CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Haskell Implementation for a Dependently Typed Language

Master's thesis in Computer science and engineering

QUFEI WANG

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

Master's thesis 2021

A Haskell Implementation for a Dependently Typed Language

QUFEI WANG

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

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

A Haskell Implementation for a Dependently Typed Language
QUFEI WANG

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

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

Typeset in LaTeX
Gothenburg, Sweden 2021

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

Abstract

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

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

Acknowledgements

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

Qufei Wang, Gothenburg, September 2021

Contents

1

Terminology & Convention

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

- **Declaration**: A *declaration* has either the form $x : A$ or $x : A = B$. The latter is also referred, rather frequently, as a *definition*. Sometimes when we want to make a distinction between these two forms, we also use the word 'declaration' specifically to indicate a term of the former form.
- **Definition**: A *definition* is a term of the form $x : A = B$, meaning that x is an element of type A, defined as B. Sometimes when we want to talk about the components of a specific definition, we also use the word 'definition' specifically to indicate the part of B, e.g., "the definition of x is B" .
- **Constant**: A *constant* is the entity that a name or identifier used in a declaration refers to, e.g., the entity that x in $x : A$ or $x : A = B$ refers to.
- **Variable**: A synonym for *constant*. More often, the word 'variable' is used to refer to the variable bound in a λ-abstraction, like the variable x in $\lambda x.A$. In most cases, these two words are interchangeable.
- **Name, String and Entity**: When we say "the name of a variable x" or "the name of a segment s", we refer to the string as an identifier that is used in the source file to declare this variable or segment. For example, if we have a line of text in our source file like this,

  x : A = B

  then when we talk about it in discussions within this paper, we could say the *name* of variable x is "x". We use double quotes around the character 'x' to emphasize that "x" as a name is a *string*, whereas x without quotes is the *entity* as a variable that the name "x" refers to. Adhering to this notation meticulously is cumbersome when we talk about a list of names. So for a list of variables, say $[x, y, z]$, instead of denoting their names as ["x", "y", "z"], we simply say that the list of the names of these variables is $[x, y, z]$.

2

Introduction

2.1 Background About Dependent Types

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

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

2.2 Issues with Dependent Types

The downside of dependent type systems lies in the difficulties of implementation, among which is the notable problem about checking the **convertibility** of terms: given two terms A, B, decide whether they are equal or not. Checking the convertibility of terms that represent types is a routinely performed task by the type checker of any typed language, thus is vital for its performance. In a simple typed language, convertibility checking is done by simply checking the syntactic identity of the symbols of the types. For example, in Java, a primitive type *int* equals only to itself, nothing more. This is because types in Java are not computable [1]:there's

[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.

no way for other terms in Java be reduced to the term *int*. In a dependently typed language, however, the problem is more complex since a type may contain terms of any expression, deciding the convertibility of types in this case entails evaluation of any terms, which requires much more computation.

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

- Producing the normal form may require more reduction steps than necessary. For example, in proving $(1+1)^{10} = 2^{(5+5)}$, it is easier if we can prove $1+1 == 2$ and $5+5 == 10$ instead of having to reduce both sides to 1024 by the definition of exponentiation.
- As the number of definition grows, the size of terms by expanding a definition can grow very quickly. For example, the inductive definition $x_n := (x_{n-1}, x_{n-1})$ makes the normal form of x_n grow exponentially.

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

2.3 Aim of the Project

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

A definition in a dependently typed language is a declaration of the form $x : A = B$, meaning that x is a constant of type A, defined as B. The problem with definitions is not about how a constant should be declared, but how it should be evaluated. Evaluation, or reduction, in dependent type theory has its concept originated in λ-*calculus* [6]. There, a term of the form $(\lambda x.M)N$ can be evaluated (or reduced) to the form $M[x := N]$, meaning the substitution of N for the free occurrences of x in M[2]. In dependent type theory, however, different evaluation strategies can have huge difference regarding the efficiency of evaluation.

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

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

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

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

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

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

2.4 Organization of This Paper

This paper is organized as follows: In chapter 3, we first describe the common pitfalls one should avoid in the implementation of a dependently typed language. Based on the pitfalls, we put forward the principles that guide us through the implementation of our system. The rest of the chapter 3 devotes to a detailed description of the basic form of our language: a pure λ-calculus extended with a universe of small types(U), dependent types and definitions. We present in detail its syntax, semantics, type checking rules and the definition mechanism plus some related important operations. In chapter 4, we present an extension of our language with the introduction of a

module system. In a same way as in chapter 3, we describe the extended syntax, semantics, type checking rules and the related important operations. Particularly, we draw our attention to the comparison with chapter 3 where operations related with the definition mechanism need to be adjusted for the 'namespaces' concept brought in by the module system. Chapter 5 presents as results a REPL program that could be used to load and type check a source file and to experiment with the definition mechanism. Chapter 6 concludes the paper with a short review of the project.

2.5 Limitations of the Project

The limitations of our work come into three aspects: expressiveness, scope and metatheory.

1. **Expressiveness/Usability:** We try to keep the syntax of the language as simple as possible in order to focus on the study of a definition mechanism. This practice inevitably affects the expressiveness and usability of our language: As has been mentioned, there is no syntax in support for creating new data types, defining recursive functions or doing pattern match on expressions; Further, there is a lack of basic input/output functionalities(one cannot even print a "Hello,world!" on the screen); Besides, because we track the names of constants in a linear manner as an approach to the name collision problem (see example 2 in section 3.1) and enforce the policy that declaration of names must not collide with the names in the current context, the common programming language feature *variable shadowing* does not exist in our language; Lastly, the result of the project is only a REPL that incorporates a type checker(along with a lexer and a parser which are automatically generated by the BNF convertor) and an interpreter, the former to load and type check a source file and the latter to evaluate expressions. There is no support in the view of a compiler that generates executable machine code(of course for a language worthy of executable program compilation, it must have basic data types and I/O operations).
2. **Scope:** The definition mechanism we established is not meant to be the most effective or applicable universally to different kinds of dependently typed languages. This means if we were to develop a more practical dependently typed language, then (1) the definition mechanism along with the type checking algorithm must be modified, perhaps dramatically, to accommodate the semantics brought in by the new language features; (2) the definition mechanism is subject to improvement by utilizing other computation control techniques; (3) the type checking algorithm is subject to improvement by optimization.
3. **Metatheory:** The properties of our language as a type theory, such as the decidability of our type checking algorithm, will not be covered in this paper. The connections between our work and other logical systems from type theory will also be ignored. Although most of the ideas used in this work originated from the work AUTOMATH, a comprehensive exposition of AUTOMATH and the connections between these two works will not be drawn here. A careful

study of the metatheory of this project and its links with other logical systems in a broader picture of type theory could be viewed as a direction for the future work.

3

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 need first to address some subtleties that differentiate our system from λ-*calculus* and that back our choice for dealing with the problems about names of the constants.

3.1 Subtleties of Dependent Type Theory

We present the subtleties by giving examples as the follows:

Example 1. Suppose we have

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

where a declaration of the form $x : A$ declares a constant x to be of type A; U is the type of small types. Then the term

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

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

However, if we modify this term to

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

where a definition of the form $x : A = B$ declares a constant x to be of type A and defined as term B, then it becomes type checked since now the type of y is exactly $(P\,a)$.

We see from this example that the definition of the bound variable (x) has an impact on the type safety of the whole expression, meaning that definitions in dependent type theory cannot be reduced to λ-*calculus*.

Example 2. Consider the formula

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

In this formula, the first declaration of x is shadowed by the second one. If we do not treat the shadowing of name x carefully and substitute x for y later when we do computations on M, then taking the type of x to be *Bool* will make M ill-typed.

This example shows that improper treat of name shadowing leads to mistakes in a dependently typed language.

Example 3. Suppose we have

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

Then the term below is well typed.

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

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

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

At this point, we have a capture of variables problem.
(3.2) should be the same as

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

which reduces to

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

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

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

which is not correct.

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

3.2 Principles of Definition Implementation

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

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

Principle 2. Forbid the shadowing of variable names.

Principle 3. Rename variable whenever necessary.

Principle 1 relates to example 1. The reason why we need to treat the definition and the type of a constant differently is that they take unequal roles in dependent type theory. Sometimes the definition of a constant is crucial to the type safety of an expression, as suggested by example 1, but this is not always the case. Consider the following expression

$$\lambda(f : A \to B)(a : A).f\ a$$

Neither the definition of f nor a is needed to ensure the type correctness of the lambda body $(f\ a)$: f could be any function from A to B and a could be any element of A, but we know for sure that the type of $(f\ a)$ is B.

These two examples illustrate the first aspect of the inequality between the type and the definition in dependent type theory: during the type checking process, the type is indispensable while the definition is only supplementary. This is also the reason why *locking* of definitions is possible during the type checking process.

The other aspect of the inequality between the type and the definition lies in the evaluation process, where only the definition matters whereas the type is discarded altogether. We will describe the evaluation process in more detail when we introduce the semantics of our language.

Principle 1 forms the basis of our definition mechanism.

Principle 2 comes as a simple strategy to avoid the pitfall revealed by example 2. During the type checking process, each declaration, including the declarations of bound variables from λ-abstraction, is checked with the top level context to ensure no name clashing occurs. Using *De Bruijn indices* is another way to avoid this issue. However, having to maintain a relation between the names and indices will complicate our implementation unnecessarily and distract us from the main aim of the project.

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

Finally, we have a fourth principle in support of our definition mechanism:

Principle 4. Defer computation.

For the efficiency of our type checking algorithm, we take the following measures to perform as few reductions as possible during the type checking process:

1. Use *closure* to pass functions around as values.
2. Apply β-reduction on multi-variable functions in an incremental manner.
3. Build a locking mechanism where definitions of certain constants could be temporarily erased.

Having introduced all these 4 principles, now we are ready to describe in detail the basic form of our language, its syntax, semantics, type checking algorithm and important operations.

3.3 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, which is 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.1.

Expression in our language is defined as follows:

Definition 3.3.1 (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 constant/variable is an expression, e.g., x, y, z.

(iii) Given two expressions M, N, a term of the form

$$M\ N$$

is an expression, which is used to represent function application.

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

$$[x : A]B$$

is an expression, which is used to represent either

- **λ-abstraction**: $\lambda[x : A].B$ - a function that given an argument x of type A, returns a term B which may dependent on x;

or

- **Dependent Product**: $\Pi x : A.B$ - the type of functions that given an argument x of type A, returns an object of type B which may dependent on x. If B does not dependent on x(x does not occur freely in B), then x could be ignored, giving us a term in the form $\Pi \_ : A.B$. This is essentially the same as the type of functions $A \to B$.

(v) 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 3.3.2 (Declaration)

(i) Given a name x and an expression A, a term of the form

$$x : A$$

is a declaration which is used to declare a variable x of type A.

(ii) Given a name x and two expressions A, B, a term of the form

$$x : A = B$$

is a declaration which is used to declare a variable x of type A and is 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 being used. A summary of the syntax could be found in table 3.1.

Exp	E	::=	$U \mid x \mid E_1\ E_2 \mid [x : E_1]E_2 \mid [x : E_1 = E_2]E^*$
Decl	D	::=	$x : E \mid x : E_1 = E_2$
Prog	P	::=	$[D]$

Table 3.1: Syntax of the Language

The syntax of our language is a subset of Mini-TT[8]. Moreover, we use the same syntax for both dependent product and λ-abstraction as an effort to maintain simplicity. This practice causes ambiguity only when an expression in the form $[x : A]M$ is viewed in isolation: it can be seen both as a dependent type and a function abstraction. This ambiguity, however, does not cause problem in usage because the meaning of a term could be deduced from its context, our type checking algorithm ensures the consistency of its usage.

3.4 Operational Semantics

Given a well-formed expression of our language, 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* given an evaluation context called *environment*. An environment is a stack that keeps track of the variables and their binding q-expressions and can be represented in one of the following forms:

Definition 3.4.1 (Environment)

(i) $()$, an empty environment where no variable exists.

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

(iii) $(\rho_1, x : A = B)$, an environment extended from a smaller environment ρ_1 by adding a definition.

Intuitively, a *q-expression* is the intermediate form of an expression under evaluation that can be converted to a "normal" expression by a "readback" procedure which we will describe in section 3.6.2. Sometimes we also call q-expressions as *values*.

Definition 3.4.2 (Q-expression)

(i) $\mathcal{U}$ is a Q-expression, it represents the result of U under evaluation.

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

(iii) A closure $\langle [x : A]M, \rho \rangle$ is a Q-expression, it represents a function equipped with an evaluation environment.

(iv) Given two Q-expressions k, v, k is not a closure, a term of the form $(k\,v)$ is a Q-expression. It represents an application that cannot be reduced further because either (1) k is not a function($k = \mathcal{U}$), or (2) k is a primitive constant

which could be a function but the definition is currently inaccessible (either because of a lack of definition or the definition is locked).

The evaluation operation is given by equations of the form $[\![M]\!]\rho = q$, meaning that the expression M evaluates to q under the environment ρ.

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

Table 3.2: Semantics of the Language

Two auxiliary functions are used in the evaluation operation:

- $\rho(x)$: find the binding q-expression of x from the environment ρ.
- $\alpha(k, v)$: apply function application of k to v.

Definitions of these two functions are given in table 3.3, 3.4.

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

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

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

Table 3.4: Function - $\alpha(k, v)$

It is not hard to see there is a correspondence between the form of an expression and that of an q-expression: U to $\mathcal{U}$; x to x_1, the former being a constant which may have a definition whereas the latter being a primitive constant whose definition is inaccessible; $M\,N$ to $k\,v$; and finally, $[x : A]B$ to $\langle [x : A]B, \rho \rangle$. The fact that expressions and q-expressions have similar structure enables us to use the same data type to represent both concepts in our implementation, providing a minor modification to the syntax of expression by adding the form that represents closure. Now a more complete summary of the syntax of our language is given in table 3.5.

So we use U for both the type of all small types and $\mathcal{U}$; variables like x, y, z for both constants and primitive constants; $M\ N$ for both function application and the result of its evaluation when M cannot be resolved to a function; $[x : A]B$ for abstraction, $\langle [x : A]B, \rho \rangle$ for closure and $[x : A = B]M$ for a *let* clause.

Mixing expression and q-expression together in one syntax does not cause problem in the implementation as long as we keep in mind which to expect in a certain operation.

Exp/Q-Exp	E	::=	$U \mid x \mid E_1\ E_2 \mid [x : E_1]E_2 \mid \langle [x : E_1]E_2, \rho \rangle \mid [x : E_1 = E_2]E^*$
Decl	D	::=	$x : E \mid x : E_1 = E_2$
Prog	P	::=	$[D]$

Table 3.5: Syntax of the Language(2)

For example, in the operation "readback" where a q-expression is converted back to an expression by eliminating any closures, an error will be raised if it is provided with an abstract or let clause. At the cost of being more careful, we get a more elegant implementation with less code duplication.

Some readers may have noticed that the real difference between expression and q-expression is closure, that is, we use closure to represent the result of evaluation on function abstractions. Closure is an important concept in functional programming and was first conceived by P. J. Landin in his paper *The Mechanical Evaluation of Expressions*[9]. 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 that we adhere to in our implementation: a λ-abstraction and the environment in which it is evaluated, as indicated in table 3.2. Introducing q-expression as a parallel but distinct concept from expression is motivated by the need to pass functions as values around in the evaluation which is achieved by using closure. Apart from this, using closures brings us two additional features that is important to our definition mechanism: (1) deferred evaluation and (2) partial application.

The meaning of deferred evaluation comes into twofold: evaluation of the body of a λ-abstraction and the substitution process in a β-reduction. The first is expressed clearly in the rule about the evaluation of a function abstraction: the body of the abstraction is left intact in the closure; The second is manifested in the function $\alpha(k, v)$, where the application of a closure with an argument results in the evaluation of the function body in a new environment. In the new environment, the head variable (x) of the closure is bound with the provided argument. In this case, if the function body (B) is another abstraction, then the substitution will not take place and be deferred to a later phase when the evaluation on the head variable is really needed. Being able to defer computation is crucial for our definition mechanism for it allows us to do as few reductions as possible in the type checking process. We will come back to this point whenever relevant in the later part of this paper.

Partial application is the idea that for a function of type $A \to B \to C$, we can provide it with only one argument $a \in A$ to get another function as the result, whose type is $B \to C$. This is also an important feature for our definition mechanism since it allows us to do reductions on multi-variable functions step by step.

The last thing we want to beg for the readers attention before we move on is the second aspect of the inequality between the type and the definition: notice how the type information is altogether discarded in the evaluation process, both when we do reduction on function application and in function $\rho(x)$.

3.5 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 A as its type. 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 error or semantic error related with names, such as duplicated declaration of names or use of undeclared name. Syntactic error is checked by the lexer and parser where a source file is parsed to a syntax tree represented by the concrete syntax mentioned at the beginning of section 3.3. Semantic error regarding names are checked when the concrete syntax tree is translated to an abstract syntax tree in a procedure we called *translation*. It is the abstract syntax tree to which the type checking algorithm is applied.

There are five forms of judgments.

checkD	$\Gamma, s \vdash D \Rightarrow \Gamma'$	D is a valid declaration and extends Γ to Γ'
checkI	$\Gamma, s \vdash M \Rightarrow t$	M is a valid expression and its type is inferred to be t
checkT	$\Gamma, s \vdash M \Leftarrow t$	M is a valid expression given type t
checkCI	$\Gamma, s \vdash u \equiv v \Rightarrow t$	u, v are convertible and their type is inferred to be t
checkCT	$\Gamma, s \vdash u \equiv v \Leftarrow t$	u, v are convertible given type t

Table 3.6: Type Checking Judgments

One thing deserves the readers' special attention: the lower case letters t, u, v appeared in table 3.6 represent q-expressions. This means that (1) the type inferred from a judgment(checkI or checkCI) and the type given as an input to a judgment(checkT, checkCT) must be in the evaluated form; (2) Only the evaluated form of expressions could be checked for convertibility. One can check this statement by the details of the judgments introduced later in the paper.

Each of the judgments is provided with two constructs acting as a premise: a type checking context Γ and a lock strategy s. A type checking context is a stack that keeps track of the types and definitions of all variables declared so far and is represented in one of the following forms:

Definition 3.5.1 (Type Checking Context)

(i) () - an empty context where no declaration exists.

(ii) $(\Gamma_1, x : A)$ - a context extended from a smaller context Γ_1 by binding a variable x to its type A.

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

Given a type checking context, we can query the type of a variable by function $\Gamma(x)$, which means to find the type of x in Γ. Function $\Gamma(x)$ is defined in table 3.7.

The first case of the function is worth some explanation. It expresses that if we try to get the type of a variable that does not exist in the type checking context,

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

Table 3.7: Function - $\Gamma(x)$

then an error is raised. In our 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.

Lock strategy is introduced as part of our definition mechanism to enable the locking/unlocking functionality on variables. A variable is *locked* when its definition(provided that it has one) is temporarily erased and *unlocked* if its definition is restored. A locked variable is in effect a primitive 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) variables by removing(restoring) their definitions from(to) the environment. This suggests a transformation from a type checking context to an environment with the definitions of variables being intentionally erased or restored. This is achieved by a function *getEnv* that given a lock strategy and a type checking context returns an environment, which is denoted by the symbol ϱ.

In the current implementation, we have four lock strategies: *LockAll, LockNone, LockList s, UnLockList s*, where s is a list of variable names. We denote these four strategies as $\bullet, \circ, \dagger(s), \ddagger(s)$ correspondingly and give the definition of ϱ in table 3.8.

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

Table 3.8: Function: getEnv

Initially, the type checking context is empty. Whenever a declaration is checked to

be valid, it is added to the underlying type checking context. We denote this process by $\Gamma, s \vdash D \Rightarrow \Gamma'$ and call it that D is a valid declaration and extends Γ to Γ'. When D has the form $x : A$, $\Gamma' = (\Gamma, x : A)$; otherwise when D has the form $x : A = B$, $\Gamma' = (\Gamma, x : A = B)$. Having introduced the relevant concepts and notations, we show the details of each of the five judgments in sequence.

3.5.1 checkD

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

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

- For a declaration of the form $x : A$, we check that A is a valid expression and has type U.
- For a definition of the form $x : A = B$, we check that
  1. A is a valid expression and has type U.
  2. B is a valid expression and has the evaluated form of A as its type.

3.5.2 checkI

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

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

$$\frac{\Gamma, s \vdash M \Rightarrow \langle [x : A]B, \rho \rangle \qquad \Gamma, s \vdash N \Leftarrow a}{\Gamma, s \vdash M\ N \Rightarrow b} \begin{pmatrix} a & = & [\![A]\!]\rho \\ \rho_0 & = & \varrho(s, \Gamma) \\ n & = & [\![N]\!]\rho_0 \\ \rho_1 & = & (\rho, x = n) \\ b & = & [\![B]\!]\rho_1 \end{pmatrix} \tag{3.7}$$

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

- The inferred type of U is itself.
- The inferred type of a variable x is the evaluated form of its type got from the type checking context.
- For application $M\ N$, we do as follows:
  1. Check M is a valid expression and its type can be inferred to be a closure.

2. Check N is a valid expression given the type of the head variable of the closure.
3. Get the environment extracted from the current context Γ, denote it as ρ_0.
4. Get the evaluated form of N from ρ_0, denote it as n.
5. Extend ρ to ρ_1 by binding x to n.
6. Return the evaluated form of B in ρ_1.

- For a let clause $[x : A = B]M$, we first check the definition is correct, then infer the type of M under the new context extended by the definition.

3.5.3 checkT

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

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

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

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

$$\frac{\Gamma, s \vdash x : A \Rightarrow \Gamma_1 \qquad \Gamma, s \vdash a \equiv a' \Rightarrow \_ \qquad \Gamma_1, s \vdash B \Leftarrow b'}{\Gamma, s \vdash [x : A]\, B \Leftarrow \langle [x' : A']\, B', \rho \rangle} \begin{pmatrix} \rho_0 & = & \varrho(s, \Gamma) \\ a & = & [\![A]\!]\rho_0 \\ a' & = & [\![A']\!]\rho \\ \rho_1 & = & (\rho, x' = x) \\ b' & = & [\![B']\!]\rho_1 \end{pmatrix} \tag{3.13}$$

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

- U is a valid expression given itself as its type.
- To check that a variable x is valid given v as its type, we do as follows:
  1. Check that x is valid and its type can be inferred to be v'.
  2. Check that v and v' are convertible.
- To check that $M\,N$ is a valid expression given v as its type, we do as follows:
  1. Check that $M\,N$ is a valid expression and its type can be inferred to be v'.
  2. Check that v and v' are convertible.

- To check that $[x : A]B$ is a valid expression given U as its type, we do as follows:
  1. Check that the declaration $x : A$ is valid and extends Γ to Γ_1.
  2. In context Γ_1, check that B is a valid expression given U as its type.
- To check that $[x : A]B$ is a valid expression given a closure $\langle [x' : A']\, B', \rho \rangle$ as its type, we do as follows:
  1. Check that the declaration $x : A$ is valid and extend Γ to Γ_1.
  2. Get the environment from Γ, denote it as ρ_0.
  3. Get the evaluated form of A from ρ_0, denote it as a.
  4. Get the evaluated form of A' from ρ, denote it as a'.
  5. Check that a and a' are convertible.
  6. Extend ρ to ρ_1 by binding x' to x.
  7. Get the evaluated form of B' from ρ_1, denote it as b'.
  8. In context Γ_1, check that B is a valid expression given b' as its type.
- To check that $[x : A = B]M$ is a valid expression given t as its type, we do as follows:
  1. Check that the definition $x : A = B$ is valid and extend Γ to Γ_1.
  2. In context Γ_1, check that M is a valid expression given t as its type.

Note that the inference rule 3.12 and 3.13 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 dependent product represented by another abstraction.

3.5.4 checkCI

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

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

$$\frac{\Gamma, s \vdash m_1 \equiv m_2 \Rightarrow \langle [x : A]B, \rho \rangle \qquad \Gamma, s \vdash n_1 \equiv n_2 \Leftarrow a}{\Gamma, s \vdash (m_1\ n_1) \equiv (m_2\ n_2) \Rightarrow v} \begin{pmatrix} a & = & [\![A]\!]\rho \\ \rho_1 & = & (\rho, x = n_1) \\ v & = & [\![B]\!]\rho_1 \end{pmatrix} \tag{3.17}$$

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

- U is only convertible to itself and has type U.
- A variable is only convertible to itself and the type is inferred to be the evaluated form of its type got from the context.

- To check that two q-expressions $m_1\ n_1$ and $m_2\ n_2$ are convertible and infer their type, we do as follows:
  1. Check that m_1 and m_2 are convertible and the type is inferred to be a closure $\langle [x : A]B, \rho \rangle$.
  2. Get the evaluated form of A from the environment ρ, denote it as a.
  3. Check that n_1 and n_2 are convertible given a as their type.
  4. Extend ρ to ρ_1 by binding variable x to n_1.
  5. Return the evaluated form of B from ρ_1 as the inferred type.
- To check that two closures are convertible and have U as their type, we check that they are convertible given type U.

The last inference rule (3.18) is only used when two terms as dependent product type ($\Pi x : A.B$) are checked for convertibility. In this case, they have type U, not a type of dependent product as in the case when a λ-abstraction is used to express a function. This reflects a 'two-tier' type structure of our system: Only U and dependent products can be used as types.

3.5.5 CheckCT

$$\frac{\Gamma_1, s \vdash m \equiv n \Leftarrow b}{\Gamma, s \vdash v1 \equiv v2 \Leftarrow \langle [x : A]\, B, \rho \rangle} \begin{pmatrix} a & = & [\![A]\!]\rho \\ y & = & \nu(\Gamma, x) \\ \Gamma_1 & = & (\Gamma, y : a) \\ \rho_0 & = & \varrho(s, \Gamma) \\ m & = & [\![v1\ y]\!]\rho_0 \\ n & = & [\![v2\ y]\!]\rho_0 \\ \rho_1 & = & (\rho, x = y) \\ b & = & [\![B]\!]\rho_1 \end{pmatrix} \tag{3.19}$$

$$\frac{\Gamma, s \vdash a_1 \equiv a_2 \Leftarrow U \qquad \Gamma_1, s \vdash b_1 \equiv b_2 \Leftarrow U}{\Gamma, s \vdash \langle [x_1 : A_1]\, B_1, \rho_1 \rangle \equiv \langle [x_2 : A_2]\, B_2, \rho_2 \rangle \Leftarrow U} \begin{pmatrix} a_1 & = & [\![A_1]\!]\rho_1 \\ a_2 & = & [\![A_2]\!]\rho_2 \\ y & = & \nu(\Gamma, x) \\ \rho_1' & = & (\rho_1, x_1 = y) \\ \rho_2' & = & (\rho_2, x_2 = y) \\ b_1 & = & [\![B_1]\!]\rho_1' \\ b_2 & = & [\![B_2]\!]\rho_2' \\ \Gamma_1 & = & (\Gamma, y : a_1) \end{pmatrix} \tag{3.20}$$

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

A new function ν is introduced in rule 3.19 and 3.20. Given a type checking context Γ and a variable x, it returns a new variable y that does not exist in Γ. Generating a new variable comes from the need to do reductions on closures so that the body of a λ-abstraction could be checked for convertibility. Because the names in a type

checking context must be unique, we need to apply a closure with a new variable to respect this rule.

- To check that two q-expressions $v1$ and $v2$ are convertible and has type $\langle [x : A]B, \rho \rangle$, we do as follows:
  1. Get the evaluated form of A from ρ, denote it as a.
  2. Get a new variable y.
  3. Extend Γ to Γ_1 by binding y to a.
  4. Get the environment from Γ, denote it as ρ_0.
  5. Evaluate $(v1\ y)$ in ρ_0, denote the result as m.
  6. Evaluate $(v2\ y)$ in ρ_0, denote the result as n.
  7. Extend ρ to ρ_1 with x bound to y .
  8. Get the evaluated form of B from ρ_1, denote it as b.
  9. In context Γ_1, check that m, n are convertible given b as their type.
- To check that two closures are convertible and have type U, we do as follows:
  1. Get the evaluated form of A_1 from ρ_1, denote it as a_1.
  2. Get the evaluated form of A_2 from ρ_2, denote it as a_2.
  3. Check that a_1 and a_2 are convertible given type U.
  4. Get a new variable y.
  5. Extend ρ_1 to ρ_1' by binding x_1 to y.
  6. Extend ρ_2 to ρ_2' by binding x_2 to y.
  7. Get the evaluated form of B_1 from ρ_1', denote it as b_1.
  8. Get the evaluated form of B_2 from ρ_2', denote it as b_2.
  9. Extend context Γ to Γ_1 by binding variable y to a_1.
  10. In context Γ_1, check that b_1, b_2 are convertible given U as their type.
- If the first two rules do not apply, we check $v1$ and $v2$ are convertible and infer their type as t', then we check t and t' are convertible.

3.6 Definition Mechanism

Having introduced the semantics and type checking rules of our language, now it is time to give a detailed description of our definition mechanism and summarize its characteristics.

The primary motivation behind our attempt 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 equal. This is the convertibility problem we introduced in section 2.2. Difficulty arises in dependent type theory because (1) a type may contain **any** expression which could entail large amount of computation, and (2) definition opens up the possibility to denote arbitrary complex

computation by a single constant. For the type checker of a dependently typed language to be efficient and thus practical, during the type checking process, it should not exceed too much the amount of reductions that are "just enough" to establish the equivalence between terms. The problem is, there is no standard way to calculate the minimum number of reductions needed, it dependents on the semantics of the language being used, namely the language designer's perception about how computation should be performed. 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 integer multiplication and exponentiation, and determine that in the evaluation process, an expression should be reduced to the normal form(i.e., no more function application could be applied), then a type checker loyal to our conception of computation will reduce both terms to 1024 to conclude that they are convertible. However, if we change our mind and see exponentiation as a primitive with no definition, then the same type checker with our updated conception about computation would only reduce both terms to 2^{10}.

For readers who have seen the semantics and type checking rules of our language, our reiteration of the example and idea from section 2.3 above is to show them that, the incorporating a locking functionality on constants is the most basic characteristic of our definition mechanism, and this mechanism is not a modularized feature with a clear boundary with the rest of the system, but built into the semantics and type checking rules of the language. This can be best illustrated by connecting some facts about the evaluation and type checking process: a constant without definition will not appear in the environment(see table 3.8); a constant without a binding value or definition in the environment will be evaluated to a primitive(in the same form of itself)(see the first rule of table 3.3); a primitive is only convertible to itself(see rule 3.16). With this observation, it is not hard to visualize how $(1+1)^{10}$ and $2^{(5+5)}$ could be checked convertible in our system by reducing both terms to 2^{10} according to the semantics of application and applying rule 3.17 about convertibility twice, provided that the set of natural numbers, addition operation, exponentiation operation have been properly defined and the definition of exponentiation is locked.

Our definition mechanism is an attempt to tackle the problem of convertibility checking by setting limit on computations based on constants. That is, a constant acts as a unit on which computations 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 only a minor variation 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: $2^m * 2^n = 2^{(m+n)}$; the other way is to reduce 2^{10} to $2 * 2^9$ by unlocking the definition of exponentiation only once. The former suggests establishing properties about data types and operations which is adopted by Haskell and proof-assistant systems like Agda; the latter indicates a dynamic changing of evaluation strategy. Although in this work, we didn't go any further towards either or the two direction, we do study and implement an operation called "head-reduction" that could limit computation on small steps each time. An introduction to this operation is given in section 3.6.2.

3.6.1 Find Minimum Set of Unlocked Constants

(further description goes here...)

3.6.2 Head Reduction

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

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

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

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

$$\frac{}{\Gamma \vdash R(e, e')} \left(\begin{array}{lcl} ns & = & \texttt{namesCont}(\Gamma) \\ ve & = & \texttt{headRedV}(\Gamma, e) \\ e' & = & \texttt{readBack}(ns, ve) \end{array} \right) \tag{3.25}$$

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

- `headRedV` :: `Cont` → `Exp` → `QE`: Evaluates an expression into a q-expression by a 'small' step under a given context. `Cont` is the type of context, `Exp` the type of expression and `QE` the type of q-expression.
- `readBack` :: `[String]` → `QE` → `Exp`: Transforms a q-expression back into an expression by eliminating all potential closures. A list of names taken from the underlying context is given as the first argument to avoid the name clashing problems.

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

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

The definitions of `headRedV, readBack` and `defVar` are given in table 3.9. Notice how the empty environment '()' is used in function `headRedV` to limit the steps of reductions performed.

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

Table 3.9: Functions: headRedV, readBack, defVar

As an example of head reduction, we present below in the listing 3.1 the result of applying head reduction to a constant named '`loop`' which we take from a file written in our language (see appendix A.3). The file represents a variation of Hurkens paradox [10] and is used as a test case for our program. There, evaluating the constant '`loop`' in an environment with all constants unlocked will cause the program to loop forever. However, we can use head reduction to show the results of the first few steps of evaluation.

Listing 3.1: Results of Head Reduction on the Constant Loop

```
step 1:
lem2 lem3
```

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

step 3:
lem1 C ([ x : U ] lem1 (delta x))
  ([ p : Pow U ] lem3 ([ z : U ] p (delta z)))

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

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

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

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

4

Extension

We describe in this chapter an extension to our language: a module system based on the idea of 'segment' from the work AUTOMATH. As introduced in section 2.3, the idea of 'segment' was developed as an abbreviation facility in AUTOMATH. For a complete description, we refer the readers to H. Balsters's work[11]. Here, we would avoid an introduction to the idea of 'segment' and how it used in AUTOMATH, but only illustrate its influence on our design of a module system by giving an example in the notation of our language as follows. Note that in the following discussion, we use the words 'segment' and 'module' interchangeably since they refer to the same structure in our language.

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 declaration. 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 a definition of segment by listing the relevant concepts and grammatical rules as follows.

Definition 4.0.1 (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 usage.
- 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.

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

- For the constants that are declared under the segment s, s are called their *declaring segment*. So for all declarations under a segment s, if we put all the declared constants into a set $\mathcal{A}$, and all the declared segments into a set $\mathcal{B}$, then the declaring segment of $\mathcal{A}$ and the parent of $\mathcal{B}$ is the same segment s.
- There is a *default segment* that is implicitly inhabited at the top-level and is denoted as *s-root*. It is the segment to which the first declaration of a program belongs.
- 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 s is the list of names that relate *s-root* to s under the relation $\rightarrow$. E.g., if a segment is declared with name "a" in the default segment, then its path is $[a]$[2]; if another segment is declared with name "b" in segment a, then its path is $[a, b]$. The path of *s-root* is the empty list.
- The *namespace* of a constant or segment is the string formed by joining the names in the path of its declaring/parent segment by the full stop character. E.g., for a constant declared in a segment whose path is $[a, b, c]$, its namespace is "a.b.c". The namespace of the constants or segments from *s-root* is the empty string.
- The *qualified name* of a constant is the string formed by joining the namespace of the constant and the name of the constant by a full stop character. E.g., the qualified name for a constant x in the default segment is ".x"; the qualified name for a constant x with namespace "a.b.c" is "a.b.c.x". We also call the usual, non-qualified name of a constant the *short name*.
- The *relative path* of a segment s to an ancestor a is the list of names that relate a to s under the relation $\rightarrow$. E.g., if b is a child of a and c is child of b, then the relative path of c to a is $[b, c]$.
- The *relative namespace* of a constant 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. E.g., 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 $[s2]$.
- In a segment, declarations of the form $x : A$ are called *parameters* of the segment.
- Declarations and definitions in a segment can be interleaved. That is, we do not require all declarations go before the definitions in a segment for the ease of use.
- A segment can be *instantiated* by giving a list of expressions. If the segment has no parameter, then the list must be empty; otherwise the list must be filled with the exact number of expressions that match the types of the parameters. The result is a new segment with the variables of the parameters of the old segment bound to the expressions provided as their definitions. E.g., for a

[2]Here we use the convention about list of names introduced in chapter 1

segment s with parameters $[x : A, y : B, z : C]$, it can be instantiated by a term of the form $s\,[e_1, e_2, e_3]$ where e_1, e_2, e_3 are expressions and have type 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 $s1 = s2\,[e_1, e_2, \ldots, e_n]$ where $s1$ is an instantiation of $s2$.
- Entities(constants and segments) in the current segment s or its descendants can be accessed by the dot operation (.), where on the left of the dot is the relative namespace of the entity to the segment s, and on the right is the name of the entity. If the relative namespace is the empty string, which means that the entity is declared under s, then it is referenced directly by its name. Both the relative namespace and the name are used in a non-quotation form, i.e., if the relative namespace is "a.b.c" and the name is "x", then variable x in segment c could be accessed from segment a by term $a.b.c.x$. This form of access is called *direct access*.
- The other form of access is *access by instantiation*, where the segment represented by the last name of the relative path is instantiated before the entity is accessed. It has the form $s_1.s_2\ \ldots\ s_n\,[e_1, e_2, \ldots, e_n]\,.\,x$ where $[s_1, \ldots, s_n]$ is the relative path of x to the ancestor in which the access happens and $[e_1, \ldots, e_n]$ are the expressions used to instantiate s_n.
- Expressions in a segment can only refer to entities that has been declared within the same segment or its descendants. This means that terms of the form $(s1\,[\ldots]\,.\,s2\,[\ldots]\ \ldots\ s_n\,[\ldots]\,.\,x)$ is not necessary since instantiation on the ancestors has no effect on the current segment. 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 current segment. Names of different segments may be the same without restriction.

Having introduced the concepts and rules about segment, we now describe in detail the syntax, semantics and type checking rules of our extended language.

4.1 Syntax of the Extended Language

We introduce below the abstract syntax of the extended language, for the concrete syntax see appendixA.2.

Expression in the extended language is defined as follows:

Definition 4.1.1 (Expression)

(i) Terms in the form U, x, y, z, $M\,N$, $[x : A]B$, $[x : A = B]M$ that are defined in 3.3.1 are still expressions in the extended language and have exactly the same meaning.

(ii) Given a non-empty list of names $[s_1, s_2, \ldots, s_n]$, an empty-possible list of ex-

pressions $[e_1, e_2, \ldots, e_m]$ and a constant x, a term of the form

$$s_1 \,.\, s_2 \,\ldots\, s_n \,[e_1, e_2, \ldots, e_m] \,.\, x$$

is an expression. When the list of expressions is not empty, it represents an *access by instantiation* to the constant x from the segment s_n whose relative path to the current segment is $[s_1, s_2, \ldots, s_n]$; otherwise it represents a *direct access* to the constant x from the segment s_n.

Declaration in the extended language is defined as follows:

Definition 4.1.2 (Declaration)

(i) Term in the form $x : A$, $x : A = B$ that are defined in 3.3.2 are still declarations in the extended language and have exactly 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 $[s_1, s_2, \ldots, s_n]$ and an empty-possible list of expressions $[e_1, e_2, \ldots, e_m]$, a term of the form

$$s = s_1 \,.\, s_2 \,\ldots\, s_n \,[e_1, e_2, \ldots, e_m]$$

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 $[s_1, s_2, \ldots, s_n]$.

A program of the extended language consists of a list of declarations as before. These top-level declarations belong to the default segment *s-root* just as we have introduced above. Within a segment, the name of a declaration still must not collide with any name of the existing declarations, but there is no constraint on the use of names between different segments. 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 4.1.

Exp	E	::=	$U \mid x \mid E_1\ E_2 \mid [x : E_1]E_2 \mid [x : E_1 = E_2]E_3 \mid$
			$s_1 \,.\, s_2 \,\ldots\, s_n \,[E_1, E_2, \ldots, E_m] \,.\, x$
Decl	D	::=	$x : E \mid x : E_1 = E_2 \mid s = \texttt{Seg}\ [D] \mid$
			$s = s_1 \,.\, s_2 \,\ldots\, s_n \,[E_1, E_2, \ldots, E_m]$
Prog	P	::=	$[D]$

Table 4.1: Syntax of the Extended Language

4.2 Operational Semantics

The fact that segments can be nested in the extended language suggests a tree like recursive structure of the evaluation environment.

Definition 4.2.1 (Environment)
An environment ρ in the semantics of the extended language has the following two properties:

(i) Property **p** that is the path of the segment represented by ρ.

(ii) Property **d** that is a dictionary relating each name x of the segment represented by ρ to one of the following entities:

- a quasi-expression v, meaning that the variable x is bound to a q-expression v in the environment;
- a definition $x : A = B$, meaning that the variable x is declared with type A and definition B in the environment;
- a sub-environment ρ', meaning that x is the name of the segment represented by ρ'.

We denote the path of ρ as ρ_p and the dictionary of ρ as ρ_d.

The definition of q-expression is the same as that of 3.4.2 and we still use the notation $[\![M]\!]\rho = q$ to express that expression M is evaluated to a q-expression q in the environment ρ.

$$
\begin{array}{lcl}
[\![U]\!]\rho & = & \mathcal{U} \\
[\![x]\!]\rho & = & \rho^*(x) \\
[\![s_1\,.\,s_2\,\ldots\,s_n\,[e_1, e_2, \ldots, e_m]\,.\,x]\!]\rho & = & \text{let } \rho_1 = \iota([s_1, \ldots, s_n], \rho, [e_1, \ldots, e_m]) \\
 & & \text{in } \rho_1^*(x) \\
[\![M_1\ M_2]\!]\rho & = & \alpha([\![M_1]\!]\rho, [\![M_2]\!]\rho) \\
[\![[x : A]\,B]\!]\rho & = & \langle [x : A]\,B, \rho \rangle \\
[\![[x : A = B]\,M]\!]\rho & = & [\![M]\!](\rho, x : A = B)
\end{array}
$$

Table 4.2: Semantics of the Extended Language

The evaluation rules for expressions in these forms $U, M\,N, [x : A]B, [x : A = B]M$ are the same as that in table 3.2. For expressions of the other two forms, the following functions deserve particular attention:

- $\rho^*(x)$: evaluate the constant x in environment ρ with the concept of namespace.
- $\iota(rp, \rho, es)$: get the environment corresponding to the segment whose relative path is rp and is instantiated by a list of expressions es in the environment ρ.

Function $\rho^*(x)$ relies further on three auxiliary functions, *lookup*, *is-qname* and *qname*.

- *lookup*(ρ, x): find the entity bound to the variable x(in short name) in the environment ρ. We use the notation
  - $\rho_d, x \mapsto \varnothing$ to denote that x is bound to nothing;
  - $\rho_d, x \mapsto v$ to denote that x is bound to a q-expression v;
  - $\rho_d, x \mapsto x : A = B$ to denote that x is bound to a definition $x : A = B$.

  Note that the name x used here must refer to a constant so that the case when x is mapped to a segment will not appear in this function.
- *is-qname*(x): a predicate testing whether x is a qualified name.

- *qname*(ρ_p, x): return the qualified name of variable x in environment ρ.

The definition of $\rho^*(x)$ is given as a pattern match function in table 4.3.

$\rho_d, x \mapsto \varnothing$	$\rightarrow$	if *is-qname*(x) then x else *qname*(x)
$\rho_d, x \mapsto v$	$\rightarrow$	v
$\rho_d, x \mapsto x : A = B$	$\rightarrow$	$[\![B]\!]\rho$

Table 4.3: Function - $\rho^*(x)$

Function ι relies on two auxiliary functions *findEnvSeg* and *bindEnvQ*.

- *findEnvSeg*(rp, ρ): find the environment ρ_1 that represents the segment whose relative path is rp to the segment represented by ρ. We also use this notation $\rightsquigarrow_{rp} \rho = \rho_1$ to express this function succinctly.
- *bindEnvQ*(ρ, qs): bind the names of the parameters of a segment represented by ρ to a list of q-expressions qs. The result is a new environment from ρ where the dictionary of ρ is updated by the list of (name, q-expression) pairs. Note that in the implementation, getting the parameters' names of a segment is handled by another procedure which we will not specify here. We also use this notation $\bigvee_{qs} \rho = \rho_1$ to express this function succinctly.

The definition of ι is given as a formula in table 4.4. Note that we overload the notation ($[\![]\!]$) below to express the evaluation on a list of expressions to get a list of q-expressions.

$$\iota(rp, \rho, es) \quad = \quad \text{let } \rho_1 = \rightsquigarrow_{rp} \rho,\ qs = [\![es]\!]\rho \text{ in } \bigvee_{qs} \rho$$

Table 4.4: Function - ι

4.3 Type Checking Algorithm

Like environment, the type checking context needs to be modified in a tree-like recursive structure.

Definition 4.3.1 (Type Checking Context)
A type checking context Γ in the semantics of the extended language has the following two properties:

(i) Property **p** that is the path of the segment represented by Γ.

(ii) Property **d** that is a dictionary relating each name x of the segment represented by Γ to one of the following entities:

- an expression A, meaning that the variable x has type A in this context;
- a definition $x : A = B$, meaning that the variable x is declared with type A and definition B in the context;
- a sub-environment Γ', meaning that x is the name of the segment represented by Γ'.

We denote the path of Γ as Γ_p and the dictionary of Γ as Γ_d.

Given a type checking context Γ, we can query the type of a variable by function $\Gamma(x)$, where x is the name or qualified name of a variable. Function $\Gamma(x)$ 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 in the outer scope of a segment cannot be referred inside the segment and (2) a *translation* process corresponds to the one mentioned in section 3.5 where proper declaration and usage of variables are checked. When x is a short name, then variable x is declared in Γ; otherwise if x is a qualified name, then variable x is declared in a descendant segment of Γ. In the latter case, we have a function *locate-seg* that given a context Γ and a qualified name x, finds the relative path of the declaring segment of x to Γ. Once we have the relative path, say rp, we can get the context of the descendant segment by a function called $findCtxSeg(\Gamma, rp)$ and denote it as $\Gamma_1 = \rightsquigarrow_{rp} \Gamma$ where Γ_1 is the context of the descendant segment. Lastly, similar with environment, we have a *lookup*(Γ, x) function that finds the entity bound to the variable x(in short name) in the type checking context Γ, we use the notation

- $\Gamma_d, x \mapsto A$ to denote that x has type A;
- $\Gamma_d, x \mapsto x : A = B$ to denote that x is declared of type A with definition B.

Also, for a variable with name x, the function *sname*(x) returns the short name of x: if x is not a qualified name, return x; otherwise return the last component of x split by the full stop character.

With these auxiliary functions introduced, the definition of $\Gamma(x)$ is given in a Haskell style pseudo-code as follows:

Listing 4.1: Function - $\Gamma(x)$

```
Γ(x) =
  let Γ' = if not is-qname(x) then Γ
           else let rp = locate-seg(Γ, x) in ⇝rp Γ
      x' = sname(x)
  in case Γ'_d, x' ↦ of
       A then A
       x : A = B then A
```

The lock strategies of the extended language have the same name and logic with that of the basic language, with three major differences:

- In transforming a type checking context to an environment, the tree structure of the segments must be retained.
- Names of variables used in the list as parameter for strategies *LockList* and *UnLockList* must be specified in qualified name.
- The lock/unlock operation can now be applied on segments, which means to lock/unlock all the constants in the segment. This is done by putting the "qualified name" of a segment to the list that is used as parameter for the lock strategies. A "qualified name" to a segment is formed by joining the names in its path by the full stop character.

5

Results

6

Conclusion

Bibliography

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

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

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

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

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

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

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

[8] 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.

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

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

[11] 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.

A

Appendix

A.1 Concrete Syntax for the Basic Language

```
position token Id (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;

comment "--";

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

A.2 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.3 Test Case

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

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

⊥ : * = [X : *] X ;

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

¬ : * -> * =
  [X : *] X -> ⊥ ;
```

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

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

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

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

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

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

C : U = tau Q ;

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

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

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

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

loop : ⊥
  = lem2 lem3 ;

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