CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

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

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 Dependent Type Theory with Definitions
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 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 lambda 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 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, October 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 the proof-assistant systems like Coq[2], 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 ($\texttt{vec}\ A\ n$). 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 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.

The downside of dependent type systems, however, lies in the difficulties of 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 affects directly 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 [1]: 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 could contain any expression as its component and deciding

[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.

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.

The first part of this project consists of the specification of a dependently typed language which features a definition mechanism where constants could be locked/unlocked manually during the type checking process. The first part contains the main theoretical results of this project and a thorough exposition of the definition mechanism is given in section 2.6. As an application of the definition mechanism, we built in the second part a module system based on the concept "segment" borrowed from the work AUTOMATH[4]. The adaptation to the concept of namespace introduced by the module system could be seen as an evidence for the scalability of our definition mechanism.

1.3 Organization of This Paper

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 as the result a REPL (read-evaluate-print-loop) 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 of the Project

1. **Expressiveness:** The expressiveness of the language is intentionally restrained as an attempt to keep the language simple 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 the limit of time, 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 understand better 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

We present the subtleties by giving examples as the follows.

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) naively, we get

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

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 we do a naive reduction in (2.2) without renaming, we get

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

which is not correct. This example comes from the PhD thesis of L.S. van Benthem Jutting in 1977[5] when he was working on AUTOMATH. It was conjectured that if one starts with a term where each bound variable is only declared once, then 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 definition by function application. For example, one can interpret the definition of $i1$ in following piece of java code

```
1    int i1 = 0;
2    int i2 = i1 + 1;
3
```

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

```
1    let x = 0 in x + x
2
```

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 program

```
1    let x : A = a, y : P x in f y
2
```

cannot be rewritten to an 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.

2.2 Principles

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 consists of a function (λ-abstraction) and an *environment* which binds free variables of the function to terms that waits 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 a function $(\lambda y.x + y)$ and an 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, 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 avoid by using closure is that by deferring substitution, 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 2.4. An example of using closure to evaluate the expression 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 during the type checking process, each declaration, including declaration of bound variable in λ-abstraction, is checked with the top level context to ensure no name collision occurs. Another approach to the name shadowing problem is called *namespaced De Bruijn indices*[1], which is a technique adopted by Agda currently[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.

```
1  module test where
2
3  open import Data.Nat
4  open import Data.Bool
5
6  test : N → Bool → N
7  test = λ (x : N) →
8    let y : N
9        y = x
10   in λ (x : Bool) → {!!}
```

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

[1] for a detailed introduction, please visit this website.

[2] tested with version 2.6.2.

- Line 6-10 is a definition of constant *test* which is a function that given a natural number and a Boolean, returns a natural number.
- Two variables of the same name x are declared, one in the outer scope at line 7, another in the inner scope at line 10.
- 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 ask for the context information.

The context information inferred by Agda is:

```
Goal: N
————————————————————————————————————————————————————————————
y      : N
y      = x_1
x      : Bool
x = x_1 : 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, thus having the 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 defined to be 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 `y =` x_1, `x:Bool` and `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 name x_1. Such an attempt will be rejected by Agda with an error message meaning that "variable not in scope". In summary, we consider allowing the feature of name shadowing causes confusion in the context information and introduces ambiguity over the usage of names (e.g., for name x_1, should it be taken as an internal identifier or a common name?). For this reason, we simply forbid the shadowing of names in our language.

2.3 Syntax of the Language

There are two kinds of syntax regarding 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.

Expression in our language is defined as follows:

Definition 2.3.1 (Expression)

(i) U is an expression, which represents a 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 defined inductively by
   (a) variables, e.g., x, y, z;
   (b) terms of the form $K\,M$, 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 function $A \to M$.

(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.

Declaration in our language is defined as follows:

Definition 2.3.2 (Declaration)

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

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

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

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

Table 2.1: Syntax of the Language

The syntax of our language is a subset of Mini-TT[6]. 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 cause 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.

2.4 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*, a stack structure in one of the following forms:

Definition 2.4.1 (Environment)

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

(ii) $(\rho_1, x = v)$, an environment extended from a smaller one ρ_1 by binding a variable x to a q-expression v;

(iii) (ρ_1, D), an environment extended from a smaller one ρ_1 by 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 2.6.2. Sometimes we also call q-expressions as *values*.

Definition 2.4.2 (q-expression)

(i) U is a q-expression, it is the result of the expression U under evaluation.

(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 a function $[x : A]M$ under evaluation in the environment ρ.

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

The form of q-expression can be summarized in table 2.2.

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

Table 2.2: Form of Q-expression

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.

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

- $\rho(x)$: find the binding q-expression of the variable x 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

- $app(k, v)$: apply function k to v.

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

Table 2.4: Function - $\rho(x)$

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

Table 2.5: Function - $app(k, v)$

Some readers may have noticed that the real difference between expression and q-expression is closure. Closure is an important concept in functional programming and was first conceived by P. J. Landin in his paper *The Mechanical Evaluation of Expressions*[7]. There, the author described closure as "...comprising the λ-expression and the environment relative to which it was evaluated..." which specified the structure of closure we adhere to in our own implementation. Closure is introduced to meet the need of passing functions as values 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, evaluation of the reducible expressions in the function body is deferred, as signified by the rule about evaluation of function 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 function *app* in table 2.5. For an application of function $[x : A]M$ to an argument v, the substitution will not happen until M is clear of abstraction. The ability to defer computation is crucial for the definition mechanism as it makes possible for saving computations during the evaluation process.

2.5 Type Checking Algorithm

The aim of the type checking algorithm is to ensure a program of 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 any syntactic or semantic error related with names, such as duplicated declaration of names or use of undeclared names. Syntactic error is 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 the abstract syntax tree on which the type checking algorithm is applied.

There are three 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.

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. Γ represents the *type checking context* which is a stack structure keeping track of the types and definitions of the variables. s represents a *lock strategy*.

Definition 2.5.1 (Type Checking Context)
A type checking context Γ has one of the three forms.

(i) (): Γ is an empty context.

(ii) $(\Gamma_1, x : A)$: Γ is a context extended from Γ_1 by a declaration $x : A$.

(iii) $(\Gamma_1, x : A = B)$: Γ is a context extended from Γ_1 by a definition $x : A = B$.

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

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 environment is the place where variables are mapped to their definitions or values (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 of s in table 2.6 being a list of locked variables, the function *getEnv* could be defined as in table 2.7.

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 function *getType*. We denote this function as $\Gamma(s, x)$ and give its definition in table 2.8.

The first case of the function means that if we try to get the type of a variable that does not exist in the type checking context, an exception is raised. In our

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

Table 2.7: Function: getEnv

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

Table 2.8: Function - getType

implementation, during the *translation* process we mentioned earlier, 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 will never actually happen during the type checking process.

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. The list of names ns is used to ensure that names newly introduced in the convertibility checking process do not collide with the names already existed in the underlying type checking context. The definition of *checkConvert* is given in table 2.9. Note that the rules presented here only checks β-convertibility, for η-convertibility please refer to appendix A.2.

$U \sim_{ns} U$	-
$x \sim_{ns} x$	-
$k_1\, v_1 \sim_{ns} k_2\, v_2$	if $k_1 \sim_{ns} k_2$ and $v_1 \sim_{ns} v_2$
$\langle [x : A]M, \rho\rangle \sim_{ns} \langle [x' : A']M', \rho'\rangle$	check that $A_\rho \sim_{ns} A'_{\rho'}$, let $y = \nu(ns, x)$ check that $M_{(\rho, x=y)} \sim_{y:ns} M'_{(\rho', x'=y)}$

Table 2.9: Predicate - CheckConvert

A new function *freshVar*, denoted as ν, is used to generate a new variable that does not collide with the existing ones from ns. Another function *namesCtx*, denoted as $\tau(\Gamma)$, is used to get the names of declarations plus the names of definitions in the let clauses in a context Γ. We use this function to provide the list of names used by *checkConvert*.

2.5.1 checkD

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

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

2.5.2 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.

2.5.3 checkI

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

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

2.6 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 given an expression E and a type A, decide 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. Difficulty arises in dependent type theory because (1) a type may contain **any** expression which could entail large amount of computation, and (2) the use of definition 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 about the integer multiplication and exponentiation, and determine that any expression should be evaluated to the 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, then 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 limit on constants. That is, a constant acts as a unit on which computation could be locked or charged. More advanced computation control technique 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 computation, 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 an function called "linear head reduction" which could limit reductions performed on expressions each time.

2.6.1 Problem of Finding the Minimum Set of Constants

One application of our definition mechanism is that given a valid context, find the minimum set of constants unfolded such that a new constant could be type checked. More formally, the problem could be formulated as:

Problem. G is a valid context and the set of constants from G is C. x is a new constant with definition $x : A = B$. Find one minimum set $C_0 \subseteq C$ such that for an arbitrary set $C' \subseteq C$

$$G \vdash_{s(C')} x : A = B \Rightarrow (G, x : A = B) \quad \textit{iff} \quad C_0 \subseteq C'$$

where $s(C')$ is the list of constants to be locked during the type checking process from $C \setminus C'$.

When x is invalid, the problem is uninteresting because C_0 does not exist. When x is valid, we assume the existence and uniqueness of C_0 and present below an

approximation algorithm to find C_0. Notice that according to the rule 2.4, checking x valid entails checking $A \in U$ and $B \in A_{\varrho(s,G)}$ for some lock strategy s, both having the pattern $G \vdash_s M \Leftarrow t$. Based on this observation the problem could be reduced to given an expression M with type t, find the minimum set of constants $S_0 \subseteq C$ unfolded such that $G \vdash_{s(C')} M \Leftarrow t$ for an arbitrary $C' \subseteq C$ if and only if $C_0 \subseteq C'$. Also notice that with the assumption of the existence of C_0 there is always a simple but inefficient method to find it: we run the type checking algorithm on all the subsets of C incrementally by the order of the number of elements each subset contains, and we choose the one with the minimum number of elements where the type checking process succeeds. Since the worst-case time complexity is exponential to the number of constants from G, we aim to find a more efficient method to this problem.

The algorithm presented below consists of a collection of auxiliary functions that are similar with the ones used in the type checking process. The idea is to discover cases where a constant must be unfolded according to the inference rules of the type checking algorithm. It resembles the type checking algorithm in many aspects but with three major differences:

1. Instead using two stack structures, a type checking context and an environment, to keep types and values of expressions, this algorithm uses only one context called *constant searching context* that keeps both the types and values of expressions during the searching process.
2. Instead of halting with an exception when encounters a locked constant that needs to be unfolded, this algorithm queries the definition of the constant from the *constant searching context* and performs a pattern match on it.
3. Expressions are not evaluated in the context, they serve the only purpose for the case analysis operations on their patterns.

Functions used by the algorithm are summarized in table 2.10 and their definitions are given in a Haskell-style pseudo-code in listing 2.1. The word "context" used in the following discussion refers particularly to the *constant searching context*.

```
{-|
* @param 1: the constant searching context that
  relates variables to their type and values
* @param 2: an expression M
* @param 3: an expression N
* @param 4: a list of names which represents
*           the constants already discovered

* this function searches for the constants that must be unfolded
* for the possibility that M has type N by the rules
* described in section  2.5.2 .
-}
matchType :: Cont -> Exp -> Exp -> [String] -> [String]
matchType G (D M) T s = matchType (G,D) M T s
------------------------------------------------------------
matchType G U U s = s
matchType G x U s =
  let T = v_type G x
  in searchCon G U T s
```

matchType(G, M, T):	In context G, find the constants that need to be unlocked for the possibility that an expression M has another expression T as its type.
inferType(G, i, K, V):	In context G, infer the type of an application $K\,V$. Because K has the form either a variable k or an application $K'\,V'$, the second argument indicates the direction of the recursion: 0 means doing the recursion towards the variable k; 1 means exiting previous recursions after having found the variable k.
searchCon(G, M, N, s):	In context G, given two expressions M, N, find the constants to be unlocked for the possibility that M and N are convertible. s is the list of constants already found to be locked.
searchFun(G, K, V, s):	In context G, given two expressions K, V, find the constants to be unlocked for the possibility that K is a function. V is the argument to the function represented by K.
head_v(K):	Given an expression $K \in \mathcal{K}$, find the left most variable.
fresh_v(G, x):	Given the name of a variable x, generates a new variable that does not exist in G.
v_val(G, x):	Get the value of variable x in the context G.
v_type(G, x):	Get the type of variable x in the context G.
merge$(G1, G2, M)$:	Given two contexts $G1, G2$, merge the context $G2$ to $G1$ by renaming variables in $G2$ that cause name collision with that of $G1$ using the function *fresh_v*. Variables renamed in $G2$ are also renamed in the same way in M.

Table 2.10: Summary of Functions Used to Find The Minimum Set of Constants

```
matchType G (K V) U s =
  let (G', T) = inferType G r 0 K V
  in searchCon G' U T s
matchType G [x:A]B U s =
  let s1 = matchType G A U s
  in matchType (G, x : A) B U s1
------------------------------------------------------------
matchType G U x s =
  let X = v_val G x
  in searchCon G U X s
matchType G x x' s =
  let T = v_type G x
  in searchCon G T x' s
matchType G (K V) x s =
  let (G', T) = inferType G 0 K V
  in searchCon G' T x s
matchType G [x:A]B x' s =
  let X' = v_val G x'
  in if X' == x'
     then error
     else x' : matchType G [x:A]B X' s
```

```
41 ------------------------------------------------------------
42 matchType G U (K V) s =
43   let (G', V', s') = searchFun G K V s
44   in searchCon G' U V' s'
45 matchType G x (K V) s =
46   let T = v_type G x
47   in searchCon G T (K V) s
48 matchType G (K1 V1) (K2 V2) s =
49   let (G', T) = inferType G 0 K1 V1
50   in searchCon G' T (K2 V2) s
51 matchType G [x:A]B (K V) s =
52   let (G', V', s') = searchFun G K V s
53   in matchType G' [x:A]B V' s'
54 ------------------------------------------------------------
55 matchType _ U [x:A]B _ = error
56 matchType G x [x:A]B s =
57   let T = v_type G x
58   in searchCon G T [x:A]B s
59 matchType G (K V) [x:A]B s =
60   let (G', T) = inferType G 0 K V
61   in searchCon G' T [x:A]B s
62 matchType G [x:A]B [x':A']B' s =
63   let s1 = matchType G A U s
64       s2 = searchCon G A A' s1
65   in matchType (G, x : A, x':A'=x) B B' s2
66
67 inferType :: Cont -> Int -> Exp -> Exp -> (Cont, Exp)
68 inferType _ _ U _ = error
69 inferType G 0 x V =
70   let T = v_type G x
71   in inferType G 1 T V
72 inferType G 1 x V = (G, x V)
73 inferType G 0 (K V) V' =
74   let (G', K') = inferType G 0 K V
75   in case K' of
76        U       -> error
77        [x:A]B  -> ((G', x : A = V'), B)
78        _       -> (G', K' V')
79 inferType G 1 (K V) V' = (G, (K V) V')
80 inferType _ 0 [x:A]B _ = error  -- not beta-normal form
81 inferType G 1 [x:A]B V = ((G, x : A = V), B)
82 inferType _ 0 (D M)  _ = error  -- not allowed in syntax
83 inferType G 1 (D M) V = inferType (G, D) 1 M V
84
85 {-|
86 * @param 1: the constant searching context that
87   relates variables to their type and values
88 * @param 2: an expression
89 * @param 3: an expression
90 * @param 4: a list of names which represents
91 *           the constants already discovered
92
93 * this function searches for the constants that must be unfolded
94 * for the possibility that m (the evaluated form of M) and n
95 * (the evaluated form of N) are convertible by the rules
96 * described in table  2.9 .
```

```
-}
searchCon :: Cont -> Exp -> Exp -> [String] -> [String]
searchCon G M [D]M' s = searchCon (G,D) M M' s
-----------------------------------------
searchCon G U U s = s
searchCon G U x s =
  let X = v_val G x
  in if X == x
     then error
     else searchCon G U X (s ++ [x])
searchCon G U (K V) s =
  let (G', V', s') = searchFun G K V s
  in searchCon G' U V' s'
searchCon _ U [x:A]B _ = error
-----------------------------------------
searchCon G x U s = searchCon G U x s
searchCon G x x' s =
  if x == x'
  then s
  else let X  = v_val G x
           X' = v_val G x'
       in if X == x && X' == x'
          then error
          else if X /= x && X' /= x' -- @open condition
               then s
               else if X /= x
                    -- only x can be unfolded
                    then searchCon G X x' (s ++ [x])
                    else searchCon G x X' (s ++ [x'])
searchCon G x (K V) s =
  let k' = head_v K
      X  = v_val G x
      K' = v_val G k'
  in if X == x && K' == k'
     then error
     else if X /= x && K' /= k' -- @open condition
          then s
          else if X /= x
               then searchCon G X (K V) (s ++ [x])
               else let (G', V', s') = searchFun G K V s
                    in searchCon G' x V' s'
searchCon G x [z:A]M s =
  let X = v_val G x
  in if X == x then error
     else searchCon G X [z:A]M (s ++ [x])
-----------------------------------------
searchCon G (K V) U s = searchCon G U (K V) s
searchCon G (K V) x s = searchCon G x (K V) s
searchCon G (K1 V1) (K2 V2) s =
  if K1 == K2
  then let (_, M1, _) = searchFun G K1 V1 s
           (_, M2, _) = searchFun G K2 V2 s
       in if M1 == M2 then s
       else searchCon G V1 V2 s
  else let (G1, M1, s1) = searchFun G K1 V1 s
           (G2, M2, s2) = searchFun G K2 V2 s
```

```
153         in if M1 == M2 then s1 ++ s2
154         else let (G', M2') = merge(G1, G2, M2)
155              in searchCon G' M1 M2' (s1 ++ s2)
156 searchCon G (K V) [x:A]B s =
157   let (G', V', s') = searchFun G K V s
158   in searchCon G' V' [x:A]B s'
159 ----------------------------------------
160 searchCon G [x:A]B U s = error
161 searchCon G [x:A]B y s = searchCon G y [x:A]B x
162 searchCon G [x:A]B (K V) s = searchCon G (K V) [x:A]B s
163 searchCon G [x:A]B [x':A']B's =
164   let s' = searchCon G A A' s
165       y  = fresh_v G x
166       G' = (G, x : A = y, x' : A' = y)
167   in searchCon G' B B' s'
168
169 {-|
170 * @param 1: constant searching context that relates
171   variables to their types and values
172 * @param 2: expression which is checked to represent a function
173 * @param 3: expression which is applied to the function represented
174 *           by @param 2
175 * @param 4: a list of names which represents the constants already
176 *           discovered
177 * @return:  1. a new constant searching context where variables
178 *              bound by the lambdas are bound to expression at
179 *              @param 3 in the constant searching context
180 *           2. function body after the lambda abstraction
181 *              being eliminated
182 *           3. the accumulated list of constants found
183 *              to be unlocked
184 *
185 * this function searches for the constants that must be unfolded
186 * for the possibility that the expression as the second
187 * argument is a function.
188 -}
189 searchFun :: Cont -> Exp -> Exp -> [String] -> (Cont, Exp, [String])
190 searchFun _ U _ _ = error
191 searchFun G x V s =
192   let X = v_val G x
193   in if X == x then error else searchFun G X V (s ++ [x])
194 searchFun G (K V) V' s =
195   let (G', K', s') = searchFun G K V s
196   in searchFun G' K' V' s'
197 searchFun G [x:A]B V s = ((G, x : A = V), B, s)
198
199 -- Given an constant searching context and a name, generates a new
200 -- variable that does not in the constant searching context
201 fresh_v :: Cont -> String -> String
202 fresh_v G x = if x not in G then x else x''
203
204 -- @param 1: an expression of K
205 -- Get the head variable of @param 1
206 head_v :: Exp -> Exp
207 head_v x      = x
208 head_v (K _)  = head_v K
```

```
209 head_v _       = error
210
211 -- @param 1: the constant searching context
212 -- @param 2: the name of a variable
213 -- Get the expression bound to the variable with
214 -- name @param 2 in context @param 1
215 v_val :: Cont -> String -> Exp
216 v_val (G, x' : A) x =
217   if x' == x then x else v_val G x
218 v_val (G, x' : A = B) x =
219   if x' == x then B else v_val G x
220
221 -- @param 1: the constant searching context
222 -- @param 2: the name of a variable
223 -- Get the type bound to the variable with
224 -- name @param 2 in context @param 1
225 v_type :: Cont -> String -> Exp
226 v_type (G, x' : A ) x    =
227   if x' == x then A else v_type G x
228 v_type (G, x' : A = B) x =
229   if x' == x then A else v_type G x
230
231 -- @param 1: context G1
232 -- @param 2: context G2
233 -- @param 3: an expression M
234 -- Merge G2 into G1 by renaming variables in G2
235 -- that name collides with that in G1.
236 -- Variables renamed in G2 are also renamed in
237 -- expression M for the purpose of mapping
238 merge :: Cont -> Cont -> Exp -> (Cont, Exp)
239 merge G1 G2 M =
240   for each declaration d in G2:
241     case d of
242       x : A -> if x in G1
243                then let x' = fresh_v(G1,x)
244                         M' = rename(x, x', M)
245                     in ((G1, x' : A), M')
246                else ((G1, x : A), M)
247       x : A = B ->
248         if x in G1
249         then let x' = fresh_v(G1, x)
250                  M' = rename(x, x', M)
251              in ((G1, x' : A = B), M')
252         else ((G1, x : A = B), M)
253   return the final result (G', M')
254
255 -- @param 1: name x1
256 -- @param 2: name x2
257 -- @param 3: an expression M
258 -- rename each free occurrence of variable x1 in M to x2
259 -- definition is ignored
260 rename :: String -> String -> Exp -> Exp
```

Listing 2.1: Approximation Algorithm For Minimum Set of Constants

The algorithm presented above is only an approximation because of the *open condi-*

tions marked by the comments in the pseudo-code, such as when checking the convertibility of two constants x, x' with different names but both having definitions. In cases like this, it is not clear which one should be unfolded to get a minimum set. Errors marked in the algorithm only occur when expression M does not have type t in terms of checking the formula $G \vdash_s M \Leftarrow t$. The *constant searching context* is basically the same with a type checking context except that we use it to hold both values and types of bound variables in function abstractions. Lastly, to use the function *matchType* on two expressions, we need to apply an operation called *readBack* to turn the q-expression t in the formula to an expression, which will be described in the following section.

2.6.2 Linear Head Reduction

Linear Head reduction was introduced in the calculus $\Delta\Lambda$ of AUTOMATH[8] and is demonstrated here as to show another way to limit computation. It forces expressions to be evaluated in "small steps" once a time instead of being fully evaluated. It features with a procedure where closures are eliminated so that the result of head reduction is an expression instead of a quasi-expression. We denote this function as δ and give its definition in table 2.11.

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

Table 2.11: Function - Head Reduction

The procedure to eliminate closures is achieved by a function called *readBack* which is 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.

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

Table 2.12: Function: readBack

For expressions in forms of K, an auxiliary function is used to head reduce this expression to a q-expression. We call this function *headRedV* and denote it as δ^*. The definition of *headRedV* is given in table 2.13.

The empty parentheses represents an empty environment. $\mathcal{V}(\Gamma, x)$ is a function to get the least evaluated form of variable x from context Γ. We call it *getVal* and give its definition in table 2.14. Note that to reduce an application $K\,N$, our approach

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

Table 2.13: Function - HeadRedV

is different with what is adopted by a Krivine machine[3]: instead of evaluating both the body and 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 2.14: Function - getVal

As an example of 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[9] 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.

[3] visit this website from wikipedia for an introduction.

3

Extension

We describe in this chapter an extension to our language: a module system based on the concept of 'segment' borrowed from the work of AUTOMATH. For an introduction to the usage of segment in AUTOMATH, we refer the readers to H. Balsters's work[4]. Here, we only illustrate its influence on our design of a module system by giving an example as follows. Note that in the following discussions, we use the words 'segment' and 'module' interchangeably.

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

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

where x 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,

$$s = [A : *,\ id : A \to A = [x : A]\ x]\ \texttt{Seg}$$

is a module which contains a declaration and a definition. The declaration $(A : *^1)$ is a parameter of the module and the definition id is the identity function defined in this module. Suppose we have another declaration $(A0 : *)$, then the expression $(s\,[A0]).id$ has $A0 \to A0$ as its type and closure $([x : A]x)(A = A0)$ as its value.

From this example we can see that a segment in our language is an abbreviation for a collection of declarations. We give the definition of segment by listing the relevant concepts and grammatical rules as follows.

Definition 3.0.1 (Segment)

- A segment can be declared as $x = \texttt{Seg}\ ds$ where x is the *name* of the segment and ds consisting of a list of declarations is the *content* of the segment.
- An *empty segment* is a segment whose content is an empty list. Declaration of an empty segment is allowed grammatically but of no practical use.
- 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 $\to$ to denote the parent-child relation such that $a \to b$ iff a is the parent of b.
- For the variables that are declared in the segment s, s is called their *declaring segment*.

[1] '*' represents U, the type of small types

- There is a *default segment* that is implicitly inhabited at the top-level and is denoted as *s-root*.
- The children segments and their children are called *descendants* to the parent segment; To the descendants, the parent segment and its parent up to *s-root* are called the *ancestors*.
- The *path* of a segment is the list of names that relate *s-root* to it 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 *s-root* is the empty list.
- 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 *s-root* is the empty string.
- 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 following discussion 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.
- The *relative path* of a segment s to an ancestor a is the list of names that relate a to s 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 $[b, c]$.
- 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 $s1$ is a segment where x is declared as a variable and $s2$ is declared as a segment; In $s2$, y is declared as a variable, then the relative namespace of x to $s1$ is the empty string and the relative namespace of y to $s1$ is "s2".
- A *parameter* of a segment is a declaration of the form $x : A$ in this segment.
- 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 s 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 with types A, B, C respectively.
- A segment can have no parameter. In that case, it can only be instantiated by an empty list and resulting in a new segment that is a replicate of itself.
- A segment can be declared by the instantiation of another segment, i.e., we have declarations of the form $s1 = s2\,[M_1, \ldots, M_n]$ where $s1$ is an instantiation

of $s2$.

- Objects (variables and segments) in a segment s 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 s whereas on the right is the name of the object. If the relative namespace is the empty string, which means the object is declared in s, it 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 segment of a by term $a.b.c.x$. This form of access is called the *direct access*.
- 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 $s_1. \ldots .s_n\,[M_1, \ldots, M_n]\,.\,x$, where $[s_1, \ldots, s_n]$ is the relative path of s_n to the parent of s_1, the segment where the access happens, and $[M_1, \ldots, M_n]$ are the expressions used to instantiate s_n.
- Expressions in a segment s can only refer to the entities from s or its descendants that have already been declared. This means that terms of the form $(s1\,[M_1, \ldots, M_i]\,. \ldots .s_n\,[N_1, \ldots, N_j]\,.\,x)$ is not necessary 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**.
- Name of a declaration cannot collide with names already existed in the same segment. There is no restriction on names from different segments.

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. Expression in the extended language is defined as follows:

Definition 3.1.1 (Expression)

(i) Terms of forms U, $[x : A]M$, $D\,M$ that are defined in 2.3.1 are expressions in the extended language with the same meaning.

(ii) Given a non-empty list of names $[s_1, \ldots, s_n]$, a name x and an empty-possible 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

$$s_1. \ldots .s_n\,[(M_1, x_1), \ldots, (M_i, x_i)]\,.\,x$$

is an expression and 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 s_n, whose relative path to the current segment is $[s_1, \ldots, s_n]$. In this case, x_1 to x_m represent the names of the parameters of s_n that should be bound to expressions M_1 to M_n correspondingly; otherwise it represents a *direct access* to the variable x in the segment s_n. Pairing each expression with the name of its corresponding parameter facilitates the evaluation and type checking process on expressions.

Declaration in the extended language is defined as follows.

Definition 3.1.2 (Declaration)

(i) Terms of the form $x : A$, $x : A = B$ that are defined in 2.3.2 are still declarations in the extended language and have the same meaning.

(ii) Given a name s and an empty-possible list of declarations ds, a term of the form

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

is a declaration which is used to declare a segment s consisting of the list of declarations ds. `Seg` in this case is a reserved word of the language.

(iii) Given a name s, a non-empty list of names $[s_1, \ldots, s_n]$ and an empty-possible list of tuples $[(M_1, x_1), \ldots, (M_i, x_i)]$, a term of the form

$$s = s_1. \ \ldots \ .s_n\,[(M_1, x_1), \ldots, (M_i, x_i)]$$

is a declaration which is used to declare a segment s by the instantiation of another segment s_n. The relative path of s_n to the current segment is $[s_1, \ldots, s_n]$.

A program of the extended language consists of a list of declarations which belong to the default segment *s-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 could be found int table 3.1.

$$\begin{array}{lll}
A, M & ::= & U \mid K \mid [x : A]M \mid D\,M \\
K & ::= & x \mid s_1. \ \ldots \ .s_n\,[(M_1, x_1), \ldots, (M_i, x_i)]\,.\,x \mid K\,M \\
D & ::= & x : A = M \\
S & ::= & s = \texttt{Seg}\ [Decl] \mid s = s_1. \ \ldots \ .s_n\,[(M_1, x_1), \ldots, (M_i, x_i)] \\
Decl & ::= & x : A \mid D \mid S \\
P & ::= & [Decl]
\end{array}$$

Table 3.1: Syntax of the Extended Language

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.

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

- $()$: ρ is an empty environment.
- $(\rho_1, x = v)$: ρ is an environment extended from ρ_1 by binding a variable x to a q-expression v, ρ shares the same path with ρ_1.
- (ρ_1, D): ρ is an environment extended from ρ_1 by a definition, ρ shares the same path with ρ_1.

- $(\rho_1, s = \rho')$: ρ is an environment extended from ρ_1 by a sub-environment ρ' which represents the child segment with name s, ρ shares the same path with ρ_1. If we denote the path of ρ as ρ_p, the path of ρ' is $\rho'_p = \rho_p + [s]$.

The definition of q-expression in the extended language is the same as 2.4.2 and we still use the notation $M_\rho = q$ to express that expression M is evaluated to q in environment ρ. Semantics of the extended language is given in table 3.2.

$$
\begin{array}{lcl}
U_\rho & = & U \\
(K\ N)_\rho & = & app(K_\rho, N_\rho) \\
([x : A]B)_\rho & = & \langle [x : A]B, \rho \rangle \\
(D\ M)_\rho & = & M_{(\rho, D)} \\
x_\rho & = & \rho(x) \\
(s_1. \ldots .s_n\,[(M_1, x_1), \ldots, (M_i, x_i)]\,.\,x)_\rho & = & \text{let } \rho_1 = \iota_\rho([s_1, \ldots, s_n], [(M_1, x_1) \ldots, (M_i, x_i)]) \\
 & & \text{in } \rho_1(x)
\end{array}
$$

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.

- $\rho(x)$: evaluate variable x in environment ρ.[2]
- $\iota_\rho(rp, ens)$: get the environment corresponding to the segment which is the result of instantiation on another segment by a list of tuples ens. The relative path of the segment being instantiated to ρ[3] 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.

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

Table 3.3: Function - $\rho(x)$

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

- *findSegEnv*(rp, ρ): find the environment ρ_1 whose relative path to ρ is rp. We use the notation $\rho_1 =\rightsquigarrow_{rp} \rho$ 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 $[a_1, \ldots, a_n]$.

[2] we overload this function with a new definition.

[3] 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.

- *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 evaluation on a list of expressions es in the environment ρ; $(\rho_1, \sum_i (x_i = v_i))$ represents the environment extended from ρ_1 by binding variables x_i from a list to q-expressions v_i from a list.

$$\begin{aligned} \iota_\rho(rp, ens) \quad = \quad & \text{let } \rho_1 =\rightsquigarrow_{rp} \rho,\ es = \mathit{mfst}(ens), \\ & ns = \mathit{msnd}(ens),\ qs = es_\rho \\ & \text{in } (\rho_1, \textstyle\sum_i (x_i = v_i)), x_i \in ns, v_i \in \\ & qs \end{aligned}$$

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.

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

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

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 function *getType* which is denoted as $\Gamma(s, x)$. 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 the segment cannot be referred inside 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 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 by 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 $\mathit{sname}(x_q)$ returns the short name x. The definition of $\Gamma(s, x)$ is given in table 3.5.

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

Table 3.5: Function - getType

For the type checking algorithm, the lock strategy in the extended language have the same meaning as that of the basic language except that variables to be locked now are 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

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

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

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

$$\frac{\begin{array}{c}\Gamma, \Gamma_0 \vdash_s (M_1, x_1) \Rightarrow \Gamma_1 \\ \vdots \\ \Gamma, \Gamma_{i-1} \vdash_s (M_i, x_i) \Rightarrow \Gamma_i\end{array} \quad \left(\begin{array}{l} rp = [s_1, \ldots\ s_n] \\ \Gamma_0 =\!\!\rightsquigarrow_{rp} \Gamma \end{array} \right)}{\Gamma \vdash_s s = s_1. \ \ldots\ .s_n\,[(M_1, x_1), \ldots, (M_i, x_i)] \Rightarrow (\Gamma, s = \Gamma_i)} \tag{3.4}$$

$\epsilon(\Gamma_p + [s])$ 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

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

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

3.3.3 checkT

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma,\Gamma_0 \vdash_s (M_1,x_1) \Rightarrow \Gamma_1 \\ \vdots \\ \Gamma,\Gamma_{i-1} \vdash_s (M_i,x_i) \Rightarrow \Gamma_i \qquad \Gamma_i(s,x) \sim_{\tau(\Gamma)} t\end{array}}{\Gamma \vdash_s s_1.\ \ldots\ .s_n\,[(M_1,x_1),\ldots,(M_i,x_i)].x \Leftarrow t} \left(\begin{array}{l} rp = [s_1,\ldots\ s_n] \\ \Gamma_0 =\!\rightsquigarrow_{rp} \Gamma \end{array} \right) \tag{3.12}$$

$\mathcal{Q}(\Gamma_p, x)$ 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

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

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

$$\frac{\begin{array}{c}\Gamma, \Gamma_0 \vdash_s (M_1, x_1) \Rightarrow \Gamma_1 \\ \vdots \\ \Gamma, \Gamma_{i-1} \vdash_s (M_i, x_i) \Rightarrow \Gamma_i\end{array}}{\Gamma \vdash_s s_1.\ \ldots\ .s_n\,[(M_1,x_1),\ldots,(M_i,x_i)].x \Rightarrow \Gamma_i(s,x)} \left(\begin{array}{l} rp = [s_1,\ldots,s_n] \\ \Gamma_0 = \rightsquigarrow_{rp} \Gamma \end{array} \right) \tag{3.15}$$

3.4 Linear Head Reduction

Function *linear head reduction* in the extended language has the same definition as that in table 2.11, so does the function *readBack*. The definition of *headRedV*, however, is different because of the introduction of segment.

$\delta^*(\Gamma, x)$	=	$\mathcal{V}(\Gamma, x)$
$\delta^*(\Gamma, s_1.\ \ldots\ .s_n$	=	let $rp = [s_1,\ldots,s_n]$, $\rho = \varrho([],\Gamma)$,
$[(M_1,x_1),\ldots,(M_i,x_i)]\,.\,x)$		$\rho_1 = \iota_\rho(rp, [(M_1,x_1),\ldots,(M_i,x_i)])$,
		in $\mathcal{V}(\rho_1, x)$
$\delta^*(\Gamma, K\,N)$	=	let $k = \delta^*(\Gamma, K)$, $n = N_{()}$ in $app(k,n)$

Table 3.7: Function - HeadRedV in Extended Language

Function *getVal* ($\mathcal{V}$) in the table 3.7 is overloaded to express: (1) $\mathcal{V}(\Gamma, x)$, the function to get the least evaluated form of variable x from context Γ; and (2) $\mathcal{V}(\rho, x)$, the function to get the least evaluated form of variable x from environment ρ. Different with that of table 2.14, when $\mathcal{V}$ is used to get the least evaluated form of a variable x form a context Γ, variable x 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* in both cases in table 3.8.

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 defined in appendix A.7. It shows the same result as appendix A.6 but performed with the involvement of segment.

$\mathcal{V}(\Gamma, x_q)$	$=$	let $rp = \mathit{locateseg}(\Gamma, x_q)$, $\Gamma' = \rightsquigarrow_{rp} \Gamma$, $x = \mathit{sname}(x_q)$ in $\mathcal{V}(\Gamma', x)$
$\mathcal{V}((), x)$	$=$	x
$\mathcal{V}(\Gamma', s = \Gamma_1)$	$=$	$\mathcal{V}(\Gamma', x)$
$\mathcal{V}((\Gamma', x' : A), x)$	$=$	if $x' == x$ then x else $\mathcal{V}(\Gamma', x)$
$\mathcal{V}((\Gamma', x' : A = B), x)$	$=$	if $x' == x$ then $B_{()}$ else $\mathcal{V}(\Gamma', x)$

$\mathcal{V}((), x)$	$=$	x
$\mathcal{V}(\rho', s = \rho_1)$	$=$	$\mathcal{V}(\rho', x)$
$\mathcal{V}((\rho', x' : A = B), x)$	$=$	if $x' == x$ then $B_{()}$ else $\mathcal{V}(\rho', x)$

Table 3.8: Function - getVal

4

Results

The result of the project is a REPL (read-evaluate-print-loop) program developed in Haskell where a file of our language could be loaded and type checked. The program features a static context and a dynamic context where 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

We have presented in this paper a language of dependent type theory as an extension to the pure lambda calculus with dependent types and definitions. We studied and implemented in the language a definition mechanism 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, i.e., 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 the syntactic identity, could be adapted to support new language features.

Future work based on this project could be conducted on 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.

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.

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] 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.

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

[6] 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.

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

[8] 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.

[9] 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

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

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

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

$$\frac{\Gamma \vdash_s x \Rightarrow \langle [z:A]M, \rho\rangle \qquad \Gamma \vdash_s A_\rho \sim A'_{\rho'} \Rightarrow \_ \qquad \Gamma' \vdash_s q1 \sim q2 \Rightarrow t}{\Gamma \vdash_s x \sim \langle [x':A']B', \rho'\rangle \Rightarrow t} \left(\begin{array}{lcl} y & = & \nu(\tau(\Gamma), x) \\ \Gamma' & = & (\Gamma, y : A_\rho) \\ q1 & = & x\ y \\ q2 & = & B'_{(\rho', x'=y)} \end{array} \right) \tag{A.4}$$

$$\frac{\Gamma \vdash_s k \Rightarrow \langle [z:A]M, \rho\rangle \qquad \Gamma \vdash_s A_\rho \sim A'_{\rho'} \Rightarrow \_ \qquad \Gamma' \vdash_s q1 \sim q2 \Rightarrow t}{\Gamma \vdash_s (k\ v) \sim \langle [x':A']B', \rho'\rangle \Rightarrow t} \left(\begin{array}{lcl} y & = & \nu(\tau(\Gamma), x') \\ \Gamma' & = & (\Gamma, y : A_\rho) \\ q1 & = & (k\ v)\ v \\ q2 & = & B'_{(\rho', x'=y)} \end{array} \right) \tag{A.5}$$

$$\frac{\Gamma \vdash_s A_\rho \sim A'_{\rho'} \Leftarrow U \qquad (\Gamma, y : A_\rho), \vdash_s B_{(\rho, x=y)} \sim B'_{(\rho', x'=y)} \Rightarrow t}{\Gamma \vdash_s \langle [x:A]B, \rho\rangle \sim \langle [x':A']B', \rho'\rangle \Rightarrow t} \left(\begin{array}{lcl} y & = & \nu(\tau(\Gamma), x_1) \end{array} \right) \tag{A.6}$$

A.2.2 CheckCT

$$\frac{(\Gamma, y : A_\rho) \vdash_s (v1\ y)_{\varrho(s,\Gamma)} \sim (v2\ y)_{\varrho(s,\Gamma)} \Leftarrow B_{(\rho, x=y)}}{\Gamma \vdash_s v1 \sim v2 \Leftarrow \langle [x:A]B, \rho\rangle} \left(\begin{array}{lcl} y & = & \nu(\tau(\Gamma), x) \end{array} \right) \tag{A.7}$$

$$\frac{\Gamma \vdash_s v1 \sim v2 \Rightarrow t' \qquad \Gamma \vdash_s t \sim t' \Rightarrow \_}{\Gamma \vdash_s v1 \sim v2 \Leftarrow t} \tag{A.8}$$

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> ∼ <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