A Haskell Implementation for a Dependent Type Theory with Definitions

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 Dependent Type Theory with Definitions

QUFEI WANG

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

A Haskell Implementation for a Dependent Type Theory with Definitions
QUFEI WANG

© QUFEI WANG, 2021.

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 Dependent Type Theory with Definitions
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. This language could be viewed as a pure λ-calculus extended with dependent types and definitions. The focus of this project is on the study of a definition mechanism where the definitions of constants could be handled efficiently during the type checking process. We later enrich the language with a module system to study how the definition mechanism should be adjusted for the introduction of the concept namespace on 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 users 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 proof assistant systems 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, November 2021

Contents

List of Tables

1

Introduction

1.1 Background

Dependent type theory originated in the work of AUTOMATH[1] initiated by N.G. de Bruijn in the 1960s. Since then it has lent much of its power to proof assistant systems like Coq[2] and Agda[3], 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 $\Pi_{x:A}B(x)$ 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 strongly 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 only a weak type system. Just as the famous saying puts it "well-type programs cannot 'go wrong'" [16]. It is in this sense we say that languages with dependent types are equipped with the highest level of correctness and precision, which makes them a natural option for building proof assistant systems.

The downside of dependent type systems, however, lies in the difficulties of their implementation. One major difficulty is checking the convertibility of terms, that is, given two terms A and B, decide whether they are equal or not. Checking the convertibility of terms that represent types is a frequently performed task by the type checker of any typed language, the way it is conducted directly affects the performance of the language. 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: there's no way for other terms in Java to be reduced to the term *int*. In a dependently typed language, however, the problem is more complex since a type may contain any expression as its component and deciding the convertibility of types in this case entails evaluation on expressions, which could incur much more computation.

1.2 Aim

The aim of this project is to study how to present definitions in dependent type theory. More specifically, we study how to do type checking in dependent type theory with the presence of definitions. A definition in dependent type theory is a declaration of the form $x : A = B$, meaning that x is a constant of type A, defined as B, where A, B are expressions of the language. The subtlety about definition in a dependent type theory is that when checking the convertibility of terms, sometimes the definition of a constant is indispensable while other times erasing the definition helps to improve efficiency by cutting off unnecessary computation. Suppose we have a definition of the exponentiation operation 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` represents the type of natural number and $*, -$ represent the multiplication and subtraction operations on natural numbers respectively. When checking the convertibility of two terms `expo` 2 10 and 1024, the definition of `expo` is necessary to reduce the first to 1024. However, if the terms are changed to `expo` $(1+1)$ 10 and `expo` 2 $(5+5)$, instead of using the definition of `expo` to reduce both terms to 1024, we could keep `expo` **locked** and only reduce both sides to the term `expo` 2 10. By showing that they can be reduced to a common term (having the same symbolic representation), we can prove their equality with much less computation. In the proof assistant system Agda, type checking relies on the algorithm of conversion checking which has increasingly become the bottleneck for type checking large programs. Sometimes type checking simply becomes too slow, yet it is not clear how to tackle this problem. A stable solution to this problem is desired so that one feasible example does not become infeasible upon the upgrade of the convertibility checking algorithm. Our work studying the role of definitions in dependent type theory is an attempt to address this problem.

The first part of this project consists of the specification of a dependently typed language which features a definition mechanism where constants could be manually locked/unlocked during the type checking process. The idea of making some constants locked to improve the efficiency of conversion checking is inspired by one unpublished work of Bruno Barras[4]. One system, named as *cubicaltt*[1] and implemented by Thierry Coquand, Anders Mörtberg et al., contains a similar locking/unlocking mechanism but there the feature was not documented in detail and what we present in this project is also a clearer way of implementation. The first part contains the main theoretical results of this project, and a thorough exposition of the definition mechanism is given in section 2.5. As an application of the definition mechanism, we built in the second part a module system based on the concept

[1]See the website https://github.com/mortberg/cubicaltt

"segment", which is borrowed from the work AUTOMATH[5]. The adaptation to the concept of namespace introduced by the module system could be seen as an evidence of the scalability of our definition mechanism.

1.3 Organization

This paper is organized as follows: chapter 2 starts with three examples to illustrate the common pitfalls one should avoid in the implementation of a dependent type theory. Based on the examples, we put forward two principles used as guidance in our own implementation. We then present in detail the syntax, semantics and type checking algorithm of this language and conclude this chapter with a thorough description of the definition mechanism. Chapter 3 starts with an introduction to the concept of segment and the relevant terminologies. This is followed by a detailed description on the syntax, semantics and type checking algorithm of the extended language. Chapter 4 presents a REPL program with commands to load and type check a source file of our language and experiment with the definition mechanism. Chapter 5 concludes the paper with a short review of this project.

1.4 Limitations

1. **Expressiveness:** The expressiveness of the language is intentionally restrained as an attempt to keep the language simple in order to focus on the study of a definition mechanism. As a consequence there is no language facility to create new data types.

2. **Metatheory:** Due to time limitations, a study on the properties of our language as a type theory and logic system will not be included. This could be seen as one of the directions of the future work.

2

Theory

Our system could be seen as an extension to λ-*calculus* with dependent types and definitions. In order for the reader to better understand the idea behind the choice of the syntax and semantics of our language, we first illustrate some subtleties of the system which suggest common pitfalls one should avoid in the implementation.

2.1 Subtleties of the System

We present the subtleties of the system by giving examples as the followings.

Example 1. Suppose we have declarations

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

Then the term below is well typed.

$$(\lambda u \ . \ u \ (u \ b))(\lambda z \ y \ x \ . \ a \ (z \ x) \ y) \tag{2.1}$$

If we do the reduction on (2.1), we get

$$\begin{aligned} (\lambda u \ . \ u \ (u \ b))(\lambda z \ y \ x \ . \ a \ (z \ x) \ y) &\implies \\ (\lambda z \ y \ x \ . \ a \ (z \ x) \ y)((\lambda z \ y \ x \ . \ a \ (z \ x) \ y) \ b) &\implies \\ (\lambda z \ y \ x \ . \ a \ (z \ x) \ y)(\lambda y \ x \ . \ a \ (b \ x) \ y) &\implies \\ \lambda y \ x \ . \ a \ ((\lambda y \ x \ . \ a \ (b \ x) \ y) \ x) \ y & \end{aligned} \tag{2.2}$$

At this point, we have a capture-of-variables problem. (2.2) 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 one reduces (2.2) naively without renaming, one gets

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

which is not correct. This example comes from the PhD thesis of L.S. van Benthem Jutting in 1977[6] when he was working on AUTOMATH. It was conjectured that if one starts with a term where each binding variable is declared only once and variables used in the terms forming an application are different, there will not be any capture of variables by reduction. This example shows that this is not the case and manifests an unusual case of the problem known as the capture of names by preforming reductions in λ-calculus.

Example 2. In non-dependent type languages like Java and Haskell, one can interpret the definitions by function application. For example, one can interpret the definition of $i2$ in following piece of Java code

```
int i1 = 0;
int i2 = i1 + 1;
```

by $i2 = (\lambda(x : int).x + 1)0$, and the definition of x in the following Haskell code

```
let x = 0 in x + x
```

by $(\lambda x.x + x)\,0$. This, however, is not always possible in a dependent type language. As an example, suppose we have

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

then the program

```
let x : A = a, y : P x in f y
```

cannot be rewritten to function application $(\lambda(x : A)(y : P\,x).f\,y)\,a$ because the former part of the formula, $\lambda(x : A)(y : P\,x).f\,y$, is not well-typed: the type of y is $P\,x$ whereas the type of the argument to f should be $P\,a$. This example shows that definitions in dependent type theory cannot be reduced to λ-*calculus*.

Example 3. Consider the formula

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

where **Nat** is the type of natural numbers and **Bool** is the type of Boolean. In this formula, the first declaration of x is shadowed by the second one. If we do not treat the shadowing of names properly, we may incorrectly conclude that we have a context where (1) the definition of y is x, (2) the type of y is **Nat**, whereas (3) the type of x is **Bool**. This example shows that improper use and treatment of name shadowing leads to inconsistency.

Example 1 and 3 provide us with insights into two common pitfalls one should avoid in the implementation: (1) capture of names during reduction and (2) improper treatment of name shadowing. As a result, we put forward two principles as a measure to ward off these two traps.

Principle 1. Use closure to postpone reduction.

Principle 2. Forbid the practice of name shadowing.

Principle 1 comes as a measure to tackle the problem of capture of names. Here, a closure is a computation structure consisting of a function (λ-abstraction) and an *environment* where the environment is a structure which binds free variables from the function to expressions to be substituted. The idea of postponed reduction is that for a function application, the actual substitution is not performed until the body of the function is clear of abstractions. For example, consider an application $(f\ 0)$ on a function f defined as follows.

$$f = \lambda x \lambda y.x + y$$

By normal β-reduction, the result would be $\lambda y.0 + y$. But if we reduce it by using closure, the result would instead be $\langle \lambda y.x + y, (x = 0) \rangle$: a closure formed by the function $(\lambda y.x + y)$ and the environment $(x = 0)$. We do not perform substitution at this stage because the body of f is still a λ-abstraction. If we apply the result to another argument, say 1, because the body of the function is now free of abstractions, the substitutions for both x and y will be performed and the result would be $0{+}1 = 1$. The reason why the problem of variable capture can be avoided by using closure is that by deferring substitutions, the structure of the function body is well preserved and by the time the substitution really happens, only the variables that are originally bound in the body will be substituted by their binding terms. We will talk more about closure later when we introduce the semantics of our language in section 2.3. An example of using closures to evaluate the expression in 2.1 could be found in appendix A.1.

Principle 2 comes as a simple strategy to avoid the pitfall revealed by example 3. It means that during the type checking process, each declaration, including the declaration of binding variable in a λ-abstraction, is checked to ensure no name collision occurs. Another approach to the name shadowing problem is called *namespaced De Bruijn indices*[1], which is the technique currently adopted by Agda [2]. The idea is to decorate the variables declared with the same name with integer indices to tell them apart from each other. However, some experience with Agda shows that the context information inferred by Agda using indexed variables can be confusing at times. As an example, consider the following Agda program.

```
test : N → Bool → N
test = λ (x : N) →
  let y : N
      y = x
  in λ (x : Bool) → {!!}
let x : A = a, y : P x in f y
```

We ignore non-essential details but only illustrate the relevant points.

[1]For a detailed introduction, please visit the website: https://www.haskellforall.com/2021/08/namespaced-de-bruijn-indices.html.

[2]Tested with version 2.6.2.

- Line 1-6 is a definition of constant `test` which is a function that given a natural number and a Boolean, returns a natural number.
- Two variables with the same name x are declared, one in the outer scope at line 2, another in the inner scope at line 5.
- The x in the outer scope has the type natural number, whereas the x in the inner scope has the type Boolean.
- In the placeholder denoted by the text `{!!}`, using the interactive proof assistant feature provided by Agda, we get the following context information from Agda.

```
Goal: ℕ
--------------------------------
y       : ℕ
y       = x₁
x       : Bool
x = x₁  : ℕ    (not in scope)
```

What Agda means in this message is that:

- `y` is of type `ℕ` and is defined to be the `x` in the outer scope having index 1.
- `x` in the inner scope is of type `Bool` with no definition.
- `x` in the outer scope (indicated by the phrase "not in scope") is of type `ℕ` and is renamed to $\mathtt{x_1}$.

The message shows that Agda is able to keep track of variables declared with the same name correctly by labeling them with indices. However, the way it presents the context information can cause confusion for users who are not familiar with this feature: by reading the text $\mathtt{y = x_1}$, `x:Bool` and $\mathtt{x = x_1}$, one may wonder how it is possible for both y and x to be equal with x_1 since they have different types. Another inefficiency with this approach is that the end user cannot refer to the x in the outer scope by the name x_1 because x_1 is used as an internal identifier and there is no variable in the source file having the name x_1. Such an attempt will be rejected by Agda with an error message saying that "variable not in scope". In summary, we consider that allowing name shadowing will cause confusion in the context information and introduce ambiguities over the usage of names. For this reason, we simply forbid shadowing of names in our language.

2.2 Syntax of the Language

There are two kinds of syntax related to the language: (1) The concrete syntax that describes the grammar used in a source file, and (2) the abstract syntax translated from the concrete syntax for clarity and better presentation. What we are going to describe below is the abstract syntax, for the concrete syntax see appendix A.3.

The expressions in our language are defined as follows:

Definition 2.2.1 (Expression)

(i) U is an expression, which represents the universe of small types. U is an element of itself, i.e., $U \in U$.

(ii) A special group of terms, denoted as $\mathcal{K}$, are expressions and they are inductively defined by

(a) variables, e.g., $x, y, z \in \mathcal{K}$;

(b) terms of the form $(K\ M) \in \mathcal{K}$, where $K \in \mathcal{K}$ and M is an expression.

(iii) Given two expressions A, M and a variable x, a term of the form

$$[x : A]M$$

is an expression, which is used to represent

- **λ-abstraction**: $\lambda_{x:A}M$ - a function that given an argument x of type A, returns a term M which may depend on x;
- **Dependent Product**: $\Pi_{x:A}M$ - the type of function that given an argument x of type A, returns a term of type M which may depend on x. When M does not depend on x, we can ignore x and rewrite it as $\Pi_{\_:A}M$. This is essentially the same as the type of functions $A \to M$.

(iv) Given three expressions A, B, M and a variable x, a term of the form

$$[x : A = B]M$$

is an expression, which is used to represent a let clause:

- let $x : A = B$ in M.

The declarations in our language are defined as follows:

Definition 2.2.2 (Declaration)

(i) A term of the form $x : A$ is a declaration where x is a variable and A is an expression. It declares a variable x of the type represented by A.

(ii) A definition $x : A = B$ is a declaration where x is a variable and A, B are expressions. It declares a variable x of type A and is defined as B.

A program[3] in our language consists of a list of declarations. The name of a declaration must not collide with any name of the existing declarations and a variable

[3] A program can also be seen as a type checking context as described in section 2.4 since they both consist of a list of declarations.

must be declared before it is used. A summary of the syntax can be found in table 2.1, where A, M, K represent expressions; D represents definitions; $Decl$ represents declarations and P represents programs.

$$
\begin{array}{lcl}
A, M & ::= & U \mid K \mid [x : A]M \mid [D]M \\
K & ::= & x \mid K\,M \\
D & ::= & x : A = M \\
Decl & ::= & x : A \mid D \\
P & ::= & [Decl]
\end{array}
$$

Table 2.1: Syntax of the Language

The syntax of our language is a subset of Mini-TT[7]. 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 of 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 cause a problem in practice because the meaning of a term could be deduced from the context and our type checking algorithm ensures the consistency of its usage.

The classification of a subset of expressions denoted as $\mathcal{K}$ indicates that expressions in the language conform to the β-normal form, i.e., expressions of the form $U\,M$, $([x : A]M)\,E$ are considered illegal. The former is easy to understand as U is not a function; the latter is subject to β-reduction which is prohibitive in the language. We use this practice as a measure to keep the brevity of the type checking algorithm.

2.3 Operational Semantics

Given a well-formed expression, we describe in this section how it is evaluated in the semantics of our language. An expression is evaluated to a *quasi-expression* or *q-expression* in an *environment*, which is a stack structure in one of the following forms:

Definition 2.3.1 (Environment)

(i) $()$ is an empty environment;

(ii) $(\rho_1, x = q)$ extends the environment ρ_1 by binding the variable x to the q-expression q;

(iii) (ρ_1, D) extends the environment ρ_1 with a definition.

A *q-expression* is the intermediate form of an expression under evaluation. It can be transformed to a "normal" expression by a procedure called "readBack" which will be introduced later in section 2.5.1.

Definition 2.3.2 (q-expression)

(i) U is a q-expression.

(ii) A variable x is a q-expression, it represents a primitive without definition.

(iii) A closure $\langle [x : A]M, \rho\rangle$ is a q-expression: it is the result of evaluating the function $[x : A]M$ in the environment ρ.

(iv) Given two q-expressions k, q where k is not a closure, a term of the form $k\,q$ is a q-expression which represents an application that cannot be reduced further.

The grammar of q-expressions can be summarized in table 2.2.

$$\begin{array}{lcl} k & ::= & x \mid k\,q \\ q & ::= & U \mid k \mid \langle [x : A]M, \rho\rangle \end{array}$$

Table 2.2: Q-expressions

The evaluation function, given in table 2.3, is denoted by formulas of the form $M_\rho = q$, meaning that the expression M evaluates to q in the environment ρ.

$$\begin{array}{lcl} U_\rho & = & U \\ x_\rho & = & \rho(x) \\ (K\,N)_\rho & = & app(K_\rho, N_\rho) \\ ([x : A]B)_\rho & = & \langle [x : A]\,B, \rho\rangle \\ (D\,M)_\rho & = & M_{(\rho,D)} \end{array}$$

Table 2.3: Semantics of the Language

Two auxiliary functions are used in the evaluation, with their definitions given in table 2.4, 2.5 respectively.

- $\rho(x)$ finds the binding q-expression of the variable x in the environment ρ.
- $app(k, q)$ applies the function k to q.

$$\begin{array}{rcl} ()(x) & = & x \\ (\rho', x' = q)(x) & = & \text{if } x' == x \text{ then } q \text{ else } \rho'(x) \\ (\rho', x' : A = B)(x) & = & \text{if } x' == x \text{ then } B_{\rho'} \text{ else } \rho'(x) \end{array}$$

Table 2.4: Function $\rho(x)$

Some readers may have noticed that the real difference between expressions and q-expressions is the closure. A closure is an important concept in functional programming and was first conceived by P. J. Landin in his paper *The Mechanical Evaluation of Expressions*[8]. There, the author described a closure as "...comprising the λ-expression and the environment relative to which it was evaluated..." which

$$\begin{array}{rcl} app(\langle [x:A]M, \rho \rangle, q) & = & M_{(\rho, x=q)} \\ app(k, q) & = & k\, q \end{array}$$

Table 2.5: Function $app(k, q)$

specified the structure of the closures we adhere to in our own implementation. Closure is introduced to meet the need of passing functions around during evaluation, and entails the introduction of q-expression as a parallel but distinct concept from expression. One major benefit brought by using closure is the ability to defer computation.

The meaning of deferred computation comes into twofold: first, the evaluation of the reducible expressions in the function body is deferred, as signified by the rule about evaluation of functions in table 2.3 where the function body is left intact; second, the substitution process in β-reduction is deferred as indicated by the definition of the function *app* in table 2.5. For an application of the function $[x : A]M$ to an argument q, the substitution will not happen until M is clear of abstractions. The ability to defer computation is crucial for the definition mechanism as it makes it possible for saving computations during the evaluation process.

2.4 Type Checking Algorithm

The aim of the type checking algorithm is to ensure that a program in our language is well-typed. Basically, for a declaration in the form $x : A$, it checks that A is a valid type, namely $A \in U$; for a declaration in the form $x : A = B$, it checks that (1) A is a valid type and (2) B is a well-typed expression and has type A. A program is said to be well-typed when each of its declaration is well-typed.

Note that the type checking algorithm does not concern with any syntactic or semantic error related with names, such as duplicated declaration of names or use of undeclared names. Syntactic errors are checked by the lexer and parser where a source file is parsed into a concrete syntax tree. Semantic error with regard to the use of names are checked when the concrete syntax tree is translated to an abstract syntax tree in a procedure called *translation*. It is on the abstract syntax tree that the type checkering algorithm is applied.

Given a *type checking context* Γ and a *lock strategy* s, the three forms of judgments used in the type checking algorithm can be given as the followings.

checkD	$\Gamma \vdash_s d \Rightarrow \Gamma'$	d is a valid declaration and extends Γ to Γ'
checkT	$\Gamma \vdash_s M \Leftarrow t$	M is a valid expression given type t.
checkI	$\Gamma \vdash_s K \Rightarrow t$	K is a valid expression and its type is inferred to be t.

Table 2.6: Type Checking Judgments

The lower case letter t represents a q-expression, meaning that the type inferred by *checkI* or given as an input in *checkT* must be an evaluated expression. A type checking context is a stack structure keeping track of the types and definitions of the variables and comes into one of the following three forms.

Definition 2.4.1 (Type Checking Context)

(i) $()$ is an empty context.

(ii) $(\Gamma_1, x : A)$ extends the context Γ_1 with a declaration $x : A$.

(iii) $(\Gamma_1, x : A = B)$ extends the context Γ_1 with a definition $x : A = B$.

In the type checking algorithm, Γ serves two main purposes: (1) provides the types of variables declared inside the context and (2) provides the environment customized by a lock strategy for evaluation.

A lock strategy is introduced as a part of our definition mechanism to provide the locking/unlocking functionality on constants. A constant is *locked* when its definition is temporarily erased and *unlocked* if restored. A locked constant is in effect a primitive variable that cannot be reduced further. Since environments are the place where variables are mapped to their definitions or q-expressions during evaluation, we can achieve the effect of locking/unlocking constants by removing/restoring their definitions from/to the environment. This suggests a procedure to transform a type checking context into an environment with the definitions of constants being erased or restored. We introduce a function *getEnv* for this purpose and denote it as ϱ in the following discussion. If we consider the symbol s in table as 2.6 being a list of locked variables, the function *getEnv* could be defined as in table 2.7.

$$
\begin{array}{lcl}
\varrho(s, ()) & = & () \\
\varrho(s, (\Gamma, x : A)) & = & \varrho(s, \Gamma) \\
\varrho(s, (\Gamma, x : A = B)) & = & \text{let } \rho = \varrho(s, \Gamma) \text{ in if } x \in s \text{ then } \rho \text{ else } (\rho, x : A = B)
\end{array}
$$

Table 2.7: Function getEnv

Given a type checking context Γ and a lock strategy s, we can get the evaluated form of the type of the variable x by the function *getType*. We denote this function as $\Gamma(s, x)$ and give its definition in table 2.8.

$$
\begin{array}{rcl}
()(s, x) & = & \mathit{error} \\
(\Gamma', x' : A)(s, x) & = & \text{if } x' == x \text{ then } A_{\varrho(s, \Gamma')} \text{ else } \Gamma'(s, x) \\
(\Gamma', x' : A = B)(s, x) & = & \text{if } x' == x \text{ then } A_{\varrho(s, \Gamma')} \text{ else } \Gamma'(s, x)
\end{array}
$$

Table 2.8: Function getType

In the type checking process, the convertibility of terms is expressed by a predicate *checkConvert* which given a list of names, decides whether two q-expressions are convertible. We use the notation $q_1 \sim_{ns} q_2$ to express that q_1 and q_2 are convertible. ns is a list of names and is used to ensure that names newly introduced in the convertibility checking process do not collide with the names already existing in the underlying type checking context. The definition of *checkConvert* is given in table 2.9. Note that the rules presented here only check β-convertibility, for η-convertibility please refer to appendix A.2.

$$\frac{}{U \sim_{ns} U} \qquad \frac{}{x \sim_{ns} x}$$

$$\frac{k_1 \sim_{ns} k_2 \qquad q_1 \sim_{ns} q_2}{k_1\, q_1 \sim_{ns} k_2\, q_2}$$

$$\frac{A_\rho \sim_{ns} A'_{\rho'} \qquad M_{(\rho, x=y)} \sim_{y:ns} M'_{(\rho', x'=y)}}{\langle [x : A]M, \rho \rangle \sim_{ns} \langle [x' : A']M', \rho' \rangle} \text{ where } y \text{ is a new variable}$$

Table 2.9: Predicate checkConvert

The function *namesCtx*, denoted as $\tau(\Gamma)$, is used to get the names from the context Γ. We use this function to provide the list of names used by *checkConvert*.

2.4.1 checkD

$$\frac{\Gamma \vdash_s A \Leftarrow U}{\Gamma \vdash_s x : A \Rightarrow (\Gamma, x : A)} \tag{2.3}$$

$$\frac{\Gamma \vdash_s A \Leftarrow U \qquad \Gamma \vdash_s B \Leftarrow A_{\varrho(s,\Gamma)}}{\Gamma \vdash_s x : A = B \Rightarrow (\Gamma, x : A = B)} \tag{2.4}$$

2.4.2 checkT

$$\frac{}{\Gamma \vdash_s U \Leftarrow U} \tag{2.5}$$

$$\frac{\Gamma(s,x) \sim_{\tau(\Gamma)} t}{\Gamma \vdash_s x \Leftarrow t} \tag{2.6}$$

$$\frac{\Gamma \vdash_s K\,N \Rightarrow t' \qquad t' \sim_{\tau(\Gamma)} t}{\Gamma \vdash_s K\,N \Leftarrow t} \tag{2.7}$$

$$\frac{\Gamma \vdash_s A \Leftarrow U \qquad (\Gamma, x : A) \vdash_s B \Leftarrow U}{\Gamma \vdash_s [x : A]B \Leftarrow U} \tag{2.8}$$

$$\frac{\Gamma \vdash_s A \Leftarrow U \qquad A_{\varrho(s,\Gamma)} \sim_{\tau(\Gamma)} A'_{\rho'} \qquad (\Gamma, x : A) \vdash_s B \Leftarrow B'_{(\rho', x'=x)}}{\Gamma \vdash_s [x : A]\,B \Leftarrow \langle [x' : A']B', \rho' \rangle} \tag{2.9}$$

$$\frac{\Gamma \vdash_s A \Leftarrow U \qquad \Gamma \vdash_s B \Leftarrow A_{\varrho(s,\Gamma)} \qquad (\Gamma, x : A = B) \vdash_s M \Leftarrow t}{\Gamma \vdash_s [x : A = B]\,M \Leftarrow t} \tag{2.10}$$

Note that the inference rules 2.8 and 2.9 differentiate between the use of an abstraction $[x : A]B$ as a dependent product or as a function. When used as a dependent product, its type is U; otherwise, its type is a closure.

2.4.3 checkI

$$\frac{}{\Gamma \vdash_s x \Rightarrow \Gamma(s,x)} \tag{2.11}$$

$$\frac{\Gamma \vdash_s K \Rightarrow \langle [x : A]B, \rho \rangle \qquad \Gamma \vdash_s N \Leftarrow A_\rho}{\Gamma \vdash_s K\,N \Rightarrow B_{(\rho, x=n)}} \left(\; n = N_{\varrho(s,\Gamma)} \; \right) \tag{2.12}$$

2.5 Definition Mechanism

The motivation to build a definition mechanism is to study how to do type checking in the presence of definitions in dependent type theory. In any typed language, one basic problem a type checker should be able to solve is to decide, given an expression E and a type A, whether E is of type A. Usually this involves getting the type of E, say T, by means of computation regarding the composition of E and decide whether T and A are convertible. Some difficulties arise in dependent type theories because (1) a type may contain **any** expression which could entail large amount of computation, and (2) the use of definitions opens up the possibility to denote arbitrary complex computation by a single constant. For a type checker of dependent type theory to be efficient, the amount of computation it performed in the convertibility checking should not exceed too much what are "just enough" to establish the equivalence of the checked terms. The problem is that there is no standard way to calculate the minimum number of reductions needed because it depends on the semantics, namely the language designer's perception of computation, of the language. For example, consider again the two formulae $(1+1)^{10}$ and $2^{(5+5)}$. To check the convertibility of these two terms, if we adopt the common arithmetic definition of integer multiplication and exponentiation, and determine that any expression should be evaluated to its normal form (no redex exists), a type checker loyal to our conception of computation will reduce both terms to 1024. However, if we change our mind and see exponentiation as a primitive with no definition, the same type checker with our updated conception will only reduce both terms to 2^{10}.

Our definition mechanism is an attempt to improve the performance of convertibility checking by setting limits on constants. That is, a constant acts as a unit on which computation could be locked or charged. More advanced computation control techniques with finer granularity is desired, as can be shown by the following example which is a variant of the example above. Consider these two formulae $2 * 2^9$ and 2^{10}. In this case, locking the definition of exponentiation will not work. One solution for this problem is to recognize and utilize the property about exponentiation $2^m * 2^n = 2^{(m+n)}$. Another way is to reduce 2^{10} to $2 * 2^9$ using the definition of exponentiation only once. The former suggests a mechanism to establish properties about data types and constants, and use these properties in the following computations, a technique that has been adopted by Haskell and proof assistant systems like Agda; the latter indicates a dynamic change of the evaluation strategy in the process of computation, a hint for more advanced intelligence for the program. Although in this work we didn't go further towards either of the two directions, we do studied and implemented a function called "linear head reduction" which could limit the computations performed on expressions in each reduction step.

2.5.1 Linear Head Reduction

Linear head reduction was introduced in the calculus $\Delta\Lambda$ of AUTOMATH[9] and is demonstrated here to show another way to limit computation. It relies on two procedures: (1) the procedure to force the subset of expressions $\mathcal{K}$ to be evaluated in "small steps" once a time instead of being fully evaluated; (2) the procedure to eliminate closures so that the result of head reduction is an expression instead of a quasi-expression. The first procedure is named *headRedV* and denoted by δ^*, its definition is given in table 2.10.

$$
\begin{array}{lcl}
\delta^*(\Gamma, x) & = & \mathcal{V}(\Gamma, x) \\
\delta^*(\Gamma, K\,N) & = & \text{let } k = \delta^*(\Gamma, K),\ n = N_{()} \text{ in } app(k, n)
\end{array}
$$

Table 2.10: Function headRedV

The empty parentheses represents an empty environment. $\mathcal{V}(\Gamma, x)$ is the function that gets the least evaluated form of variable x from context Γ. We call it *getVal* and give its definition in table 2.11. Note that to reduce an application $K\,N$, our approach is different with the one adopted by a Krivine machine[4]: instead of evaluating both the body and the argument of a function within a given environment ρ (i.e., $(K_\rho\,N_\rho)$), we only unfold the body but do not distribute the environment to the argument.

$$\begin{array}{lcl}
\mathcal{V}((), x) & = & x \\
\mathcal{V}((\Gamma', x' : A), x) & = & \text{if } x' == x \text{ then } x \text{ else } \mathcal{V}(\Gamma', x) \\
\mathcal{V}((\Gamma', x' : A = B), x) & = & \text{if } x' == x \text{ then } B_{()} \text{ else } \mathcal{V}(\Gamma', x)
\end{array}$$

Table 2.11: Function getVal

The second procedure is named *readBack* and denoted by $\mathcal{R}$. Given a list of names and a q-expression, it eliminates all the closures in the q-expression to transform it into an expression. The definition of *readBack* is given in table 2.12.

$$\begin{array}{lcl}
\mathcal{R}(_, U) & = & U \\
\mathcal{R}(_, x) & = & x \\
\mathcal{R}(ns, k\ q) & = & \text{let } K = \mathcal{R}(ns, k),\ N = \mathcal{R}(ns, q) \text{ in } K\,N \\
\mathcal{R}(ns, \langle [x : A]B, \rho \rangle)) & = & \text{let } y = \nu(ns, x),\ A' = \mathcal{R}(ns, A_\rho), \\
 & & B' = \mathcal{R}(y : ns, B_{(\rho, x=y)}) \text{ in } [y : A']B'.
\end{array}$$

Table 2.12: Function readBack

Finally, the definition of *linear head reduction* is given in table 2.13 where the function is denoted by δ.

$$\begin{array}{lcl}
\delta(\Gamma, U) & = & U \\
\delta(\Gamma, [x : A]M) & = & \text{let } M' = \delta((\Gamma, x : A), M) \text{ in } [x : A]M' \\
\delta(\Gamma, D\,M) & = & \text{let } M' = \delta((\Gamma, D), M) \text{ in } D\,M' \\
\delta(\Gamma, K) & = & \mathcal{R}(\tau(\Gamma), \delta^*(\Gamma, K))
\end{array}$$

Table 2.13: Function linear head reduction

As an example of linear head reduction, we apply this function continuously, first on a constant named "loop" from a source file of our language, later on the expression resulting from the last application, to see how the evaluation on the constant "loop" evolves. The source file is a variation of the Hurkens paradox[10] and is given in appendix A.5. The result of the first ten steps of head reduction are shown in appendix A.6 and one can see that there are patterns of terms recurring and replicating themselves as the evaluation goes further.

[4]Visit this website from wikipedia for an introduction.

2.5.2 Problem of Finding the Minimum Set of Constants

Regarding the definition mechanism, there is one conjecture from the unpublished work of Bruno Barras[4] stating that for any expression M of type N, there exists a minimum set of unfolded constants such that the type checking algorithm can check $M \in N$. Using the notation in section 2.4, this conjecture can be stated more formally as

Conjecture 1. Given a valid context Γ and two expressions M, N where M has type N, there exists a unique set of constants s_0 such that

$$\Gamma \vdash_s M \Leftarrow N \quad \textit{iff} \quad s \subseteq s_0$$

for an arbitrary set of locked constants s.

The idea of this conjecture comes from the fact that constants can be used as primitives to save computations in the conversion checking of terms. During the type checking process, we wish to lock as many constants as possible to improve the performance without affecting the correctness of the type checking process. This could be illustrated by a simple example using the syntax of our language as follows.

```
id : * -> * = [A : *] A
T : *
t : T
------------------------
test1 : T = t
test2 : id T = t
```

In this short program, *id* is a constant that given any type A returns A itself. T is a primitive of type U and t is a primitive of type T. Suppose Γ is a context consisting of *id*, T and t and we run the type checking algorithm with Γ on the two new definitions $test1$ and $test2$. *test1* will always be type checked because $t \in T$; *test2*, however, will only be type checked when the constant *id* is unfolded, otherwise the type checking algorithm would consider the two terms T and *id* T not convertible. This is one case showing that the definition of a constant is necessary for the type checking algorithm to make correct judgments. On the other hand, if we change the declaration of t to `t:id T`, erase the declaration of *test1* and run the type checking algorithm with Γ on *test2* again, we can save the computation involving the expansion of the constant *id* and one beta reduction by treating *id* as a primitive.

This example shows that the motivation to find the minimum set of unfolded constants is clear: for large proof systems with nontrivial definitions of constants, unfolding constants and performing the ensuing reductions unwisely may cause huge loss of performance. If the conjecture holds and we have an efficient algorithm to this problem, we can achieve the highest possible performance in the current type checking algorithm.

In our attempt to prove this conjecture, noticing that it is possible to infer the type of any valid expression in the form of another expression, the conjecture can be reduced to

Conjecture 2. Given a valid context Γ and two semantically equivalent expressions M, N, there exists a unique set of constants s_0 such that

$$M_{\varrho(s,\Gamma)} \sim_{\tau(\Gamma)} N_{\varrho(s,\Gamma)} \quad \textit{iff} \quad s \subseteq s_0$$

for an arbitrary set of locked constants s.

The function to infer the type of any valid expression in the form of an expression is given in table 2.14, it uses the function *getType* (defined in table 2.8) and the function *readBack* (defined in table 2.12) to get the evaluated form of the type of a variable and transform it back into an expression. Also notice how we use s^*, the set of all constants from Γ, in the second case and the empty environment in the third case to avoid accidentally unlocking a constant by keeping all constants locked in the operation.

$\mathit{inferT}(\Gamma, U)$	$=$	U
$\mathit{inferT}(\Gamma, x)$	$=$	$\mathcal{R}(\tau(\Gamma), \Gamma(S, x))$,
		s^* represents the set of the constants from Γ
$\mathit{inferT}(\Gamma, K\, N)$	$=$	let $T = \mathit{inferT}(\Gamma, K)$ in $\mathcal{R}(\tau(\Gamma), \mathit{app}(T_{()}, N_{()}))$
$\mathit{inferT}(\Gamma, [x : A]M)$	$=$	let $M' = \mathit{inferT}((\Gamma, x : A), M)$ in $[x : A]M'$
$\mathit{inferT}(\Gamma, D\, M)$	$=$	let $M' = \mathit{inferT}((\Gamma, D), M)$ in $D\, M'$

Table 2.14: Function inferT

We can prove the conjecture provided the following properties hold for our system.

1. Each expression has a normal form.
2. We have a way to evaluate each expression M to its normal form step by step in sequence such that in each step
   (a) only one constant is unfolded.
   (b) the selection of the unfolded constant can be proved to lead to the minimum set.

If these two properties hold for our system, given two semantically equivalent expressions M, N, we can find the minimum set of unfolded constants by

1. Unfold M to its normal form resulting in a sequence of expressions $M_0, \ldots, M_p$ where $M_0 = M$.
2. Unfold N to its normal form resulting in a sequence of expressions $N_0, \ldots, N_q$ where $N_0 = N$.
3. Compare the syntactic identity of

$$(M_0, N_0), (M_1, N_0), (M_0, N_1), (M_1, N_1), (M_2, N_0), (M_2, N_1), (M_0, N_2) \ldots$$

until the first time we have $0 \leq p' \leq p$, $0 \leq q' \leq q$ such that $M_{p'} == N_{q'}$.

4. The minimum set of constants is the union of the constants unlocked from M_0 to $M_{p'}$, N_0 to N'_q. Because of the property 2.(b), this set is also unique.

Unfortunately, these two properties mentioned above do not hold in our system. For (1), we have the constant “loop” defined in appendix A.5 that does not have a normal form; for (2), when facing a term of the form $x\ y$, it is not clear whether we should unlock x or y so that the set of constants found at the end is minimal. In fact, we managed to find a counter-example for this conjecture in our system as follows.

```
k : * -> * -> * = [A : *][B : *]A
a : *
b : *
c : * = b
```

To check the convertibility of two expressions $k\,a\,b$ and $k\,a\,c$, we have two different minimum set of unlocked constants $\{k\}$ and $\{c\}$. This finding is important as it shows that there is no optimal strategy in general for our system and it is necessary for the user to provide a list of unfolded constants to help the type checker achieve higher type checking efficiency.

3

Extension

In chapter 2 we introduced and described a definition mechanism which features a locking/unlocking functionality on the constants of programs. In order to show that this mechanism is flexible and scalable to incorporate more language features, we introduce in this chapter a module system as an extension to the language and an enhancement to the definition mechanism. A module is a list of declarations and itself must be declared with a name by a declaration. The fact that modules can be nested suggests a modification to the semantics of the language such that a variable is no longer uniquely identified by its name but by its name and *namespace*, the nested structures of modules in which this variable is declared. The module system in the extended language is built on the idea 'segment' borrowed from the work of AUTOMATH. For an introduction to the usage of segments in AUTOMATH we refer the readers to H. Balsters's work[5]. In the following discussions, we use the words 'segment' and 'module' interchangeably and we first illustrate the concept of segment in our language by giving an example as the following.

Example 4. The idea of *segment* is to have a new form of declaration

$$\varsigma = \texttt{Seg}\ ds$$

where ς is the name of the segment and ds a list of declarations. The word `Seg` is designed as a language keyword and a segment can also be seen as a module with parameters. For example,

$$\varsigma = \texttt{Seg}\ [A : *,\ id : A \to A = [a : A]\ a]$$

is a module which contains a declaration and a definition. The declaration $(A : *)$ is a parameter of the module and the definition id is the identity function defined in this module. Segments can be instantiated by providing definitions to their parameters. Suppose we have another declaration $(A0 : *)$, then the segment ς can be instantiated by $(\varsigma\,[A0])$ and the expression $(\varsigma\,[A0]).id$ has $(A0 \to A0)$ as its type and closure $\langle [a : A]a, (A = A0)\rangle$ as its value.

A collection of the terminologies regarding the use of segments in the extended language is summarized as the following. These terminologies will be used in the description of the syntax, semantics and type checking algorithm of the extended language.

- **Segment:** A segment can be declared as $\varsigma = \texttt{Seg}\ ds$ where ς is the *name* of the segment and ds consisting of a list of declarations is the *content* of the segment.

- **Parent, Children:** Segments can be nested, i.e., a segment can be declared within another segment. The segment which contains other segments is called the *parent* and the segment(s) contained in a parent is(are) called the *child(children)*. We use the symbol $\rightarrow$ to denote the parent-child relation such that $a \rightarrow b$ iff a is the parent of b.

- **Ancestors, Descendants:** The children segments and their children are *descendants* of the parent segment. For the descendants, their parent and the parent of their parent up to ς*-root* are called the *ancestors*.

- **Declaring Segment:** For the variables that are declared in one segment, this segment is called their *declaring segment*.

- **Default Segment:** There is a *default segment* that is implicitly inhabited at the top-level and is denoted as ς*-root*.

- **Path:** The *path* of a segment is the list of names that relate ς*-root* to this segment under the relation $\rightarrow$. For example, if a segment is declared with name "a" in the default segment, its path is $[a]$; if another segment is declared with name "b" in segment a, its path is $[a, b]$. The path of ς*-root* is the empty list.

- **Namespace:** The *namespace* of a variable or segment is the string formed by joining the names in the path of its declaring/parent segment by the full stop character. For example, for a variable declared in a segment whose path is $[a, b, c]$, its namespace is "a.b.c". The namespace of the variables or segments in ς*-root* is the empty string.

- **Qualified Name, Short Name:** The *qualified name* of a variable is the string formed by joining its namespace and name by a full stop character. For example, the qualified name of a variable x in the default segment is ".x"; the qualified name of a variable x with namespace "a.b.c" is "a.b.c.x". We also call the usual, non-qualified name the *short name*. In the discussion of chapter 3, whenever we use the word "name" we mean the *short name* unless otherwise specified. We also use the notation with 'q' in the subscript of a lower case letter to denote a variable in its qualified name, e.g., x_q, y_q.

- **Relative Path:** The *relative path* of a segment ς to an ancestor a is the list of names that relate a to ς under the relation $\rightarrow$. For example, if b is a child of a and c is child of b, the relative path of c to a is $[b, c]$.

- **Relative Namespace:** The *relative namespace* of a variable or segment to an ancestor a is the string formed by joining the names in the relative path of its declaring/parent segment to a by the full stop character. For example, if a

is a segment where x is declared as a variable and b is declared as a segment. In b, y is declared as a variable. Then the relative namespace of x to a is the empty string and the relative namespace of y to a is "b".

- **Parameter:** A *parameter* of a segment is a declaration of the form $(x : A)$ in this segment.

- **Instantiation:** A segment can be *instantiated* by giving a list of expressions. If the segment has no parameter, the list must be empty; otherwise the expressions in the list must have the same type as the parameters of the segment correspondingly. The result is a new segment with the variables of the parameters in the old segment bound to the expressions provided as their definitions. For example, for a segment ς with parameters $[x : A, y : B, z : C]$, it can be instantiated by a term of the form $s\,[M_1, M_2, M_3]$ where M_1, M_2, M_3 are expressions of types A, B, C respectively.

- **Direct access:** Items, variables and segments, in a segment ς or its descendants can be accessed by the dot operation (.): on the left of the operator is the relative namespace of the object to the segment ς whereas on the right is the name of the object. If the relative namespace is the empty string, which means the item is declared in ς, then this item is referred directly by its name. Both of the relative namespace and the name are used without quotes, i.e., if the relative namespace is "a.b.c" and the name is "x", variable x in segment c could be accessed from the parent of segment a by term $a.b.c.x$. This form of access is called the *direct access*.

- **Access By Instantiation:** The other form of access is *access by instantiation* where the segment referred by the name at the end of a relative path is instantiated before the object is accessed. It has the form $\varsigma_1. \ldots .\varsigma_n\,[M_1, \ldots, M_n]\,.\,x$, where $[\varsigma_1, \ldots, \varsigma_n]$ is the relative path of the segment ς_n to the parent of segment ς_1, and $[M_1, \ldots, M_n]$ are the expressions used to instantiate ς_n.

- **Reference Confinement:** Expressions in a segment ς can only refer to the items from ς or its descendants that have already been declared. This means that terms of the form $(\varsigma_1\,[M_1, \ldots, M_i]\,. \ldots .\varsigma_n\,[N_1, \ldots, N_j]\,.\,x)$ is not needed because instantiation on the ancestors has no effect on the descendants. We take a step further and consider terms of this form illegal in our language. We call this *The Rule of Reference Confinement.*

3.1 Syntax of the Extended Language

We introduce below the abstract syntax of the extended language, for the concrete syntax see appendix A.4. Expressions in the extended language are defined as follows:

Definition 3.1.1 (Expression)

(i) Terms of the form U, $[x : A]M$, $[D]\,M$ as defined in 2.2.1 are expressions in the extended language with the same meaning.

(ii) Given a non-empty list of names $[\varsigma_1, \ldots, \varsigma_n]$, a name x and a possibly empty list of tuples $[(M_1, x_1), \ldots, (M_i, x_i)]$ where M_j represents an expression and x_j a name, a new form of term

$$\varsigma_1. \ \ldots \ .\varsigma_n\,[(M_1, x_1), \ldots, (M_i, x_i)]\,.\,x$$

is an expression which belongs to the subset $\mathcal{K}$. When the list of tuples is not empty, it represents an access by instantiation to the variable x in the segment ς_n, whose relative path to the current segment is $[\varsigma_1, \ldots, \varsigma_n]$. In this case, x_1 to x_i represent the names of the parameters of ς_n that should be bound to expressions M_1 to M_i correspondingly; otherwise it represents a direct access to the variable x in the segment ς_n. Pairing each expression with the name of its corresponding parameter is not mandatory but helps to facilitate the evaluation and type checking procedure.

Declarations in the extended language are defined as follows.

Definition 3.1.2 (Declaration)

(i) Terms of the form $x : A$, $x : A = B$ as defined in 2.2.2 are still declarations in the extended language and have the same meaning.

(ii) Given a name ς and a possibly empty list of declarations ds, a term of the form

$$\varsigma = \texttt{Seg}\ ds$$

is a declaration of a segment ς consisting of the list of declarations ds.

(iii) Given a name ς, a non-empty list of names $[\varsigma_1, \ldots, \varsigma_n]$ and a possibly empty list of tuples $[(M_1, x_1), \ldots, (M_i, x_i)]$, a term of the form

$$\varsigma = \varsigma_1. \ \ldots \ .\varsigma_n\,[(M_1, x_1), \ldots, (M_i, x_i)]$$

is a declaration of a segment ς by the instantiation of another segment ς_n. The relative path of ς_n to the current segment is $[\varsigma_1, \ldots, \varsigma_n]$.

A program in the extended language consists of a list of declarations which belong to the default segment ς-*root*. Each segment is uniquely identified by its path and each variable is uniquely identified by its qualified name. A summary of the syntax of the extended language can be found in table 3.1.

3.2 Operational Semantics

In the evaluation operation, each segment has a representation of an environment. The fact that segments can be nested suggests a tree-like structure for the environment.

$$
\begin{array}{lll}
A, M & ::= & U \mid K \mid [x : A]M \mid [D]\, M \\
K & ::= & x \mid \varsigma_1. \ldots .\varsigma_n\, [(M_1, x_1), \ldots, (M_i, x_i)]\, .\, x \mid K\, M \\
D & ::= & x : A = M \\
Seg & ::= & \varsigma = \texttt{Seg}\, [Decl] \mid \varsigma = \varsigma_1. \ldots .\varsigma_n\, [(M_1, x_1), \ldots, (M_i, x_i)] \\
Decl & ::= & x : A \mid D \mid Seg \\
P & ::= & [Decl]
\end{array}
$$

Table 3.1: Syntax of the Extended Language

Definition 3.2.1 (Environment)
An environment ρ is a stack structure with an attribute p which represents the path of its corresponding segment and can be expressed in one of the following forms.

- $()$ represents an empty environment.
- $(\rho_1, x = q)$ extends a smaller environment ρ_1 by binding a variable x to a q-expression q; ρ shares the same path with ρ_1.
- (ρ_1, D) extends a smaller environment ρ_1 by a definition; ρ shares the same path with ρ_1.
- $(\rho_1, \varsigma = \rho')$ extends a smaller environment ρ_1 by a sub-environment ρ' which is a child segment with name ς; ρ shares the same path with ρ_1. If we denote the path of ρ as ρ_p, then the path of ρ' is $\rho'_p = \rho_p + [\varsigma]$.

The definition of q-expression in the extended language is the same as 2.3.2 and we still use the notation $M_\rho = q$ to express that the expression M is evaluated to q in the environment ρ. Semantics of the extended language is given in table 3.2.

$$
\begin{array}{lll}
U_\rho & = & U \\
(K\, N)_\rho & = & app(K_\rho, N_\rho) \\
([x : A]B)_\rho & = & \langle [x : A]B, \rho \rangle \\
([D]M)_\rho & = & M_{(\rho, D)} \\
x_\rho & = & \rho(x) \\
(\varsigma_1. \ldots .\varsigma_n\, [(M_1, x_1), \ldots, (M_i, x_i)]\, .\, x)_\rho & = & \text{let } \rho_1 = \iota_\rho([\varsigma_1, \ldots, \varsigma_n], [(M_1, x_1) \ldots, (M_i, x_i)]) \\
 & & \text{in } \rho_1(x)
\end{array}
$$

Table 3.2: Semantics of the Extended Language

The evaluation rules for expressions in forms of $U, (K\, N), [x : A]M, [D]M$ remain the same as that in table 2.3. To evaluate variables from the current segment and variables accessed by instantiation, two auxiliary functions are needed.

- $\rho(x)$: evaluates the variable x in environment ρ.[1]
- $\iota_\rho(rp, ens)$: gets the environment corresponding to the segment which is the

[1] We overload this function with a new definition.

result of instantiation on another segment by a list of tuples ens. The relative path of the segment being instantiated to ρ[2] is rp.

Function $\rho(x)$ relies on function $\mathcal{Q}(\rho_p, x)$: given the path of ρ, it returns the qualified name of variable x in ρ. The definition of $\rho(x)$ is given in table 3.3.

$$\begin{array}{rcl} ()(x_q) & = & x_q \\ ()(x) & = & \mathcal{Q}(()_p, x) \\ (\rho, x' = q)(x) & = & \text{if } x == x' \text{ then } q \text{ else } \rho(x) \\ (\rho, x' : A = B)(x) & = & \text{if } x == x' \text{ then } B_\rho \text{ else } \rho(x) \\ (\rho, \varsigma = \rho')(x) & = & \rho(x) \end{array}$$

Table 3.3: Function $\rho(x)$

Function ι relies further on function *findSegEnv* and two operations *mfst, msnd*.

- *findSegEnv*(rp, ρ): finds the environment ρ_1 whose relative path to ρ is rp. We use the notation $\rho_1 =\leadsto_{rp} \rho$ to express this function for brevity.
- *mfst*: extracts the first element from each tuple in a list, so for a list of tuples of the form $[(a_1, b_1), \ldots, (a_n, b_n)]$, the result is $[a_1, \ldots, a_n]$.
- *msnd*: extracts the second element from each tuple in a list, so for a list of tuples of the form $[(a_1, b_1), \ldots, (a_n, b_n)]$, the result is $[b_1, \ldots, b_n]$.

The definition of ι is given in table 3.4, where es_ρ represents the evaluation on a list of expressions es in the environment ρ; $(\rho_1, \sum_i(x_i = q_i))$ represents the environment which extends ρ_1 by binding the variables x_i from the list ns to the q-expressions q_i from qs.

$$\begin{array}{rcl} \iota_\rho(rp, ens) & = & \text{let } \rho_1 =\leadsto_{rp} \rho,\ es = mfst(ens), \\ & & ns = msnd(ens),\ qs = es_\rho \\ & & \text{in } (\rho_1, \sum_i(x_i = q_i)), x_i \in ns, q_i \in qs \end{array}$$

Table 3.4: Function ι

3.3 Type Checking Algorithm

During the type checking process, each segment has a representation of a type checking context which is constructed in a tree-like structure.

[2]More precisely, it should be the segment represented by ρ. To avoid verbosity, we adopt the practice to use the word "environment (ρ)" instead of the phrase "the segment represented by the environment (ρ)" whenever there is no ambiguity. We use the same practice when we talk about type checking context in the following sections.

Definition 3.3.1 (Type Checking Context)
A type checking context Γ is a stack structure with an attribute p which represents the path of its corresponding segment and can be expressed in one of the following forms.

- $()$ represents an empty context.
- $(\Gamma_1, x : A)$ extends a smaller context Γ_1 by a declaration; Γ shares the same path with Γ_1.
- (Γ_1, D) extends a smaller context Γ_1 by a definition; Γ shares the same path with Γ_1.
- $(\Gamma_1, \varsigma = \Gamma')$ extends a smaller context Γ_1 by a sub-context Γ' which represents the child segment with name ς; Γ shares the same path with Γ_1. If we denote the path of Γ as Γ_p, the path of Γ' is $\Gamma'_p = \Gamma_p + [\varsigma]$.

Given a type checking context Γ and a lock strategy s, we can get the evaluated form of the type of a variable x by the function *getType* which is denoted as $\Gamma(s, x)$. The function $\Gamma(s, x)$ will always succeed because only variables from Γ or its descendants are queried for types. This is guaranteed by (1) the rule of *Reference Confinement* which regulates that variables outside a segment cannot be referred in the segment and (2) a translation procedure which converts a concrete syntax tree to an abstract syntax tree where proper declaration and usage of variables are checked. If x appears in the form of a short name, it is declared in Γ; otherwise x is declared in a descendant of Γ. To find the type of a variable in a descendant segment, we introduce a function *locateSeg* which given a context Γ and a variable x in its qualified name, finds the relative path of the declaring segment of x to Γ. The relative path *rp* returned from this function can be used to get the context of the descendant with the function $findSegCtx$. We use the notation $\Gamma_1 =\!\rightsquigarrow_{rp} \Gamma$ to express that Γ_1 is the descendant of Γ whose relative path is *rp*. For a qualified name x_q, the function $sname(x_q)$ returns the short name of the variable x. The definition of $\Gamma(s, x)$ is given in table 3.5.

$$
\begin{array}{rcl}
\Gamma(s, x_q) & = & \text{let } rp = \mathit{locateSeg}(\Gamma, x_q),\ \Gamma_1 =\!\rightsquigarrow_{rp} \Gamma \\
 & & x = \mathit{sname}(x_q) \text{ in } \Gamma_1(s, x) \\
(\Gamma', x' : A)(s, x) & = & \text{if } x' == x \text{ then } A_{\varrho(s,\Gamma')} \text{ else } \Gamma'(s, x) \\
(\Gamma', x' : A = B)(s, x) & = & \text{if } x' == x \text{ then } A_{\varrho(s,\Gamma')} \text{ else } \Gamma'(s, x) \\
(\Gamma', \varsigma = \Gamma_1)(s, x) & = & \Gamma'(s, x)
\end{array}
$$

Table 3.5: Function getType

For the type checking algorithm, the lock strategy in the extended language has the same meaning as that of the basic language except that variables to be locked are now specified by their qualified names. There are four forms of judgments:

checkD	$\Gamma \vdash_s d \Rightarrow \Gamma'$	d is a valid declaration and extends Γ to Γ'.
checkT	$\Gamma \vdash_s M \Leftarrow t$	M is a valid expression given type t.
checkI	$\Gamma \vdash_s K \Rightarrow t$	K is a valid expression and its type is inferred to be t.
checkInst	$\Gamma, \Gamma' \vdash_s (M, x) \Rightarrow \Gamma_1$	M has the same type as the variable x in Γ'. Γ_1 is the segment resulting from the instantiation on the parameter x of segment Γ' by M.

Table 3.6: Forms of Judgment

3.3.1 checkD

$$\frac{\Gamma \vdash_s A \Leftarrow U}{\Gamma \vdash_s x : A \Rightarrow (\Gamma, x : A)} \tag{3.1}$$

$$\frac{\Gamma \vdash_s A \Leftarrow U \qquad \Gamma \vdash_s B \Leftarrow A_{\varrho(s,\Gamma)}}{\Gamma \vdash_s x : A = B \Rightarrow (\Gamma, x : A = B)} \tag{3.2}$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash_s d_1 \Rightarrow \Gamma_1 \\ \vdots \\ \Gamma_{n-1}, s \vdash_s d_n \Rightarrow \Gamma_n\end{array}}{\Gamma \vdash_s \varsigma = \mathtt{Seg}\,[d_1, \ldots, d_n] \Rightarrow (\Gamma, \varsigma = \Gamma_n)} \left(\Gamma_0 = \epsilon(\Gamma_p + [\varsigma]) \right) \tag{3.3}$$

$$\frac{\begin{array}{c}\Gamma, \Gamma_0 \vdash_s (M_1, x_1) \Rightarrow \Gamma_1 \\ \vdots \\ \Gamma, \Gamma_{i-1} \vdash_s (M_i, x_i) \Rightarrow \Gamma_i\end{array}}{\Gamma \vdash_s \varsigma = \varsigma_1. \ldots .\varsigma_n\,[(M_1, x_1), \ldots, (M_i, x_i)] \Rightarrow (\Gamma, \varsigma = \Gamma_i)} \left(\begin{array}{l} rp = [\varsigma_1, \ldots \varsigma_n] \\ \Gamma_0 = \rightsquigarrow_{rp} \Gamma \end{array} \right) \tag{3.4}$$

$\epsilon(\Gamma_p + [\varsigma])$ in rule 3.3 is a function that given a path $\Gamma_p + [s]$ returns an empty context with that path.

3.3.2 checkInst

$$\frac{\Gamma \vdash_s M \Leftarrow \Gamma'(s, x)}{\Gamma, \Gamma' \vdash_s (M, x) \Rightarrow \mathcal{U}(\Gamma', x, M_{\varrho(s,\Gamma)})} \tag{3.5}$$

$\mathcal{U}(\Gamma', x, q)$ is a function that turns the parameter x of the segment Γ' to a definition, i.e., suppose x is declared as $x : A$ in Γ', this function returns a new context with the same content as Γ' except that x has a definition $x : A = q$.

3.3.3 checkT

$$\frac{}{\Gamma \vdash_s U \Leftarrow U} \tag{3.6}$$

$$\frac{\Gamma(s,x) \sim_{\tau(\Gamma)} t}{\Gamma \vdash_s x \Leftarrow t} \tag{3.7}$$

$$\frac{\Gamma \vdash_s K\, N \Rightarrow t' \qquad t' \sim_{\tau(\Gamma)} t}{\Gamma \vdash_s K\, N \Leftarrow t} \tag{3.8}$$

$$\frac{\Gamma \vdash_s A \Leftarrow U \qquad (\Gamma, x : A) \vdash_s B \Leftarrow U}{\Gamma \vdash_s [x : A]B \Leftarrow U} \tag{3.9}$$

$$\frac{\Gamma \vdash_s A \Leftarrow U \qquad A_{\varrho(s,\Gamma)} \sim_{\tau(\Gamma)} A'_{\rho'} \qquad (\Gamma, x : A) \vdash_s B \Leftarrow B'_{(\rho', x'=x_q)}}{\Gamma \vdash_s [x : A]\, B \Leftarrow \langle [x' : A']B', \rho' \rangle} \left(\; x_q = \mathcal{Q}(\Gamma_p, x) \;\right) \tag{3.10}$$

$$\frac{\Gamma \vdash_s A \Leftarrow U \qquad \Gamma \vdash_s B \Leftarrow A_{\varrho(s,\Gamma)} \qquad (\Gamma, x : A = B) \vdash_s M \Leftarrow t}{\Gamma \vdash_s [x : A = B]\, M \Leftarrow t} \tag{3.11}$$

$$\frac{\begin{array}{c} \Gamma, \Gamma_0 \vdash_s (M_1, x_1) \Rightarrow \Gamma_1 \\ \vdots \\ \Gamma, \Gamma_{i-1} \vdash_s (M_i, x_i) \Rightarrow \Gamma_i \qquad \Gamma_i(s,x) \sim_{\tau(\Gamma)} t \end{array}}{\Gamma \vdash_s \varsigma_1. \ldots .\varsigma_n\, [(M_1, x_1), \ldots, (M_i, x_i)].x \Leftarrow t} \left(\begin{array}{l} rp = [\varsigma_1, \ldots \varsigma_n] \\ \Gamma_0 = \rightsquigarrow_{rp} \Gamma \end{array} \right) \tag{3.12}$$

$\mathcal{Q}(\Gamma_p, x)$ in rule 3.10 is a function that given the path of Γ returns the qualified name of variable x from Γ.

3.3.4 checkI

$$\frac{}{\Gamma \vdash_s x \Rightarrow \Gamma(s,x)} \tag{3.13}$$

$$\frac{\Gamma \vdash_s K \Rightarrow \langle [x : A]B, \rho \rangle \qquad \Gamma \vdash_s N \Leftarrow A_\rho}{\Gamma \vdash_s K\, N \Rightarrow B_{(\rho, x=n)}} \left(\; n = N_{\varrho(s,\Gamma)} \;\right) \tag{3.14}$$

$$\frac{\begin{array}{c} \Gamma, \Gamma_0 \vdash_s (M_1, x_1) \Rightarrow \Gamma_1 \\ \vdots \\ \Gamma, \Gamma_{i-1} \vdash_s (M_i, x_i) \Rightarrow \Gamma_i \end{array}}{\Gamma \vdash_s \varsigma_1. \ldots .\varsigma_n\, [(M_1, x_1), \ldots, (M_i, x_i)].x \Rightarrow \Gamma_i(s,x)} \left(\begin{array}{l} rp = [\varsigma_1, \ldots, \varsigma_n] \\ \Gamma_0 = \rightsquigarrow_{rp} \Gamma \end{array} \right) \tag{3.15}$$

3.4 Linear Head Reduction

The function *linear head reduction* in the extended language has the same definition as that in table 2.13, so does the function *readBack*. The definition of *headRedV*, however, is different because of the introduction of segments.

$$\begin{array}{lcl}
\delta^*(\Gamma, x) & = & \mathcal{V}(\Gamma, x) \\
\delta^*(\Gamma, \varsigma_1. \ldots .\varsigma_n & = & \text{let } rp = [\varsigma_1, \ldots, \varsigma_n],\ \rho = \varrho([], \Gamma), \\
[(M_1, x_1), \ldots, (M_i, x_i)] \,.\, x) & & \rho_1 = \iota_\rho(rp, [(M_1, x_1), \ldots, (M_i, x_i)]), \\
 & & \text{in } \mathcal{V}(\rho_1, x) \\
\delta^*(\Gamma, K\, N) & = & \text{let } k = \delta^*(\Gamma, K),\ n = N_{()} \text{ in } app(k, n)
\end{array}$$

Table 3.7: Function headRedV in the Extended Language

The function *getVal* ($\mathcal{V}$) in table 3.7 is overloaded to express: (1) $\mathcal{V}(\Gamma, x)$, the function that gets the least evaluated form of variable x in the context Γ; and (2) $\mathcal{V}(\rho, x)$, the function that gets the least evaluated form of variable x in the environment ρ. A difference with table 2.11 is that the variable x here could be in the form of its qualified name x_q. In this case, the function needs to locate the sub-context where x is declared in a similar way as that of table 3.5. We give the definition of *getVal* for both cases in table 3.8.

$$\begin{array}{lcl}
\mathcal{V}(\Gamma, x_q) & = & \text{let } rp = locateSeg(\Gamma, x_q),\ \Gamma' = \leadsto_{rp} \Gamma,\ x = sname(x_q) \text{ in } \mathcal{V}(\Gamma', x) \\
\mathcal{V}((), x) & = & \mathcal{Q}(()_p, x) \\
\mathcal{V}(\Gamma', \varsigma = \Gamma_1) & = & \mathcal{V}(\Gamma', x) \\
\mathcal{V}((\Gamma', x' : A), x) & = & \text{if } x' == x \text{ then } \mathcal{Q}(\Gamma'_p, x) \text{ else } \mathcal{V}(\Gamma', x) \\
\mathcal{V}((\Gamma', x' : A = B), x) & = & \text{if } x' == x \text{ then } B_{()} \text{ else } \mathcal{V}(\Gamma', x)
\end{array}$$

$$\begin{array}{lcl}
\mathcal{V}((), x) & = & \mathcal{Q}(()_p, x) \\
\mathcal{V}(\rho', x' = \rho_1) & = & \mathcal{V}(\rho', x) \\
\mathcal{V}((\rho', x' : A = B), x) & = & \text{if } x' == x \text{ then } B_{()} \text{ else } \mathcal{V}(\rho', x)
\end{array}$$

Table 3.8: Function getVal

An example of the function head reduction in the extended language is given in appendix A.8, which is the result of applying head reduction repeatedly to the constant "loop" and the results are given in appendix A.7. It shows the same result as in appendix A.6 but is performed with the involvement of segment.

4

Results

The theoretical result of this project is that we designed and implemented a dependently typed language. Particularly, we studied and implemented a definition mechanism where the constants of a program can be locked/unlocking during the type checking process. This definition mechanism with its locking/unlocking functionality prove to be an effective way to improve the efficiency in the type checking algorithm in the dependent type theory. By extending the language with a module system, we showed that the definition mechanism is flexible and scalable enough to incorporate more language features.

The practical result of the project is a REPL (read-evaluate-print-loop) program developed in Haskell where a source program of our language could be loaded and type checked. The program features a static context and a dynamic context. The former is the context formed by loading a source program and can be extended by declarations from the user input. The latter serves as a buffer where the user can give names to expressions. Values of the constants from the static context cannot be changed, while variables from the dynamic context can be bound to new expressions without restriction. The feature of the dynamic context, together with other commands such as *hRed* (head reduction) are provided for ease of use to experiment with the definition mechanism built into the language. The source code of the program could be found at the Github repository: `https://github.com/WANG-QUFEI/Master-Thesis`. A summary of the commands provided by the program is listed in appendix A.9.

5

Conclusion

In this paper we presented a language of dependent type theory as an extension to the pure lambda calculus with dependent types and definitions. We studied and implemented a definition mechanism in the language where convertibility checking with the presence of definitions during the type checking process could be handled more efficiently. As an application of the definition mechanism, we extended the language with a module system to show that the core concepts used in this mechanism, such as using closures to defer computation, transforming constants into primitives to avoid definition expansion, checking the convertibility of terms on the level of their intermediate form of evaluation by their syntactic identity, etc., could be adapted to support new language features. The experience we got in the design and implementation of the language helps us understand better the dependent type theory and the inherent difficulties in its type checking algorithm.

Our work could be seen as a study into the basic problem of how definitions in the dependent type theory should be presented in an efficient way. As larger programs and more sophisticated problems put more demand on the performance of the proof assistant systems, a practical and efficient definition mechanism is crucial to tackle these challenges for the further development of the dependent type theory.

Future work based on this project could be conducted in three directions:

1. More language facilities towards a well defined core language for functional programming: such as language support for basic data types, functions with the ability to pattern match on expressions and user defined (inductive) data types.

2. Metatheory study on the properties of this language as a logic system, such as the decidability of the type checking algorithm.

3. Incorporation of the languages formulated in the work of AUTOMATH. As one of the pioneering work in the field of dependent type theory, AUTOMATH provides ideas that are borrowed by this work and more left to be studied for a better understanding of the dependent type theory and the foundation of mathematical logic.

Bibliography

[1] 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.

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

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

[4] B. Barras, “A module system based on opacity,” 2016.

[5] H. Balsters, “Lambda calculus extended with segments: Chapter 1, sections 1.1 and 1.2 (introduction),” in *Studies in Logic and the Foundations of Mathematics*, vol. 133, pp. 339–367, Elsevier, 1994.

[6] L. van Benthem Jutting, *Checking Landau's "Grundlagen" in the Automath system.* StichtingMathematisch Centrum, 1977.

[7] T. Coquand, Y. Kinoshita, B. Nordström, and M. Takeyama, “A simple type-theoretic language: Mini-tt,” *From Semantics to Computer Science; Essays in Honour of Gilles Kahn*, pp. 139–164, 2009.

[8] P. J. Landin, “The mechanical evaluation of expressions,” *The computer journal*, vol. 6, no. 4, pp. 308–320, 1964.

[9] N. G. de Bruijn, “Generalizing automath by means of a lambda-typed lambda calculus,” in *Studies in Logic and the Foundations of Mathematics*, vol. 133, pp. 313–337, Elsevier, 1994.

[10] 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 Evaluation Using Closure

In the following demonstration, we use $\rightarrow_\lambda$ to denote the erase of λs in β-reduction, $\rightarrow_s$ to denote the substitution and $()_e$ to denote the environment.

$$\begin{aligned}
(\lambda u\ .\ u\ (u\ b))(\lambda z\ y\ x\ .\ a\ (z\ x)\ y) &\rightarrow_\lambda \\
(u\ (u\ b))(u = \lambda z\ y\ x\ .\ a\ (z\ x)\ y)_e &\rightarrow_s \\
(\lambda z\ y\ x\ .\ a\ (z\ x)\ y)((\lambda z\ y\ x\ .\ a\ (z\ x)\ y)\ b) &\rightarrow_\lambda \\
(\lambda y\ x\ .\ a\ (z\ x)\ y)(z = (\lambda z\ y\ x\ .\ a\ (z\ x)\ y)\ b)_e &
\end{aligned}$$

To show that the problem of capture of names could be avoided, we apply the result to arguments y_0, x_0.

$$\begin{aligned}
(\lambda y\ x\ .\ a\ (z\ x)\ y)(z = (\lambda z\ y\ x\ .\ a\ (z\ x)\ y)\ b)_e\ y_0\ x_0 &\rightarrow_\lambda \\
(a\ (z\ x)\ y)(z = (\lambda z\ y\ x\ .\ a\ (z\ x)\ y)\ b, y = y_0, x = x_0)_e &\rightarrow_s \\
a\ (((\lambda z\ y\ x\ .\ a\ (z\ x)\ y)\ b)\ x_0)\ y_0 &\rightarrow_\lambda \\
a\ ((\lambda y\ x\ .\ a\ (z\ x)\ y)(z = b)_e\ x_0)\ y_0 &\rightarrow_\lambda \\
a\ (\lambda x\ .\ a\ (z\ x)\ y)(z = b, y = x_0)_e\ y_0 &
\end{aligned}$$

Suppose we apply the closure in the middle to another argument x_1, we get

$$\begin{aligned}
(\lambda x\ .\ a\ (z\ x)\ y)(z = b, y = x_0)_e\ x_1 &\rightarrow_\lambda \\
(a\ (z\ x)\ y)(z = b, y = x_0, x = x_1)_e &\rightarrow_s \\
a\ (b\ x_1)\ x_0 &
\end{aligned}$$

which is correct.

A.2 η-Conversion

To check η-convertibility, instead of using a predicate as that in table 2.9, two new forms of judgments are needed.

checkCI	$\Gamma \vdash_s q_1\, q_2 \Rightarrow t$	q_1 and q_2 are convertible and their type can be inferred as t
checkCT	$\Gamma \vdash_s q_1\, q_2 \Leftarrow t$	q_1 and q_2 are convertible given t as their type

Table A.1: New Judgments for Checking η-Convertibility

A.2.1 CheckCI

$$\frac{}{\Gamma \vdash_s U \sim U \Rightarrow U} \tag{A.1}$$

$$\frac{x == y}{\Gamma \vdash_s x \sim y \Rightarrow \Gamma(s, x)} \tag{A.2}$$

$$\frac{\Gamma \vdash_s k_1 \sim k_2 \Rightarrow \langle [x : A]B, \rho\rangle \qquad \Gamma \vdash_s v_1 \sim v_2 \Leftarrow A_\rho}{\Gamma \vdash_s (k_1\ v_1) \sim (k_2\ v_2) \Rightarrow B_{(\rho, x=v_1)}} \tag{A.3}$$

$$\frac{\Gamma \vdash_s \langle [x : A]B, \rho\rangle \sim \langle [x' : A']B', \rho'\rangle \Leftarrow U}{\Gamma \vdash_s \langle [x : A]B, \rho\rangle \sim \langle [x' : A']B', \rho'\rangle \Rightarrow U} \tag{A.4}$$

A.2.2 CheckCT

$$\frac{(\Gamma, y : A_\rho) \vdash_s (k_1\ y)_{\varrho(s,\Gamma)} \sim (k_2\ y)_{\varrho(s,\Gamma)} \Leftarrow B_{(\rho, x=y)}}{\Gamma \vdash_s k_1 \sim k_2 \Leftarrow \langle [x : A]B, \rho\rangle} \Big(\ y \ = \ \nu(\tau(\Gamma), x)\ \Big) \tag{A.5}$$

$$\frac{\Gamma \vdash_s A_\rho \sim A'_{\rho'} \Leftarrow U \qquad (\Gamma, y : A_\rho), \vdash_s B_{(\rho, x=y)} \sim B'_{(\rho', x'=y)} \Leftarrow U}{\Gamma \vdash_s \langle [x : A]B, \rho\rangle \sim \langle [x' : A']B', \rho'\rangle \Leftarrow U} \Big(\ y \ = \ \nu(\tau(\Gamma), x_1)\ \Big) \tag{A.6}$$

$$\frac{\Gamma \vdash_s v1 \sim v2 \Rightarrow t' \qquad \Gamma \vdash_s t \sim t' \Rightarrow \_}{\Gamma \vdash_s v1 \sim v2 \Leftarrow t} \tag{A.7}$$

A.3 Concrete Syntax for the Basic Language

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

entrypoints Context, CExp, CDecl;

Ctx. Context ::= [CDecl];
```

```
CU.       CExp2 ::= "*";
CVar.     CExp2 ::= Id;
CApp.     CExp1 ::= CExp1 CExp2;
CArr.     CExp  ::= CExp1 "->" CExp;
CPi.      CExp  ::= "[" Id ":" CExp "]" CExp;
CWhere.   CExp  ::= "[" Id ":" CExp "=" CExp "]" CExp ;

CDec.     CDecl ::= Id ":" CExp;
CDef.     CDecl ::= Id ":" CExp "=" CExp;

terminator CDecl ";";

coercions CExp 3;

layout toplevel;

comment "--";

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

A.4 Concrete Syntax for the Extended Language

```
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.5 Variation of Hurkens Paradox

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

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

abs : * = [X : *] X

not : * -> * = [X : *] X -> abs

A : * = [X : *] (T X -> X) -> X

intro : T A -> A =
  [t : T A][X : *][f : T X -> X] f ([g : Pow X] t ([z : A] g (z X f)))

match : A -> T A =
  [z : A]z (T A) ([t : T (T A)][g : Pow A] t ([x : T A] g (intro x)))

delta : A -> A = [z : A] intro (match z)

Q : T A = [p : Pow A][z : A]match z p -> p z

cDelta : Pow A -> Pow A =  [p : Pow A][z:A]p (delta z)

a0 : A = intro Q

lem1 : [p : Pow A]Q p -> p a0 = [p : Pow A][h : Q p]h a0 ([x : A]h (delta x))

Ed : Pow A = [z:A][p:Pow A]match z p -> p (delta z)

lem2 :  Ed a0  = [p:Pow A]lem1 (cDelta p)

B : Pow A =  [z : A] not (Ed z)

lem3 : Q B = [z : A] [k : match z B] [l : Ed z] l B k ([p:Pow A]l (cDelta p))
```

```
lem4 : not (Ed a0) = lem1 B lem3

loop : abs = lem4 lem2
```

A.6 Example of Head Reduction

```
1:  lem4 lem2
2:  lem1 B lem3 lem2
3:  lem3 a0 ([ x : A ] lem3 (delta x)) lem2
4:  lem2 B ([ x : A ] lem3 (delta x)) ([ p : Pow A ] lem2 (cDelta p))
5:  lem1 (cDelta B) ([ x : A ] lem3 (delta x))
      ([ p : Pow A ] lem2 (cDelta p))
6:  lem3 (delta a0) ([ x : A ] lem3 (delta (delta x)))
      ([ p : Pow A ] lem2 (cDelta p))
7:  lem2 (cDelta B) ([ x : A ] lem3 (delta (delta x)))
      ([ p : Pow A ] lem2 (cDelta (cDelta p)))
8:  lem1 (cDelta (cDelta B)) ([ x : A ] lem3 (delta (delta x)))
      ([ p : Pow A ] lem2 (cDelta (cDelta p)))
9:  lem3 (delta (delta a0)) ([ x : A ] lem3 (delta (delta (delta x))))
      ([ p : Pow A ] lem2 (cDelta (cDelta p)))
10: lem2 (cDelta (cDelta B)) ([ x : A ] lem3 (delta (delta (delta x))))
      ([ p : Pow A ] lem2 (cDelta (cDelta (cDelta p))))
```

A.7 Variation of Hurkens Paradox with Segment

```
 lambek = seg

  T : * -> *

  mon : [X : *][Y : *] (X -> Y) -> (T X -> T Y)

  A : * = [X : *] (T X -> X) -> X

  intro : T A -> A =

    [z : T A][X : *][f : T X -> X]

    [u : A -> X = [a : A] a X f]

    [v : T A -> T X = mon A X u] f (v z)

  match : A -> T A = [a : A] a (T A) (mon (T A) A intro)

  mint : T A -> T A =

    [z : T A] match (intro z)
```

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

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

mon0 : [X:*][Y:*](X -> Y) -> (T X -> T Y) =

  [X:*][Y:*][f:X -> Y][u : T X][v: Pow Y] u ([x:X] v (f x))

s = lambek [T, mon0]

A : * = s.A

intro : T A -> A = s.intro

match : A -> T A = s.match

abs : * = [X : *] X

not : * -> * = [X : *] X -> abs

delta : A -> A = [z : A] intro (match z)

Q : T A = [p : Pow A][z : A] match z p -> p z

cDelta : Pow A -> Pow A =  [p : Pow A] [z:A] p (delta z)

a0 : A = intro Q

lem1 : [p : Pow A] Q p -> p a0 = [p : Pow A][h : Q p] h a0 ([x : A] h (delta x))

Ed : Pow A = [z:A][p:Pow A] match z p -> p (delta z)

lem2 :  Ed a0  = [p:Pow A] lem1 (cDelta p)

B : Pow A =  [z : A] not (Ed z)

lem3 : Q B = [z : A] [k : match z B] [l : Ed z] l B k ([p:Pow A] l (cDelta p))

lem4 : not (Ed a0) = lem1 B lem3

loop : abs = lem4 lem2
```

A.8 Example of Head Reduction With Segment

```
1:  lem4 lem2
2:  lem1 B lem3 lem2
3:  lem3 a0 ([ x : A ] lem3 (delta x)) lem2
4:  lem2 B ([ x : A ] lem3 (delta x)) ([ p : Pow A ] lem2 (cDelta p))
```

```
5:  lem1 (cDelta B) ([ x : A ] lem3 (delta x)) ([ p : Pow A ] lem2 (cDelta p))
6:  lem3 (delta a0) ([ x : A ] lem3 (delta (delta x)))
      ([ p : Pow A ] lem2 (cDelta p))
7:  lem2 (cDelta B) ([ x : A ] lem3 (delta (delta x)))
      ([ p : Pow A ] lem2 (cDelta (cDelta p)))
8:  lem1 (cDelta (cDelta B)) ([ x : A ] lem3 (delta (delta x)))
      ([ p : Pow A ] lem2 (cDelta (cDelta p)))
9:  lem3 (delta (delta a0)) ([ x : A ] lem3 (delta (delta (delta x))))
      ([ p : Pow A ] lem2 (cDelta (cDelta p)))
10: lem2 (cDelta (cDelta B)) ([ x : A ] lem3 (delta (delta (delta x))))
      ([ p : Pow A ] lem2 (cDelta (cDelta (cDelta p))))
```

A.9 REPL Command List

<statement>	A statement could be an expression or a declaration. For an expression, it will be type checked and evaluated and the result will be bound to the name "_it" in the dynamic context. For an declaration, it will be type checked and added to the static context.
:load <file_path>	Load the file of path <file_path> with the current locking strategy. Once successfully loaded, the context of the file will become the new static context and the dynamic context will be reset to its initial state.
:let <name> = <expression>	Bind an expression to a name. The expression will be type checked first and if it is valid, its type will be inferred and a definition consisting of the name, the type and the expression will be added to the dynamic context.
:type <expression>	Infer the type of an expression after it is type checked.
:hRed <expression>	Apply head reduction on an expression after it is type checked.
:show -lock | -context	Option "-lock": show the current lock strategy; Option "-context": show the current type checking context.
:lock -all | -none | -add | -remove	Change lock strategy. "-all": lock all constants; "-none": lock no constant; "-add [variables]": add a list of names to be locked; "-remove [variables]": remove a list of names to be locked. Default strategy is "-none".
:set -conversion <beta | eta>	Set the convertibility check support, β-conversion or η-conversion.
:check_convert <exp1> $\sim$ <exp2>	Check the convertibility of two expressions if they are both valid
:quit	Stop and quit.
:?, :help	Show this usage message.

Table A.2: REPL Command List