Var`, the run-time execution detemines that the output will be a normal value or an error expression, denoted by `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 `interpret, eval` or `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 Interpterter describes algebraic semantics of Minila programs. And the Virtual Machine describes semantics of Machine language. The Compiler transforms Minila programs into sequences of Machine intructions, 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
```

```
op compile-exp : Exp -> CList
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

```
th-exp(E:Exp,N:Nat) =
       ( 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:Stm` statement. Therefore, the correctness of the Compiler for a statement S is defined as a 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,

```
op interpret : Stm Env -> Env .
op compile : Stm -> CList .
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 Machined 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

```
(interpret(Stm,Env1) = Env2) iff (vm(compile(Stm),Env1) = Env2)
```

In practice, the `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 `iff`, the CafeOBJ keyword. The `iff` has "if and only if" meaning, it's stronger than the `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 `vm` are compared through `Env2`, an intermediate Environment. When the `Stm` is a infinite loop statement, the `(interpret(Stm,Env1) = Env2)` equation becomes $false$ and the same thing for the `(vm(compile(Stm),Env1) = Env2)`.

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

3.2 Verification by Using CafeOBJ

In CafeOBJ language, a formalization is verified by executing *proof scores*. If all proof scores for a formalization are passed (returned *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 `th` which stored in the `THEOREM` module, hence it begins with "`open THEOREM`" and ending with "`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 `th` is reduced with arbitrary inputs, `stm, env-normal` and `env-halt`.

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

After executed by CafeOBJ, the "`red th(stm,env-normal,env-halt)`" returns *true*. It means that the `th` is satisfied with the variable existing in the Environment. However, there is one more case for the `var1`, when the `var1` does not exist in the `env-normal`. Therefore, for that case, we need an another proof score which has an assumption like "`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 *Case Splitting*.

3.2.2 Case Splitting

A proof score usually requires some assumptions to be able to reduce into *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 "`lookup(var1,env-normal) = nat`" assumption is adding, we must conduct a following proof score,

```
open THEOREM
-- arbitrary values
   op var1 : -> Var .
-- assumptions
   eq stm = var := var1  ; .
-- (lookup(var1,env-normal) = nat) = false .
-- var1 was not  stored in env-normal
   eq (lookup(var1,env-normal) = exp-err) = true .
-- check
   red th(stm,env-normal,env-halt) .
close
```

Instuitively, the "`(lookup(var1,env-normal) = exp-err) = true`" and the "`lookup(var1,env-normal) = nat`" are contradicted. The `lookup` of `var1` returns a

value `nat` is the oppsite to that no value is returned which determined as `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 `exp` with respect to `env-normal`, the result might be a natural number or `exp-err` of the error case. However, a number might be equal 0 or `s(nat)`, a non zero number. Therefore, we have three cases as follows,

```
eq evalExp(exp,env-normal)  = 0 .
eq evalExp(exp,env-normal)  = s(nat) .
eq evalExp(exp,env-normal)  = exp-err .
```

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,

```
eq evalExp(exp,env-normal) = 0 .
eq (evalExp(exp,env-normal) = 0) = false .
```

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 `EnvNormal, EnvErr, EnvNonHalt`, respectly. The Case Splitting for evaluation of `stm` statement with respect to `env-normal` by the `eval` operator of Interpreter is showed as follows,

```
eq eval(stm,env-normal) = env1-normal .
eq eval(stm,env-normal) = env-err .
eq eval(stm,env-normal) = env-non-halt .
```

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

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

If proof scores for all cases return *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 *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 `th`.

3.3 Correctness's Verification

3.3.1 Theorem th

The theorem 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 th. Theorem th is the detail of formalization 3.1.2.

The th is specified in THEOREM module as follows,

```
op th : Stm EnvHalt EnvHalt -> Bool .
th(Stm:Stm,Env1:EnvHalt,Env2:EnvHalt) =
  (interpret(Stm,Env1) =  Env2)  implies  (vm(compile(Stm),Env1) = Env2) .
```

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

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

In order to verify the theorem th for an arbitrary statement Stm, we apply induction on the Stm with

- Basic Cases: The th will be verified with input Stm is a *basic* statement including the *empty*, *assignment*, *if*, *while*, *for* and *repeat* statement.
- Induction Cases: The th will be verified with input Stm is a sequential statement. The sequential statement is constructed with a basic statement including the *assignment*, *if*, *while*, *for*, *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

+ stm = estm	①	(Empty statement)
+ stm = var := nat ;	②	(Assignment statement)
+ stm = var := var1 ;	③	(Assignment statement)

+ `stm = var := exp ;` ④ (Assignment statement)
+ `stm = if exp then stm1 else stm2 if` ⑤
+ `stm = while exp do stm1 od` ⑥
+ `stm = for var from e1 to e2 do stm1 od` ⑦
+ `stm = repeat stm1 until exp .` ⑧

- Induction Cases: Sequential statement beginning with a basic statement

  + `stm = var := nat ; stm*1` (with an assignment statement)
  + `stm = var := var1 ; stm*1` (with an assignment statement)
  + `stm = var := exp ; stm*1` (with an assignment statement)
  + `stm = if exp then stm1 else stm2 fi stm*1` (with an If statement)
  + `stm = while exp do stm1 od stm*1` (with a while statement)
  + `stm = for var from e1 to e2 do stm1 od stm*1` (with a for statement)
  + `stm = repeat stm1 until exp stm*1` (with a repeat statement)

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 *true* without any assumptions.

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

Similarly, the case ② returns *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,

```
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 4^{th} assumption, the proof score returns *false*. Looking at the 4^{th} 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 2^{nd} 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 4^{th} assumption,

```
eq exec2(nth((genForExp(exp) @ (store(var) | (quit | nil))),0),
      (genForExp(exp) @ (store(var) | (quit | nil))),0,empstk,env-normal)
   = ((< var , nat >) | env-normal) .
```

is a lemma, called `th2`, then the `th2` will become $false$ by the 4^{th} 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,

```
exec2(nth((store(var) | (quit | nil)),0),
          (store(var) | (quit | nil)),0,(nat | empstk),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,

```
eq th2(Exp:Exp,V:Var,Stk:StackNormal,Env:EnvNormal) =
     (exec2(nth((genForExp(Exp) @ (store(V) | quit | nil)),0),
         (genForExp(Exp) @ (store(V) | quit | nil)),0,Stk,Env)
    = exec2(nth((store(V) | quit | nil),0),
         (genForExp(Exp) @ (store(V) | quit | nil)),len(genForExp(Exp)),
            (evalExp(Exp,Env) | Stk),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.

```
eq inv1(CL:CList,CL1:CList,PC:Nat) =
    (nth(CL @ CL1,len(CL) + PC) = nth(CL1,PC) ) .
eq inv2(Stm:Stm,CL:CList) =
     generator(Stm,CL) = (CL @ generator(Stm,nil)) .
eq sd-lemma(N1:Nat,N2:Nat) = (sd((N1 + N2),N1) = N2) .
```

The `nth(CL:CList,PC:Nat)` returns a command at position `PC` of the `CL` list. The `inv1` lemma shows that a command at `PC` of the `CL1` is same as a command at position `(len(CL) + PC)` of the `(CL @ CL1)`. The `inv2` shows the meaning of the `generator` function for an abstract statement `Stm`. And the `sd-lemma` means that $((N1+N2)-N1) = N2$ with $N1$, $N2$ are natural numbers.

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

Generalizing Lemmas

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

```
 eq th3(Exp:Exp,CL1:CList,CL:CList,Stk:StackNormal,Env:EnvNormal) =
    (exec2(nth((genForExp(Exp) @ CL),0),
          (CL1 @ genForExp(Exp) @ CL),len(CL1),Stk,Env)
   = exec2(nth(CL,0),
          (CL1 @ genForExp(Exp) @ CL),(len(CL1) + len(genForExp(Exp))),
                                      (evalExp(Exp,Env) | Stk),Env)) .
```

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

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

The verification of `th3` will be done by checking proof scores for each cases of expression by applied Case Splitting. The `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 `e1` and `e2` are expressions.

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

```
open THEOREM
-- arbitrary values
   ops e1 e2 : -> Exp .
   op n1 : -> Nat .
-- assumptions
   eq evalExp(e1,env-normal) = n1 .
```

```
   --
   eq evalExp(e2,env-normal) = 0 .
   ....
-- check
   red th3(e1 // e2,cl1,cl,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,

```
op _|_ : ExpFull Stack -> StackErr .
ceq (E:ExpFull | Stk:Stack) = stkErr if (E = exp-err) .
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 ⑥. The input `stm` is as follows,

```
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,

```
eq evalExp(exp,env-normal) = 0 .
```

execution of the `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 `stm1`, then the `stm` is looped again. The local `stm1` could be applied as the induction hypothesis and the `eval(stm1,env-normal)` might be used for semantics of `stm1` with respect to `env-normal`. The input Environment for the next execution of `stm` becomes `eval(stm1,env-normal)`. In general, after many looping times, the Environment has a form as follows,

```
eval(stm1,eval(stm1,...(eval(stm1,env-normal)))) .
```

The general form of input Environment after N looping times will be specified by `create-env` operator in module `THEOREM-CONDITION` as follows,

```
mod THEOREM-CONDITION {
    ...
  op create-env : Stm Env Nat -> Env
  eq create-env(Stm:Stm,EV:EnvNormal,0) = EV .
  eq create-env(Stm:Stm,EV:EnvNormal,s(N:Nat))
       = eval(Stm,create-env(Stm,EV,N)) .
  ...
}
```

The `create-env(stm1,env-normal,N)` is a returned Environment after doing the `stm1` N times with respect to `env-normal`. It's clear that the `create-env(stm1,env-normal,N)` might be an error or non-termination Environment, respectly `env-err` or `env-non-halt`.

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

```
  op notZero : Exp Stm EnvNormal Nat -> Bool .
  eq notZero(Exp:Exp,Stm:Stm,EV:EnvNormal,0)
       = not (evalExp(Exp,EV) = 0 or evalExp(Exp,EV) = exp-err) .
  eq notZero(Exp:Exp,Stm:Stm,EV:EnvNormal,s(N:Nat)) =
       not (create-env(Stm,EV,s(N)) = env-non-halt or
            create-env(Stm,EV,s(N)) = env-err)
       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) .
  --
  op notZeroEqZero : Exp Stm EnvNormal Nat -> Bool .
  eq notZeroEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,0) = (evalExp(Exp,EV) = 0) .
  eq notZeroEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,s(N:Nat)) =
       not (create-env(Stm,EV,s(N)) = env-non-halt or
            create-env(Stm,EV,s(N)) = env-err)
       and (notZero(Exp,Stm,EV,N)  and
            evalExp(Exp,create-env(Stm,EV,s(N))) = 0) .
  --
  op existEqZero : Exp Stm EnvNormal Nat -> Bool
  eq existEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,N:Nat) =
      not (create-env(Stm,EV,N) = env-non-halt or
             create-env(Stm,EV,N) = env-err)
        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):
  $\forall k \in$ Nat, $k \leq N$, evalExp(Exp,create-env(Stm,EV,k)) > 0
- notZeroEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,N:Nat):
  $\forall k \in$ Nat, $k < N$, (evalExp(Exp,create-env(Stm,EV,k)) > 0) and
  (evalExp(Exp,create-env(Stm,EV,N)) $= 0$)
- existEqZero(Exp:Exp,Stm:Stm,EV:EnvNormal,k:Nat):
  $\exists k \in$ Nat, evalExp(Exp,create-env(Stm,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 $\mathrm{N}+1$ times. If `notZeroEqZero(Exp,Stm,EV,N)` is *true*, the `Stm` certainly termintes 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,

$N \in$Nat, $\exists M \leq N$, existEqZero(Exp,Stm,EV,N) $\longrightarrow$ notZeroEqZero(Exp,Stm,EV,M)

```
eq notZeroEqZero-lemma(Exp:Exp,Stm:Stm,EV:EnvNormal,N:Nat,M:Nat) =
   existEqZero(Exp,Stm,EV,N) implies
           (M < s(N) implies notZeroEqZero(Exp,Stm,EV,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.

```
--> 1.i)
open THEOREM
   op m : -> 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`. Becase 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,

$$N \in \text{Nat}, \exists M \leq N,$$
$$\textit{not}\ (\text{notZero(Exp,Stm,EV,N)}) \longrightarrow \text{evalExp(Exp,create-env(Stm,EV,M))} = 0\ .$$

```
eq notZero-lemma(Exp:Exp,Stm:Stm,EV:EnvNormal,N:Nat,M:Nat) =
   (not notZero(Exp,Stm,EV,N) and
    not (evalExp(Exp,create-env(Stm,EV,N)) = exp-err) and
    not (  create-env(Stm,EV,N) = env-non-halt or
           create-env(Stm,EV,N) = env-err) )
         implies (M < s(N) implies notZeroEqZero(Exp,Stm,EV,M)) .
```

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

- $\exists N \in$Nat, existEqZero(Exp,Stm,EV,N) $=$ *true*
- $\forall N \in$Nat, 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,

```
eq definition(Exp:Exp,Stm:Stm,EV:EnvNormal,N:Nat) =
     notZero(Exp,Stm,EV,N)
          implies (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) =
   not (eval(Stm,EV) = env-non-halt) implies
      (exec2(nth((generator(Stm,nil) @ CL),0),
             (CL1 @ generator(Stm,nil) @ CL),len(CL1),Stk,EV)
     = exec2(nth(CL,0),
             (CL1 @ generator(Stm,nil) @ CL),
                   len(generator(Stm,nil)) + len(CL1),Stk,eval(Stm,EV)))  .
```

The structure of `th4` is quite similar to the `th3`. The `th4` considers a general list, `(CL1 @ generator(Stm,nil) @ CL)`, in which the position of `generator(Stm,nil)` is arbitrary. In the `th4`, the Interpreter specifies semantics of statement `Stm` with respect to `EV` by executing `eval(Stm,EV)`. And the Virtual Machine executes the list of commands that generated by `generator(Stm,nil)` of the Compiler. This execution is done by the `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 `eval(Stm,EV)` could not return a specific Environment, and the right hand side of `th4` equation might be not detemined. 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 `env-non-halt` of `EnvNonHalt` sort in the Enviroment Hierachy at 2.2. And in theorem `th4`, the exception for infinite loop statements is expressed by following condition,

```
not (eval(Stm,EV) = env-non-halt)
```

it means when `Stm` is an infinite loop statement, this condition becomes $false$ and the `th4` is trivial.

When building the theorem `th4`, we tried not to include the above condition, but the provement 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 `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 `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 `th4`.

In the `th4`, the value of `eval(Stm,EV)` is replaced for the value is generated by the `exec2` of Virtual Machine. It shows that semantics of a statement `Stm` that are generated by the Interpreter and the Virtual Machine are equivalent. However, the `th4` also confirms that the input `Stm` is limited with finite loop statements by the termination condition. The correctness defined by the `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 `th4` is an important step for proving the `th`, the main object of this research. Intuitively, the `th4` is a specific case of the `th2`. The accepted statements of `th4` are just finite statements, that is a subset of Minila statement. The condition of `th4` is one reason of that the `iff` relation is not satisfied for the `th`. By applied Case Splitting for non termination situations, only `(interpret(Stm,Env1) = Env2)` becomes $false$ and the `th` is trivial with the `implies` relation.

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

The significance of Environment Hiearchy

In practice, a statement could not be determined as an infinite loop statement at beginning. 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-terminiation. Applied Case Splitting for the statement object requires hierachy of Environment which is desiged like the Environment Hierachy 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 Hierachy,

```
eq eval(stm,env-normal) = env1-normal .
eq eval(stm,env-normal) = env-err .
eq eval(stm,env-normal) = env-non-halt .
```

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,

```
stm = if exp then stm1 else stm2 fi .
```

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 scrores we see that there are three assumptions for two splitted cases,

```
eq existEqZero(exp,stm1,env-normal,0) = true .
eq existEqZero(exp,stm1,env-normal,s(nat)) = true .
eq notZero(exp,stm1,env-normal,k) = true .
```

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

```
eq evalExp(exp,env-normal) = 0 .
```

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

```
eq notZeroEqZero(exp,stm1,env-normal,s(nat)) = true .
```

is replaced by three assumptions

```
eq eval(stm1,create-env(stm1,env-normal,nat)) = env1-normal .
eq notZero(exp,stm1,env-normal,nat) = true .
eq evalExp(exp,env1-normal) = 0 .
```

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

```
red definition(exp,stm1,env-normal,k)
      implies th4(stm,cl1,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 `iff` for theorem `th1`, but `implies` for the `th`

We could realize that the `th4` is a general form of the theorem `th1` as follows,

```
eq th1(Stm:Stm,EV:EnvNormal,EV2:EnvHalt) =
   not(interpret(Stm,EV) = env-non-halt)
     implies ( interpret(Stm,EV) = EV2 iff  vm(compile(Stm),EV) = EV2 ) .
```

it's exactly a specific case of the theorem `th` with the condition accepts only finite loop statements, `not (interpret(Stm,EV) = env-non-halt)`.

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

```
eq th(Stm:Stm,Env1:EnvHalt,Env2:EnvHalt) =
  (interpret(Stm,Env1) =  Env2)  implies  (vm(compile(Stm),Env1) = Env2) .
```

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

```
eq definition(Exp:Exp,Stm:Stm,EV:EnvNormal,N:Nat) =
     notZero(Exp,Stm,EV,N)
           implies (interpret(while Exp do Stm od,EV) = env-non-halt) .
```

mentioned in above section. This is a definition for the meaning of while statement. With this definition as hypothesis, the `th1` (or `th4`) is validated. However the `th` could not validate with the `iff` relation because the rightside, `vm(compile(Stm),Env1) = Env2`, of `th` could become $false$. There is no definition for the Virtual Machine in this case. It means the `th` only satified with the `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 `th1` and `th`.

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

The complete verifications of the `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 corrrectness 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,

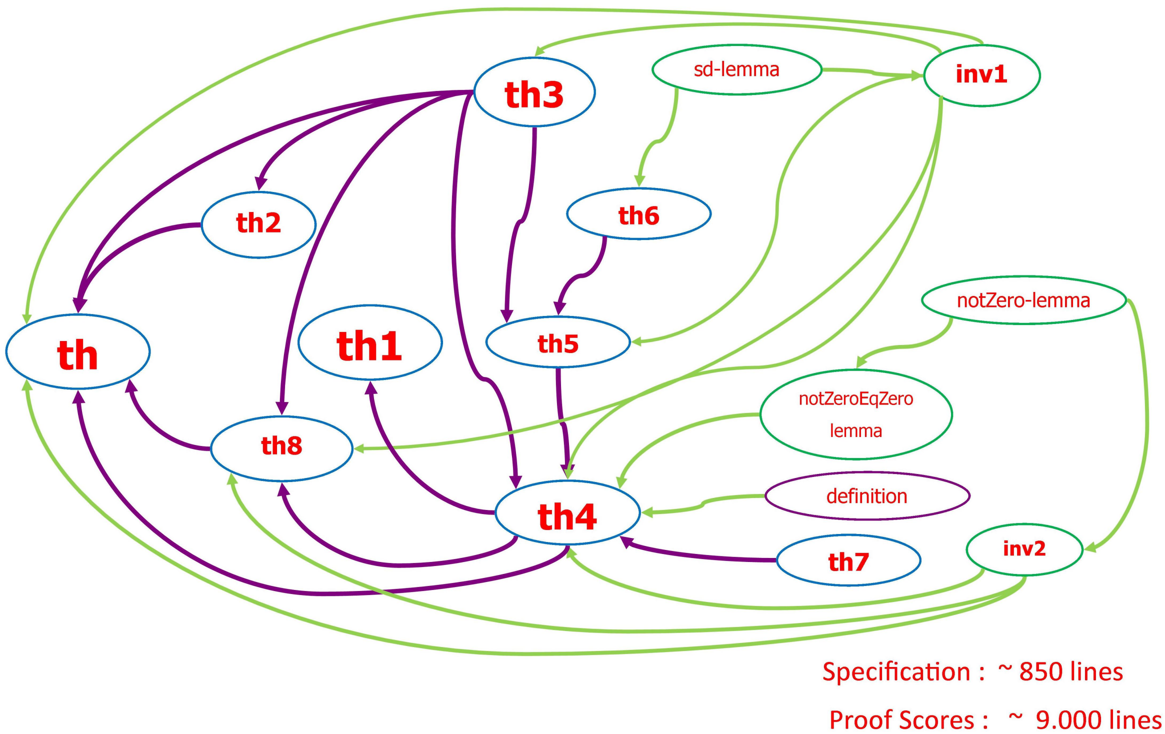

Figure 3.1: Theorem Relation Diagram

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 separete 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 theorm `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 hiearchy, especilly the Environment Hierachy. 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 theorm `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 Hierachy.

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 Hierachy

Environment Hierachy is one possible method when specify and verify the statement of Minila language, especially for the infinite looping statement. With this hierachy, 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 arithmetic 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 Algerbraic 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 langauge) to Power PC assembly code, using the Cop proof assistant both for programming the compiler and for proving its correctness.

Strecker [15] and Klein and Kipkow [8] verified non-optimizing byte-code compilers from a subset of Java to a subset of the Java Virtual Machine using Isabelle/HOL. They did not address compiler optimizations nor generation of actual machine code.

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 verfication 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

[1] A. Sampaio. *An Algebaic Approach to Compiler Design*, AMAST Series in Computing: Vol.4,page 2, 3

[2] A. Sampaio. *A comparative study of theorem provers: Proving correctness of compiling specifications*. Technical Report PRG-TR-20-90, Oxford University Computing Laboratory, 1990.

[3] Diaconescu, R. and K. Futatsugi. *CafeOBJ report: The Language, Proof Techniques, and Methodlogies for Object-Oriented Algebraic Specification*. AMAST Series in Computing 6, World Scientific, 1998

[4] D. Leinenbach, W.Paul, and E. Petrova. *Towards the formal verification of a C0 compiler: Code generation and implementation correctness*. In Int.Conf. on Software Engineering and Formal Methods (SEFM 2005), pages 2-11. IEEE Computer Society Press, 2005.

[5] F. da Silva *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992. Published as LFCS Report Series ECS-LFCS-92-241 or CST-95-92.

[6] http://www.ldl.jaist.ac.jp/cafeobj/index.html

[7] G. Goos and W. Zimmermann. *Verification of compilers*. In Correct System Design, Recent Insight and Advances, volume 1710 of LNCS, page 201-230. Springer-Verlag, 1999.

[8] G. Klein and T. Nipkow. *A machine-checked modle for a Java-like language, virtual machine and compiler* Technical Report 0400001T.1, National ICT Australia, Mar.2004. To appear in ACM TOPLAS.

[9] J.A.Goguen, G. Malcolm. *Algebraic Semantics of Imperative Programs.*

[10] J. A. Goguen et al. *In troducing OBJ*. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, California, USA, 1992. Revised version to appera in J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Practice*. Cambridge University Press, 1997.

[11] John McCarthy, and James Painter. *Correctness Of A Compiler For Arithmetic Expression* Proceedings Symposium in Applied Mathematics, Vol 19, Mathematical Aspects of Computer Science, 1967.

[12] J. Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Computer Science Department, Aarhus University, 1992.

[13] P. D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[14] M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Ab- straction.* Lecture Notes in Computer Science (LNCS), Vol. 1283. Springer-Verlag, Heidelberg, Germany, 1997.

[15] M. Strecker. *Formal verification of a Java compiler in Isabelle.* In Proc. Conference on Automated Deduction (CADE), volume 2392 of LNCS, pages 63-77. Springer-Verlag, 2002.

[16] M. Strecker. *Compiler verification for C0.* Technical report, Universite Paul Sabatier, Toulouse, April 2005.

[17] M. A. Dave. *Compiler verification: a bibliography.* SIGSOFT

[18] W. Polak. *Compiler Specification and Verification.* volume 124 of Lecture Notes in Computer Science. Springer-Verlag, 1981.

[19] X. Leroy *Formal certification of a compiler back-end or: programming a compiler with a proof assistant.* In 33rd Symposium on the Principles of Programming Languages (2006), ACM, pages 42-54.