A Haskell Implementation for a Dependently Typed Language

Master's thesis in Computer science and engineering

QUFEI WANG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

A Haskell Implementation for a Dependently Typed Language

QUFEI WANG

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

A Haskell Implementation for a Dependently Typed Language
QUFEI WANG

Supervisor: Thierry Coquand, Department of Computer Science and Engineering
Examiner: Ana Bove, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

A Haskell Implementation for a Dependently Typed Language
QUFEI WANG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

We present in this paper a simple dependently typed language. The basic form of this language contains only a universe of small types, variables, lambda abstraction, function application and dependent product as its syntax. There is no data types and mutual recursive/inductive definitions is not supported. This language could be viewed as a lambda calculus extended with dependent types and constant definitions. The focus of this project is not on the expressiveness of the language but on the study of a definition mechanism where the definitions of constants could be handled efficiently during the type checking process. Keeping the syntax simple helps us focus on our purpose and makes an elegant implementation possible. We later enriched the language with a module system not to increase its expressiveness, but to study how the definition mechanism should be adjusted for the introduction of namespaces to the variables. The outcome of our work is a REPL(read-evaluate-print-loop) program through which a source file of our language could be loaded and type checked. The program also provides auxiliary functions for the user to experiment with and observe the effect of the definition mechanism. The syntax of our language is specified by the BNF converter and the program is implemented in Haskell. We hold the expectation that our work could contribute to the development of the proof systems that are based on the dependent type theory.

Keywords: computer science, dependent type theory, functional programming, type checker.

Acknowledgements

This project would not have been possible without the support of many people. Many thanks to my supervisor Thierry Coquand for his guidance, patience and share of knowledge. Thank you to my examiner Ana Bove for her precious time and suggestions on the quality of the work. Most importantly, I want to thank my parents for their unconditional love, and my wife Kefang Zhao for her long standing consideration and support.

Qufei Wang, Gothenburg, September 2021

Contents

1 Terminology **1**

2 Introduction **3**

2.1 Some Background About Dependent Types 3
2.2 Issues with Dependent Type 3
2.3 Aim of the Project 4
2.4 Limitations 6

3 Theory **7**

3.1 Subtleties of Dependent Type Theory 7
3.2 Definitions in a Dependent Type Theory 8
3.3 Syntax of the Language 10
3.4 Operational Semantics 11
3.5 Type Checking Rules 12
3.5.1 Type Checking Context 12
3.5.2 checkDecl 15
3.5.3 checkInferT 15
3.5.4 checkWithT 16
3.5.5 checkEqualInferT 17
3.5.6 CheckEqualWithT 19
3.6 Locking Mechanism 20
3.7 Head Reduction 21

4 Extension **25**

5 Results **27**

6 Conclusion **29**

Bibliography **31**

A Appendix **I**

A.1 Haskell Source Code I
A.2 Concrete Syntax I
A.3 Test Case II

1

Terminology

In order to make clear of the potential ambiguity or unnecessary confusion over the words we choose to use in this paper, we list below the terminology we use and their meanings:

- **Declaration:** A *declaration* has either the form $x : A$ or $x : A = B$. The latter is also referred, rather frequently, as a *definition*. Sometimes when we want to make a distinction between these two forms, we also use the word 'declaration' specifically to indicate a term of the former form.

- **Definition:** A *definition* is a term of the form $x : A = B$, meaning that x is an element of type A, defined as B. Sometimes when we want to talk about the components of a specific definition, we also use the word 'definition' specifically to indicate the part of B, e.g., "the definition of x is B" .

- **Constant:** A *constant* is the entity that a name or identifier used in a declaration refers to, e.g., the entity that x in $x : A$ or $x : A = B$ refers to.

- **Variable:** A synonym for *constant*. More often, the word 'variable' is used to refer to the variable bound in a λ-abstraction, like the variable x in $\lambda x.A$. In most cases, these two words are interchangeable.

2

Introduction

2.1 Some Background About Dependent Types

Dependent type theory has lent much of its power to the proof-assistant systems like Coq [1], Lean [2], and functional programming languages like Agda [3] and Idris [4], and contributed much to their success. Essentially, dependent types are types that depend on *values* of other types. As a simple example, consider the type that represents vectors of length n comprising of elements of type A, which can be expressed as a dependent type (`vec` A n). Readers may easily recall that in imperative languages such as C/C++ or Java, there are array types which depend on the type of their elements, but not types that depend on values of other types. More formally, suppose we have defined a function which to an arbitrary object x of type A assigns a type $B(x)$, then the Cartesian product $(\Pi x \in A)B(x)$ is a type, namely the type of functions which take an arbitrary object x of type A into an object of type $B(x)$.

The advantage of having a strong typed system built into a language lies in the fact that well typed programs exclude a large portion of run-time errors than those without or with weak type systems. Just as the famous saying puts it "well-type programs cannot 'go wrong'" [16]. It is in this sense that we say languages with dependent types are guaranteed with the highest level of correctness and precision, which makes them a natural option for building proof assistant systems.

2.2 Issues with Dependent Type

The downside of dependent type systems lies in the difficulties of implementation, among which is the notable problem of checking the **convertibility** of terms: to decide whether a term A is identical with term B. Checking the convertibility of terms that represent types is a routinely performed task by the type checker of any typed language, thus is vital for its performance. In a simple typed language, convertibility checking is done by simply checking the syntactic identity of the symbols of the types. For example, in Java, a primitive type *int* equals only to itself, nothing more.

This is because types in Java are not computable [1]:there's no way for other terms in Java be reduced to the term *int*. In a dependently typed language, however, the problem is more complex since a type may contain terms of any expression, deciding the convertibility of types in this case entails evaluation of any terms, which requires much more computation.

One common approach to deciding the convertibility of terms in dependent type theory, whenever the property of confluence holds, is *normalization by evaluation* (NbE) [5], which reduces terms to their canonical representation for comparison. This method, however, does not scale to large theories for various reasons, among which:

- Producing the normal form may require more reduction steps than necessary. For example, in proving $(1+1)^{10} = 2^{(5+5)}$, it is easier if we can prove $1+1 == 2$ and $5+5 == 10$ instead of having to reduce both sides to 1024 by the definition of exponentiation.

- As the number of definition grows, the size of terms by expanding a definition can grow very quickly. For example, the inductive definition $x_n := (x_{n-1}, x_{n-1})$ makes the normal form of x_n grow exponentially.

In this project, we shall focus on the first issue, that is, how to perform as few constant expansions as possible when deciding the convertibility of two terms in a dependently typed language.

2.3 Aim of the Project

The first aim of the project is to study how to present *definitions* in a dependently typed language, more precisely, how to do type checking in the presence of *definitions*? We hope that the definitions of constants could be expanded as few times as possible during the type checking process. We claim that a good definition mechanism can help improve the performance of a language that is based on dependent type theory. We will justify this claim later by giving an analysis to the example above. Before that, we shall first make it clear for the reader this question: what exactly is the problem of definition and why is it important?

A definition in a dependently typed language is a declaration of the form $x : A = B$, meaning that x is a constant of type A, defined as B. The problem with definitions is not about how a constant should be declared, but how it should be evaluated. Evaluation, or reduction, in dependent type theory has its concept originated in λ-*calculus* [6]. There, a term of the form $(\lambda x.M)\ N$ can be evaluated (or reduced) to the form $M[x := N]$, meaning the substitution of N for the free occurrences of x

[1]Technically speaking, the type of an object in Java can be retrieved by the Java *reflection* mechanism and presented in the form of another object, thus subject to computation. Here, we stress on the fact that a type as a term is not computable on the syntactic level, e.g. being passed as an argument to a function.

in M[2]. In dependent type theory, however, different evaluation strategies can have huge difference regarding the efficiency of evaluation.

For example, if we define the exponentiation function on natural numbers as

$$\begin{aligned}&\texttt{expo} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}\\ &\texttt{expo}\ \_\ \ 0 = 1\\ &\texttt{expo}\ \ n\ \ m = n * (\texttt{expo}\ \ n\ \ (m-1))\end{aligned}$$

where `Nat` is the type of natural number and $(*)$ is the operator of multiplication. Then when we try to prove the convertibility of two terms: $(1+1)^{10}$ and $2^{(5+5)}$, instead of unfolding the definition of `expo` multiple times, we keep the constant `expo` **locked** and only reduce both sides to the term $(\texttt{expo}\ \ 2\ \ 10)$. Then by showing that they can be reduced to a common term(having the same symbolic representation), we proved their equality with much less computation. Locking a constant has the effect of turning its definition $x : A = B$ into a declaration $x : A$, so that the type information can still be used in the type checking process, whereas the definition is erased so that the constant cannot be reduced further.

The second aim of the project is to extend the language with a module system based on the idea 'segment' originated from the work AUTOMATH [7]. AUTOMATH was initiated by Dutch mathematician N.G.de Bruijn where special languages were developed to express concepts in mathematics. The ingenuity of the work is that the correctness(in the sense of mathematical deduction) of a text written in AUTOMATH could be checked by a computer program mechanically. The idea of 'segment' was introduced as a facility of abbreviation in AUTOMATH for easier reference to patterns of strings that usually appear as part of a larger formula. For example, using the notations from λ-calculus, consider a function abstraction of the form $\lambda x_1 . \lambda x_2 \ldots \lambda x_n . \phi$, where $x_1, \ldots, x_n$ are bound variables. If n is large and this pattern of abstraction appears multiple times in a larger formula, then the abstraction part over $x_1, \ldots, x_n$, i.e., $\lambda x_1 . \lambda x_2 \ldots \lambda x_n$ without the body ϕ could be taken as a 'segment' in AUTOMATH.

We use the idea 'segment' to build a module system that shares similarities with those of the common programming languages like Java and Haskell, but less expressive. Again, our aim is not to increase the expressiveness of the language but to study how the definition mechanism built in the first step should be adjusted to accommodate the concept of 'namespaces' introduced by the module system. The result could be used as an evidence for the portability and scalability of our definition mechanism.

[2]There is a problem of the capture of free variables which we will not elaborate here. Curious and uninformed readers are encouraged to read articles introducing *λ-calculus.*, especially the 1984 book *The lambda calculus: its syntax and semantics* by Hendrik Pieter Barendregt

2.4 Limitations

The limitations of our work come into three aspects: expressiveness, scope and meta-theory.

1. **Expressiveness/Usability:** We try to keep the syntax of the language as simple as possible in order to focus on the study of a definition mechanism. This practice inevitably affects the expressiveness and usability of our language: As has been mentioned, there is no syntax in support for creating new data types, defining recursive functions or doing pattern match on expressions; Further, there is a lack of basic input/output functionalities(one cannot even print a "Hello,world!" on the screen); Besides, because we track the names of constants in a linear manner as an approach to the name collision problem (see example 2 in section 3.1) and enforce the policy that declaration of names must not collide with the names in the current context, the common programming language feature *variable shadowing* does not exist in our language; Lastly, the result of the project is only a REPL that incorporates a type checker(together with a lexer and parser which are automatically generated by the BNF convertor) and an interpreter, the former to load and type check a source file and the latter to evaluate expressions. There is no support in the view of a compiler that generates executable machine code(of course for a language worthy of executable program compilation, it must have basic data types and IO functions).

2. **Scope:** The definition mechanism we established is not meant to be the most effective or applicable universally to different kinds of dependently typed languages. However, the ideas suggested in this paper are highly likely to find a much wider using scenario.

3. **Metatheory:** We do not present the metatheory behind our system. Since our system shares much of its idea with Mini-TT, there should be some correspondence between the metatheory of these two systems, such as the property of the decidability of the type checking algorithm. But we will not conduct an analysis on this due to the limit of time and the limit of my knowledge.

3

Theory

Our system could be seen as an extension to a λ-*calculus* with dependent types and definitions. In order for the reader to understand better the idea behind the choice of the syntax and semantics of our language, we need first to address some subtleties that differentiate our system from λ-*calculus* and that back our choice for dealing with the names of the constants.

3.1 Subtleties of Dependent Type Theory

We present the subtleties by giving the following examples:

Example 1. Suppose we have

$$a : A, \quad P : A \to U, \quad f : P\, a \to P\, a$$

Then the term

$$\lambda(x : A)(y : P\, x)\ .\ f\, y$$

is not well typed because the type of y is $(P\, x)$ not $(P\, a)$.

However, if we modify this term to

$$\lambda(x : A = a)(y : P\, x)\ .\ f\, y$$

then it is well typed.

We see here that the definition of x impacts the type safety of the whole expression. This example shows that definitions in dependent type theory cannot be reduced to λ-*calculus*.

Example 2. Suppose we have

$$\lambda(x : \texttt{Nat})(y : \texttt{Nat} = x)(x : \texttt{Bool})\ .\ M$$

In this term, the first declaration of x is shadowed by the second one. Later when we do some computation on M, if we do not take the shadowing of the name of x carefully, then the constant y will become ill formed.

This example shows that in a dependent type theory, names of variables must be handled with great care.

Example 3. Suppose we have

$$\begin{aligned}
&x : A \\
&y : A \\
&b : A \to A \to A \\
&u : (A \to A \to A) \to (A \to A \to A) \\
&a : (A \to A) \to (A \to A) \\
&z : A \to A \to A
\end{aligned}$$

Then the term below is well typed.

$$(\lambda u\ .\ u\ (u\ b))(\lambda z\ y\ x\ .\ a\ (z\ x)\ y) \tag{3.1}$$

If we do the reduction on (3.1) naively, we get

$$\begin{aligned}
(\lambda u\ .\ u\ (u\ b))(\lambda z\ y\ x\ .\ a\ (z\ x)\ y) &\implies \\
(\lambda z\ y\ x\ .\ a\ (z\ x)\ y)((\lambda z\ y\ x\ .\ a\ (z\ x)\ y)\ b) &\implies \\
(\lambda z\ y\ x\ .\ a\ (z\ x)\ y)(\lambda y\ x\ .\ a\ (b\ x)\ y) &\implies \\
\lambda y\ x\ .\ a\ ((\lambda y\ x\ .\ a\ (b\ x)\ y)\ x)\ y &
\end{aligned} \tag{3.2}$$

At this point, we have a capture of variables problem.

(3.2) should be the same as

$$\lambda y\ x\ .\ a\ ((\lambda y\ x'\ .\ a\ (b\ x')\ y)\ x)\ y$$

which reduces to

$$\lambda y\ x\ .\ a\ (\lambda x'\ .\ a\ (b\ x')\ x)\ y$$

But if we do a naive reduction in (3.2) without renaming, we get

$$\lambda y\ x\ .\ a\ (\lambda x\ .\ a\ (b\ x)\ x)\ y$$

which is not correct.

This example shows another aspect of subtlety when dealing with names of variables in a dependent type theory: the capture of variables.

3.2 Definitions in a Dependent Type Theory

The examples listed above provide us with insights into the common pitfalls one should avoid when implementing definitions in dependent type theory. From there, we derived the following principles that guide us through the pitfalls in our own implementation:

Principle 1. For definitions in the form $x : A = B$, treat the type A and the definition B separately.

Principle 2. Forbid the shadowing of variable names.

Principle 3. Rename variable whenever necessary.

Principle 1 relates to example 1. As has been suggested in the example, the definition of a constant can be important to ensure the type safety of an expression. In other cases, however, the definition is not needed, like in this expression $\lambda(f : A \to B)(a : A) \ . \ f \ a$: f could be any function from A to B and a could be any element of A. Regardless of their specific values, we know for sure that the term $f \ a$ has type B. These facts indicate that type and definition take unequal roles in dependent type theory: one can declare a constant without a definition, but cannot declare a constant without a type.

In our implementation, we use two constructs, ρ and Γ, to keep track of the variables with their definitions and with their types. We call ρ the *environment* and Γ the *context*. Essentially, they are list like structures that can be extended with declarations or a single expression acting as a definition or type. We use ρ to get the definition of a constant, Γ for the type. We have an operation to convert a context Γ to an environment ρ, but not the other way around. All the major operations, e.g. type checking, head reduction, etc., exposed by our Haskell program are performed under a top level context.

Principle 2 comes as a simple strategy to avoid the pitfall revealed by example 2. During the type checking process, each declaration, including the declarations from λ-abstractions, is checked with the top level context (actually, we have only one level context) to ensure no naming clash occurs. Using *De Bruijn index* is another, maybe better, from the point of view of the user, way to avoid the name clashing issue. However, having to maintain the relationship between names and indices may unnecessarily complicate our implementation and obscure the main aim of the project.

Principle 3 is less specific by using the phrase 'whenever necessary'. Indeed, it is hard to generalize a rule that works in all conditions. The practice of variable renaming is dependent on the syntax of the language and its evaluation strategy. In our implementation, we rename variables in two situations: one is convertibility checking and the other is reading back a term to the normal form.

Finally, we have a fourth, pillar principle in support of our locking/unlocking mechanism:

Principle 4. Deferred evaluation.

In order to reduce unnecessary reductions during the type checking process, we exploit a locking mechanism where computations are deferred as much as possible. We do this by

1. Using *closure* to carry the intermediate evaluated results.

2. Applying β-reduction on multi-variable functions in an incremental manner.
3. Only unlock a name when reductions on that name is expected.

Having introduced all these 4 principles, now we are ready to describe in detail the syntax and semantics of our language and the operations we built upon it.

3.3 Syntax of the Language

What we describe below is the abstract syntax of our language. For the concrete syntax defined at the source code level, see appendix A.2.

A program of our language consists of a list of declarations. A declaration has either the form $x : A$ or $x : A = B$, where A, B are expressions. A summary of the syntax can be found in table 3.1.

expression	M, N, A, B	$::=$	$U \mid x \mid M\ N \mid [D]M$
declaration	D	$::=$	$x : A \mid x : A = B$

Table 3.1: Language Syntax

The meaning of each expression constructor is explained in table 3.2.

U	:	The type of small types. U is also an element of itself.
x	:	Variables with names, e.g. 'x' , 'y', 'z'.
$M\ N$	:	Function application.
$[D]M$	:	Depending on the form of D, it has different meanings.

Table 3.2: Expressions

An expression in the form $[x : A]\ M$ can be used to represent

- **Dependent Product:** $\Pi x : A \,.\, M$ - the type of functions which take an arbitrary object x of type A into an object of type M (M may dependent on x).
- **λ-abstraction:** $\lambda\,(x : A)\ .\ M$ - a function that takes a variable x of type A into an expression M.

When x does not appear in M (M does not depend on x), this expression is the same as $[\_ : A]M$. When used as a type of function, it means non-dependent functions of type $A \to M$, which we provide as a syntax sugar; When used as a λ-abstraction, it means the constant function $\lambda(\_ : A)\ .\ M$ that always return M regardless of the input argument.

An expression in the form $[x : A = B]M$ can be used to represent

- A *let* clause: *let* $x : A = B$ *in* M, or

- A *where* clause: M *where* $x : A = B$.

The syntax of our language is a substantial subset of Mini-TT. Moreover, we use the same syntax for both dependent product and λ-abstraction as an effort to maintain simplicity. This practice causes ambiguity only when an expression in the form $[x : A]M$ is viewed in isolation: it can be seen both as a dependent type and a function abstraction. This ambiguity, however, does not exist in the type checking rules when the meaning of a term is clear in a certain context.

3.4 Operational Semantics

An *expression* is evaluated to a *quasi-expression*(or *q-expression*) under a given environment. The intuition about the *q-expressions* is that they are intermediate form of expressions and can be computed to ordinary expressions.

The syntax of q-expression is given in table 3.3.

$$\text{q-expression} \quad u, v \quad ::= \quad U \mid x \mid u\,v \mid \langle [x : A]M, \rho \rangle$$

Table 3.3: Syntax of Q-expressions

The meaning of each form of q-expression is given in table 3.4.

U	:	Q-expression form of U.
x	:	Q-expression form of a variable without a definition, a *neutral value*.
$u\ v$	:	Q-expression form of application, where u is not a closure.
$\langle [x : A]M, \rho \rangle$	:	A closure, a function extended with an environment.

Table 3.4: Meaning of Q-expressions

Note that in our Haskell implementation, we use the same syntax for both expressions and q-expressions, since the syntax is similar.

An environment is defined as

$$\rho ::= () \mid \rho,\ x = v \mid \rho,\ x : A = B$$

meaning that an environment could be (i) empty; (ii) extended by a variable bound with a q-expression; (iii) extended by a variable with its definition.

We give the semantics of our language by equations of the form $[\![M]\!]\rho = v$, which means that the expression M evaluates to v under the environment ρ.

The function `appVal` is defined as:

$$\begin{array}{lcl}
[\![U]\!]\rho & = & \mathrm{U} \\
[\![x]\!]\rho & = & \rho(x) \\
[\![M_1\ M_2]\!]\rho & = & \texttt{appVal}([\![M_1]\!]\rho, [\![M_2]\!]\rho) \\
[\![[x : A]\, B]\!]\rho & = & \langle [x : A]\, B, \rho\rangle \\
[\![[x : A = B]\, M]\!]\rho & = & [\![M]\!](\rho, x : A = B)
\end{array}$$

Table 3.5: Semantics of Language

$$\begin{array}{lcl}
\texttt{appVal}(\langle [x : A]B, \rho\rangle, v) & = & [\![B]\!](\rho, x = v) \\
\texttt{appVal}(v1, v2) & = & v1\ v2
\end{array}$$

Table 3.6: Function: appVal

The lookup function to find the value of a variable x in ρ is defined as

$$\begin{array}{rcl}
()(x) & = & x \\
(\rho, x = v)(x) & = & v \\
(\rho, y = v)(x) & = & \rho(x)(y \neq x) \\
(\rho, x : A = B)(x) & = & [\![B]\!]\rho \\
(\rho, y : A = B)(x) & = & \rho(x)(y \neq x)
\end{array}$$

Note that the type information in a definition is always discarded.

3.5 Type Checking Rules

3.5.1 Type Checking Context

The type checking procedure is performed under a context Γ:

$$\Gamma ::= () \mid \Gamma,\ x : A \mid \Gamma,\ x : A = B$$

meaning that a type checking context could be (i) empty; (ii) extended by a variable bound with an expression as its type; (iii) extended by a variable with its definition.

The lookup operation to find the type of a variable x in Γ is defined as

$$\begin{array}{rcl}
()(x) & = & \texttt{error} \\
(\Gamma, x : A)(x) & = & A \\
(\Gamma, y : A)(x) & = & \Gamma(x)(y \neq x) \\
(\Gamma, x : A = B)(x) & = & A \\
(\Gamma, y : A = B)(x) & = & \Gamma(x)(y \neq x)
\end{array}$$

Note that the definition part is always discarded.

In our implementation, when parsing the source file into the abstract syntax of our language, we make sure that each variable is properly declared with a type and the name of the variable does not clash with the existing ones. By doing so, we ensure that the error condition in the lookup operation will never occur during the type checking process and each variable's name is unique.

In order to facilitate the process of variable renaming, we also defined two auxiliary functions - `varsCont` and `freshVar`:

- `varsCont` :: `Cont` → [`String`]: return the names of a context. (`Cont` represents the type of context.)
- `freshVar` :: `String` → [`String`] → `String`: given a name s and a list of names (usually the names of a context), return s if it does not belong to the list; otherwise, return a new name that is not in the list.

The definitions of these two functions are given in the table 3.7. Note that when the fist argument passed to the function `freshVar` is an empty string, which represents a dummy variable, we replace the argument with a string of value "var" and then apply the function. The reason is that in our implementation, the type checking context doesn't keep track of the dummy variables because they do not appear in the body of a λ-abstraction. This means that when trying to generate a new name in a context using a dummy variable, if we do not replace it with a non-empty string value, we will always get the empty string as the result. However, binding an existing variable to a dummy variable with an empty string as its name will cause problem when checking the convertibility of terms. Therefore, for the sake of valid variable renaming, we must replace the empty string with an non-empty constant to get a valid name.

$\texttt{varsCont}(())$	$=$	`[]`
$\texttt{varsCont}((\Gamma, x : \_))$	$=$	$x : \texttt{varsCont}(\Gamma)$
$\texttt{varsCont}((\Gamma, x : \_ = \_))$	$=$	$x : \texttt{varsCont}(\Gamma)$
$\texttt{freshVar}(\varepsilon, ss)$	$=$	$\texttt{freshVar}(\texttt{'var'}, ss)$ (ε represents the empty string)
$\texttt{freshVar}(s, ns)$	$=$	if $s \in ns$ then $\texttt{freshVar}(s', ns)$ else s (s' means append s with an apostrophe character)

Table 3.7: Functions: varsCont, freshVar

The locking/unlocking mechanism in our system is implemented via a concept called *lock strategy* plus a function called `getEnv`:

- `getEnv` :: `LS` → `Cont` → `Env`: given a lock strategy, extract an environment from the context. (`LS` represents the type of lock strategy, `Cont` the type of context and `Env` the type of environment.)

The idea is that when we lock a constant, we need to remove its definition from the environment, such that when evaluated, this constant becomes a neutral value, cutting off all the possibilities for further evaluation; When we unlock the constant later, we need to restore its definition to the environment.

During the type checking process, the context Γ is always extended with all the definitions declared so far. By the function `getEnv` and a lock strategy s that represents our intention about the locking/unlocking condition of each variable, we can conveniently get the environment ρ that effectuates our locking strategy.

In the current implementation, we have 4 lock strategies: `LockAll, LockNone, LockList vs, UnLockList vs`, where `vs` is a list of variables. By referring to these four strategies, we give the definition of `getEnv` in table 3.8.

$\texttt{getEnv}(\texttt{LockAll}, \Gamma)$	$=$	$()$
$\texttt{getEnv}(\texttt{LockNone}, ())$	$=$	$()$
$\texttt{getEnv}(\texttt{LockNone}, (\Gamma, x : A))$	$=$	$\texttt{getEnv}(\texttt{LockNone}, \Gamma)$
$\texttt{getEnv}(\texttt{LockNone}, (\Gamma, x : A = B))$	$=$	let ρ = $\texttt{getEnv}(\texttt{LockNone}, \Gamma)$
		in $(\rho, x : A = B)$
$\texttt{getEnv}(\texttt{LockList vs}, ())$	$=$	$()$
$\texttt{getEnv}(l\texttt{@(LockList vs)}, (\Gamma, x : A))$	$=$	$\texttt{getEnv}(l, \Gamma)$
$\texttt{getEnv}(l\texttt{@(LockList vs)}, (\Gamma, x : A = B))$	$=$	let $\rho = \texttt{getEnv}(l, \Gamma)$
		in if $x \in vs$ then ρ
		else $(\rho, x : A = B)$
$\texttt{getEnv}(\texttt{UnLockList vs}, ())$	$=$	$()$
$\texttt{getEnv}(l\texttt{@(UnLockList vs)}, (\Gamma, x : A))$	$=$	$\texttt{getEnv}(l, \Gamma)$
$\texttt{getEnv}(l\texttt{@(UnLockList vs)}, (\Gamma, x : A = B))$	$=$	let $\rho = \texttt{getEnv}(l, \Gamma)$
		in if $x \notin vs$ then ρ
		else $(\rho, x : A = B)$

Table 3.8: Function: getEnv

During the type checking process, after a declaration is type checked, it is added to the underling type checking context. We denote the extension of a context by a declaration as

$$\Gamma \vdash x : A \Rightarrow (\Gamma, x : A)$$
$$\Gamma \vdash x : A = B \Rightarrow (\Gamma, x : A = B)$$

Table 3.9 lists out the judgments used during the type checking process. There, Γ is the type checking context and s is the lock strategy. Note that the name collision check is performed before the type checking process, so we do not need to check the name uniqueness of each constant in the declarations anymore.

checkDecl	$\Gamma, s \vdash D \Rightarrow \Gamma'$	D is a correct declaration and extends Γ to Γ'
checkInferT	$\Gamma, s \vdash M \Rightarrow t$	M is a correct expression and its type is inferred to be t
checkWithT	$\Gamma, s \vdash M \Leftarrow t$	M is a correct expression given type t
checkEqualInferT	$\Gamma, s \vdash u \equiv v \Rightarrow t$	u, v are convertible and their type is inferred to be t
checkEqualWithT	$\Gamma, s \vdash u \equiv v \Leftarrow t$	u, v are convertible given type t

Table 3.9: Type Checking Judgments

3.5.2 checkDecl

$$\frac{\Gamma, s \vdash A \Leftarrow U}{\Gamma, s \vdash x : A \Rightarrow \Gamma_1} \tag{3.3}$$

$$\frac{\Gamma, s \vdash A \Leftarrow U \qquad \Gamma, s \vdash B \Leftarrow t}{\Gamma, s \vdash x : A = B \Rightarrow \Gamma_1} \begin{pmatrix} \rho & = & \mathtt{getEnv}(s, \Gamma) \\ t & = & [\![A]\!]\rho \end{pmatrix} \tag{3.4}$$

For a declaration $x : A$, we check that A is valid and has type U; For a definition $x : A = B$, we check further that B has type t, which is the q-expression of A evaluated in the environment extracted by applying function `getEnv` to s and Γ.

3.5.3 checkInferT

$$\frac{}{\Gamma, s \vdash U \Rightarrow U} \tag{3.5}$$

$$\frac{}{\Gamma, s \vdash x \Rightarrow t} \begin{pmatrix} \rho & = & \mathtt{getEnv}(s, \Gamma) \\ A & = & \Gamma(x) \\ t & = & [\![A]\!]\rho \end{pmatrix} \tag{3.6}$$

U has itself as its type; A variable x is well typed when there is a type bound to it in Γ.

$$\frac{\Gamma, s \vdash M \Rightarrow \langle [x : A]B, \rho \rangle \qquad \Gamma, s \vdash N \Leftarrow va}{\Gamma, s \vdash M\ N \Rightarrow v^*} \begin{pmatrix} va & = & [\![A]\!]\rho \\ \rho_0 & = & \mathtt{getEnv}(s, \Gamma) \\ vn & = & [\![N]\!]\rho_0 \\ \rho_1 & = & (\rho, x = vn) \\ v^* & = & [\![B]\!]\rho_1 \end{pmatrix} \tag{3.7}$$

For application $M\ N$, we do as follows

1. Check M is valid and its type can be inferred to be of the form $\langle [x : A]B, \rho\rangle$.
2. Check N has the right type to be applied to M.
3. Get the environment extracted from the current context Γ, denote it as ρ_0.
4. Get the q-expression of N evaluated from ρ_0, denote it as vn.
5. Extend ρ to ρ_1 by binding x to vn.
6. Return the q-expression of B evaluated in ρ_1.

$$\frac{\Gamma, s \vdash x : A = B \Rightarrow \Gamma_1 \qquad \Gamma_1, s \vdash M \Rightarrow t}{\Gamma, s \vdash [x : A = B]\, M \Rightarrow t} \tag{3.8}$$

For expression in the form of a *let* clause $[x : A = B]\, M$, we first check the definition is correct, then infer the type of M under the new context.

3.5.4 checkWithT

$$\frac{}{\Gamma, s \vdash U \Leftarrow U} \tag{3.9}$$

$$\frac{\Gamma, s \vdash x \Rightarrow v' \qquad \Gamma, s \vdash v \equiv v' \Rightarrow \_}{\Gamma, s \vdash x \Leftarrow v} \tag{3.10}$$

As we have already known, U has U as its type; To check that a variable x has type v, we first infer the type of x as v', then we check that v' and v are convertible.

$$\frac{\Gamma, s \vdash M\ N \Rightarrow v' \qquad \Gamma, s \vdash v' \equiv v \Rightarrow \_}{\Gamma, s \vdash M\ N \Leftarrow v} \tag{3.11}$$

$$\frac{\Gamma, s \vdash x : A \Rightarrow \Gamma_1 \qquad \Gamma_1, s \vdash B \Leftarrow U}{\Gamma, s \vdash [x : A]\, B \Leftarrow U} \tag{3.12}$$

To check that an application $M\ N$ has type v, we first infer its type v', then we check that v and v' are convertible; To check that an abstraction $[x : A]\, B$ has type U, we first check that declaration $x : A$ is valid and extend Γ to Γ_1, then we check that B has type U in Γ_1.

$$\frac{\Gamma, s \vdash x : A \Rightarrow \Gamma_1 \qquad \Gamma, s \vdash va \equiv va' \Rightarrow \_ \qquad \Gamma_1, s \vdash B \Leftarrow vb'}{\Gamma, s \vdash [x : A]\, B \Leftarrow \langle [x' : A']\, B', \rho \rangle} \left(\begin{array}{lcl} \rho_0 & = & \texttt{getEnv}(s, \Gamma) \\ \rho_1 & = & (\rho, x' = x) \\ va & = & [\![A]\!]\rho_0 \\ va' & = & [\![A']\!]\rho \\ vb' & = & [\![B']\!]\rho_1 \end{array} \right) \tag{3.13}$$

To check that an abstraction $[x : A]\, B$ has a closure $\langle [x' : A']\, B', \rho \rangle$ as its type, we do as follows

1. Check declaration $x : A$ is valid and extend Γ to Γ_1.
2. Get the environment from the Γ, denote it as ρ_0.
3. Get the q-expression of A evaluated in ρ_0, denote it as va.
4. Get the q-expression of A' in ρ, denote it as va'.
5. Check that va and va' are convertible.
6. Extend ρ to ρ_1 by binding x' to x.
7. Get the q-expression of B' evaluated in ρ_1, denote it as vb'.
8. Check that B has type vb' in context Γ_1.

$$\frac{\Gamma, s \vdash x : A = B \Rightarrow \Gamma_1 \qquad \Gamma_1, s \vdash M \Leftarrow t}{\Gamma, s \vdash [x : A = B]\, M \Leftarrow t} \tag{3.14}$$

For an expression in the form of a *let* clause $[x : A = B]\, M$, we first check the definition $x : A = B$ is correct and extend Γ to Γ_1, then check that M has the required type in Γ_1.

3.5.5 checkEqualInferT

$$\frac{}{\Gamma, s \vdash U \equiv U \Rightarrow U} \tag{3.15}$$

$$\frac{x =:= y \qquad \Gamma, s \vdash x \Rightarrow v}{\Gamma, s \vdash x \equiv y \Rightarrow v} \tag{3.16}$$

The first rule states that U is equal to itself and has type U; The second states that a variable equals to itself and the type is inferred to be the q-expression of its bound type.

$$\frac{\Gamma, s \vdash M_1 \equiv M_2 \Rightarrow \langle [x : A]\, B, \rho \rangle \qquad \Gamma, s \vdash N_1 \equiv N_2 \Leftarrow va}{\Gamma, s \vdash (M_1\ N_1) \equiv (M_2\ N_2) \Rightarrow v} \left(\begin{array}{lcl} va & = & [\![A]\!]\rho \\ \rho_0 & = & \texttt{getEnv}(s, \Gamma) \\ vn & = & [\![N_1]\!]\rho_0 \\ \rho_1 & = & (\rho, x = vn) \\ v & = & [\![B]\!]\rho_1 \end{array} \right) \tag{3.17}$$

To check that two applications $M_1\ N_1$ and $M_2\ N_2$ are convertible and infer their type, we do as follows

1. Check M_1 and M_2 are convertible and has type in the form of a closure $\langle [x : A] B, \rho \rangle$.
2. Get the q-expression of A evaluated in the environment ρ, denote it as va.
3. Check N_1 and N_2 are convertible given va as their type.
4. Get the environment from the current context Γ, denote it as ρ_0.
5. Get the q-expression of N_1 evaluated in ρ_0, denote it as vn.
6. Extend ρ to ρ_1 by binding variable x to vn.
7. Return the q-expression of B evaluated in ρ_1 as the inferred type.

$$\frac{\Gamma, s \vdash \langle [x : A]\, B, \rho \rangle \equiv \langle [y : A']\, B', \rho' \rangle \Leftarrow U}{\Gamma, s \vdash \langle [x : A]\, B, \rho \rangle \equiv \langle [y : A']\, B', \rho' \rangle \Rightarrow U} \tag{3.18}$$

We check the convertibility of two closures by checking that they are convertible given type U. This inference rule is only used when two terms representing **types** are checked for convertibility[1]. In this case, the abstractions from the closures are always seen as elements of the type U, not as elements of types in the form of some other closures. This reflects a 'two-tier' type structure of our system: Only U and elements of U (in the form of an abstraction, as indicated by rule 3.12) are eligible to be used as types.

[1]Readers who are doubtful about this can check by going over the rules we present in this section.

3.5.6 CheckEqualWithT

$$\frac{\Gamma_1, s \vdash m \equiv n \Leftarrow vb}{\Gamma, s \vdash v1 \equiv v2 \Leftarrow \langle [x : A]\, B, \rho \rangle} \left(\begin{array}{lcl} va & = & [\![A]\!]\rho \\ ns & = & \texttt{varsCont}(\Gamma) \\ y & = & \texttt{freshVar}(x, ns) \\ \Gamma_1 & = & (\Gamma, y : va) \\ \rho_0 & = & \texttt{getEnv}(s, \Gamma) \\ m & = & [\![v1\ y]\!]\rho_0 \\ n & = & [\![v2\ y]\!]\rho_0 \\ \rho_1 & = & (\rho, x = y) \\ vb & = & [\![B]\!]\rho_1 \end{array} \right) \tag{3.19}$$

To check that two q-expressions $v1$ and $v2$ are convertible and has type $\langle [x : A]\, B, \rho \rangle$, we do as follows:

1. Generate a fresh variable y from the context Γ.
2. Extend ρ to ρ_1 with x bound to y .
3. Get the q-expression of B evaluated in ρ_1, denote it as vb.
4. Get the environment from the current context, denote it as ρ_0.
5. Evaluate application $(v1\ y)$ in ρ_0, denote the result as m.
6. Evaluate application $(v2\ y)$ in ρ_0, denote the result as n.
7. Get the q-expression of A evaluated in ρ, denote it as va.
8. Extend context Γ to Γ_1 with the new variable y typed with va.
9. Check that m, n are convertible in the context Γ_1 with vb given as the type.

This rule accommodates for η-conversion, where $\lambda x.f\,x$ and f can be checked to be convertible. This is the reason why we apply $v1$ and $v2$ with the new variable, and check the convertibility of the result. We generate a new variable and do variable renaming as a respect to principle 3 in section 3.2. Note that we do not replace each x in B to y manually, but add the binding $x = y$ to the environment in the closure and rely on the evaluation operation to achieve the desired effect.

$$\frac{\Gamma, s \vdash va_1 \equiv va_2 \Leftarrow U \qquad \Gamma_1, s \vdash vb_1 \equiv vb_2 \Leftarrow U}{\Gamma, s \vdash \langle [x_1 : A_1]\, B_1, \rho_1 \rangle \equiv \langle [x_2 : A_2]\, B_2, \rho_2 \rangle \Leftarrow U} \left(\begin{array}{lcl} va_1 & = & [\![A_1]\!]\rho_1 \\ va_2 & = & [\![A_2]\!]\rho_2 \\ ns & = & \mathtt{varsCont}(\Gamma) \\ y & = & \mathtt{freshVar}(x_1, ns) \\ \rho_{21} & = & (\rho_1, x_1 = y) \\ \rho_{22} & = & (\rho_2, x_2 = y) \\ vb_1 & = & [\![B_1]\!]\rho_{21} \\ vb_2 & = & [\![B_2]\!]\rho_{22} \\ \Gamma_1 & = & (\Gamma, y : va_1) \end{array} \right) \tag{3.20}$$

To check that two closures are convertible and has type U, we do as follows:

1. Get the q-expression of A_1 evaluated in ρ_1, denote it as va_1.
2. Get the q-expression of A_2 evaluated in ρ_2, denote it as va_2.
3. Check va_1 and va_2 are convertible given type U.
4. Generate a fresh variable y from the context Γ.
5. Extend ρ_1 to ρ_{21} with x_1 bound to y.
6. Extend ρ_2 to ρ_{22} with x_2 bound to y.
7. Get the q-expression of B_1 evaluated in ρ_{21}, denote it as vb_1.
8. Get the q-expression of B_2 evaluated in ρ_{22}, denote it as vb_2.
9. Extend context Γ to Γ_1 with the new variable y typed with va_1.
10. Check that vb_1, vb_2 are convertible in the context Γ_1 with U given as the type.

$$\frac{\Gamma, s \vdash v1 \equiv v2 \Rightarrow t' \qquad \Gamma, s \vdash t \equiv t' \Rightarrow \_}{\Gamma, s \vdash v1 \equiv v2 \Leftarrow t} \tag{3.21}$$

To check that in the general case, $v1$ and $v2$ are convertible given type t, we first check $v1$ and $v2$ are convertible and infer their type as t', then we check t and t' are convertible.

3.6 Locking Mechanism

As has been introduced, a locking mechanism in our system is realized by setting up a *lock strategy*, and use it to extract an environment from the underlying type checking context. The *environment* is the place where a variable is bound to its definition and the context in which the evaluation of an expression takes place. A variable without a bound q-expression evaluates to itself. In that case, it is a *neutral value* about which we know nothing and cannot be reduced further. We adjust the lock strategy so that the definition of a constant could be erased or restored from the environment. In this way, we effectively lock/unlock a variable.

This is a locking mechanism applied to definitions where a constant acts as a *locking unit*. The lock status of variables are independent of each other, meaning that locking/unlocking a constant does not entail other constants in its definition being locked/unlocked. An alternative is to apply locking on expressions, where we define a metric of computation such that during evaluation, only certain 'steps' of reductions are performed. We did not build this alternative in our system but will elaborate the idea more in section 3.7 when we talk about *head reduction*.

One application of our locking mechanism is that in a well typed context, find the minimum set of constants to be unlocked such that a declaration of a new constant could be type checked. The existence of such a minimum set dependents on two conditions: (i) The decidability of our type checking algorithm; (ii) The declaration of the new constant is well typed under the context. For condition (i), it relates to the metatheory of our system which we will not touch upon in this project, therefore we only take the assumption that it holds. Condition (ii) can be easily checked by type checking the declaration with all constants unlocked and see if it succeeds. Apart from existence, we also claim that once the minimum set exists, it is also unique. We give a proof of this in the following sections. One thing to note is that assuming the minimum set exists, there is always a trivial algorithm: one starts with all the constants in the context unlocked and locks the names one by one to see if it is needed. This algorithm is inefficient for there are potentially large number of constants that are irrelevant with the declaration being checked.

(A first attempt to find the algorithm starts from all constants being locked, whenever the type checking process cannot proceed, it tries to find a constant that causes the halt and unlocks that constant. It repeats this trail and error process until the constant is type checked. Compared with the trivial algorithm, it has the advantage that all irrelevant names are excluded from the beginning, thus more efficient. However, later there is a flaw discovered about the algorithm: there are cases where more than one constants are to be selected as the next constant to be unlocked and the algorithm cannot determine correctly which one to choose. I need to study the algorithm more carefully and present a solution (if not possible, an approximation) along with a proof in the final report.)

3.7 Head Reduction

We mentioned *head-reduction* earlier as an alternative to implement a locking mechanism on expressions. The intuition about head reduction is that it allows expressions to be evaluated step by step instead of being fully evaluated at one time. More precisely, head reduction defines a binary relation R on the set of all expressions E: for $a, b \in E$, if $\Gamma \vdash R(a, b)$ holds, then we say a is *head-reduced* to b. When incorporated into a locking mechanism, head reduction has the advantage that terms not fully evaluated could be checked for convertibility, thus giving the prospect that equality between two terms could be established with less computation. We give the definition of head reduction by a set of inference rules that describe the binary relation it defines on the expressions of our language.

$$\frac{}{\Gamma \vdash R(U, U)} \tag{3.22}$$

$$\frac{\Gamma \vdash R(A, A') \qquad \Gamma_1 \vdash R(M, M')}{\Gamma \vdash R([x : A]M, [x : A']M')} \begin{pmatrix} va & = & [\![A]\!]() \\ \Gamma_1 & = & (\Gamma, x : va) \end{pmatrix} \tag{3.23}$$

$$\frac{\Gamma_1 \vdash R(M, M')}{\Gamma \vdash R([x : A = B]M, [x : A = B]M')} \begin{pmatrix} \Gamma_1 & = & (\Gamma, x : A = B) \end{pmatrix} \tag{3.24}$$

$$\frac{}{\Gamma \vdash R(e, e')} \begin{pmatrix} ns & = & \texttt{varsCont}(\Gamma) \\ ve & = & \texttt{headRedV}(\Gamma, e) \\ e' & = & \texttt{readBack}(ns, ve) \end{pmatrix} \tag{3.25}$$

The rules above states that: For U, it head reduces to itself; For abstraction $[x : A]M$, if A head reduces to A' and M head reduces to M' when Γ is extended to Γ_1, then it head reduces to $[x : A']M'$; For a let clause $[x : A = B]M$, if M head reduces to M' in the extended context, then it head reduces to $[x : A = B]M'$; For expressions in the other form, the head reduction operation relies on two more primitive functions: `headRedV` and `readBack`, whereas `headRedV` relies further on the function `defVar`.

- `headRedV` :: `Cont` $\rightarrow$ `Exp` $\rightarrow$ `QE`: Evaluates an expression into a q-expression by a 'small' step under a given context. `Cont` is the type of context, `Exp` the type of expression and `QE` the type of q-expression.

- `readBack` :: `[String]` $\rightarrow$ `QE` $\rightarrow$ `Exp`: Transforms a q-expression back into an expression by eliminating all potential closures. A list of names taken from the underlying context is given as the first argument to avoid the name clashing problems.

- `defVar` :: `String` $\rightarrow$ `Cont` $\rightarrow$ `Exp`: Get the definition of a constant from the context.

This means that an expression e is first evaluated by a 'small' step, then read back to an expression e' (e' could be the same as e).

The definitions of `headRedV, readBack` and `defVar` are given in table 3.10. Notice how the empty environment ‘()’ is used in function `headRedV` to limit the steps of reductions performed.

$\texttt{headRedV}(\Gamma, x)$	$=$	let $M_x = \texttt{defVar}(x, \Gamma)$ in $[\![M_x]\!]()$
$\texttt{headRedV}(\Gamma, (e1\ e2))$	$=$	let $v1 = \texttt{headRedV}(\Gamma, e1)$, $v2 = [\![e2]\!]()$ in $\texttt{appVal}(v1, v2)$
$\texttt{readBack}(\_, U)$	$=$	U
$\texttt{readBack}(\_, x)$	$=$	x
$\texttt{readBack}(ns, (v1\ v2))$	$=$	let $e_1 = \texttt{readBack}(ns, v1)$, $e_2 = \texttt{readBack}(ns, v2)$ in $(e_1\ e_2)$
$\texttt{readBack}(ns, \langle[\varepsilon : A]B\rangle\rho)$	$=$	let $A' = \texttt{readBack}(ns, [\![A]\!]\rho)$, $B' = \texttt{readBack}(ns, [\![B]\!]\rho)$ in $[\varepsilon : A']B'$
$\texttt{readBack}(ns, \langle[x : A]B\rangle\rho)$	$=$	let $y = \texttt{freshVar}(x, ns)$, $va = [\![A]\!]\rho$, $\rho_1 = (\rho, x = y)$, $vb = [\![B]\!]\rho_1$, $A' = \texttt{readBack}(ns, va)$, $B' = \texttt{readBack}((y : ns), vb)$, in $[y : A']B'$
$\texttt{defVar}(x, ())$	$=$	x
$\texttt{defVar}(x, (\Gamma, x' : \_))$	$=$	if $x == x'$ then x else $\texttt{defVar}(x, \Gamma)$
$\texttt{defVar}(x, (\Gamma, x' : \_ = M))$	$=$	if $x == x'$ then M else $\texttt{defVar}(x, \Gamma)$

Table 3.10: Functions: headRedV, readBack, defVar

As an example of head reduction, we present below in the listing 3.1 the result of applying head reduction to a constant named ‘`loop`’ which we take from a file written in our language (see appendix A.3). The file represents a variation of Hurkens paradox [8] and is used as a test case for our program. There, evaluating the constant ‘`loop`’ in an environment with all constants unlocked will cause the program to loop forever. However, we can use head reduction to show the results of the first few steps of evaluation.

Listing 3.1: Results of Head Reduction on the Constant Loop

```
step 1:
lem2 lem3

step 2:
lem3 B lem1 ([ p : Pow U ] lem3 ([ z : U ] p (delta z)))

step 3:
lem1 C ([ x : U ] lem1 (delta x))
```

```
  ([ p : Pow U ] lem3 ([ z : U ] p (delta z)))

step 4:
lem3 ([ z : U ] B (delta z)) ([ x : U ] lem1 (delta x))
  ([ p : Pow U ] lem3 ([ z : U ] p
    (delta (delta z))))

step 5:
lem1 (delta C) ([ x : U ] lem1 (delta (delta x)))
  ([ p : Pow U ] lem3 ([ z : U ] p
    (delta (delta z))))

step 6:
lem3 ([ z : U ] B (delta (delta z)))
  ([ x : U ] lem1 (delta (delta x)))
    ([ p : Pow U ] lem3 ([ z : U ] p
      (delta (delta (delta z)))))

step 7:
lem1 (delta (delta C)) ([ x : U ] lem1
  (delta (delta (delta x))))
    ([ p : Pow U ] lem3 ([ z : U ]
      p (delta (delta (delta z)))))
```

4

Extension

5

Results

6

Conclusion

Bibliography

[1] G. Huet, G. Kahn, and C. Paulin-Mohring, "The coq proof assistant a tutorial," *Rapport Technique*, vol. 178, 1997.

[2] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *International Conference on Automated Deduction*, pp. 378–388, Springer, 2015.

[3] U. Norell, "Dependently typed programming in agda," in *International school on advanced functional programming*, pp. 230–266, Springer, 2008.

[4] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation.," *J. Funct. Program.*, vol. 23, no. 5, pp. 552–593, 2013.

[5] U. Berger, M. Eberl, and H. Schwichtenberg, "Normalization by evaluation," in *Prospects for Hardware Foundations*, pp. 117–137, Springer, 1998.

[6] H. P. Barendregt *et al.*, *The lambda calculus*, vol. 3. North-Holland Amsterdam, 1984.

[7] N. G. De Bruijn, "A survey of the project automath," in *Studies in Logic and the Foundations of Mathematics*, vol. 133, pp. 141–161, Elsevier, 1994.

[8] A. J. Hurkens, "A simplification of girard's paradox," in *International Conference on Typed Lambda Calculi and Applications*, pp. 266–278, Springer, 1995.

A

Appendix

A.1 Haskell Source Code

A.2 Concrete Syntax

```
position token Id ((char - ["\\\n\t[]():;,.0123456789 "])
  (char - ["\\\n\t[]():;,. "])*);

entrypoints Context, Exp, Decl;

Ctx. Context ::= [Decl] ;

U.        Exp2 ::= "*" ;
Var.      Exp2 ::= Ref ;
SegVar.   Exp2 ::= Ref "[" [Exp] "]" "." Id ;
App.      Exp1 ::= Exp1 Exp2 ;
Arr.      Exp  ::= Exp1 "->" Exp ;
Abs.      Exp  ::= "[" Id ":" Exp "]" Exp ;
Let.      Exp  ::= "[" Id ":" Exp "=" Exp "]" Exp ;

Dec.      Decl ::= Id ":" Exp ;
Def.      Decl ::= Id ":" Exp "=" Exp ;
Seg.      Decl ::= Id "=" "seg" "{" [Decl] "}" ;
SegInst.  Decl ::= Id "=" Ref "[" [Exp] "]" ;

Ri.       Ref  ::= Id ;
Rn.       Ref  ::= Ref "." Id ;

separator Decl ";" ;

separator Exp "," ;

coercions Exp 3;
```

```
layout "seg";

layout toplevel;

comment "--";

comment "{-" "-}";
```

A.3 Test Case

```
Pow : * -> * =
  [X : *] X -> * ;

T : * -> * =
  [X : *] Pow (Pow X) ;

⊥ : * = [X : *] X ;

funT : [X : *] [Y : *] (X -> Y) -> T X -> T Y =
  [X : *][Y : *][f : X -> Y][t : T X][g : Y -> *] t ([x : X] g (f x)) ;

¬ : * -> * =
  [X : *] X -> ⊥ ;

U : * = [X : *] (T X -> X) -> T X ;

tau : T U -> U =
  [t : T U][X : *][f : T X -> X] [p : Pow X] t ([x : U] p (f (x X f))) ;

sigma : U -> T U =
  [z : U] z U tau ;

delta : U -> U = [z : U] tau (sigma z) ;

Q : T U =
  [p : U -> *][z : U] sigma z p -> p z ;

B : Pow U =
  [z : U] ¬ ([p : Pow U] sigma z p -> p (delta z)) ;

C : U = tau Q ;

lem1 : Q B
  = [z : U] [k : sigma z B] [l : [p : Pow U] sigma z p -> p (delta z)] l B k ([p :

A : * = [p : Pow U] Q p -> p C ;
```

```
lem2 : ¬ A
  = [h : A] h B lem1 ([p : Pow U] h ([z : U] p (delta z))) ;

lem3 : A
  = [p : Pow U] [h : Q p] h C ([x : U] h (delta x)) ;

loop : ⊥
  = lem2 lem3 ;

delta2 : U -> U = [z : U] delta (delta z) ;
```