Halftime Report

Qufei Wang

July 3, 2021

Contents

1 Report Structure

This report will be structured into three parts: First, a brief summary of the current progress of the project, including what has been done, what remains to be done and the deviation from the planning report; Second, the time plan for the remaining work; Third, a draft of the final report.

2 Project Progress

2.1 What Has Been Done

We have worked out a Haskell program that includes a type checker and a REPL interface which provides commands to experiment with the locking/unlocking mechanism. The first part of the project is to study how to present *definitions* in a dependent type theory, where constants could be locked/unlocked during evaluation. For this part, we have finished most of the work.

2.2 Remaining Work

For the first part of the project, what remains is to study more about the theoretical background of *definitions* in a proof system, particularly the theory about *closure* and *head-reduction*. This involves some literature study and an enhancement on the text in the draft of the final report that is related with the definition mechanism.

The second part of the project is to add a module mechanism with the notion of *segment*. The idea of 'segment' comes from the system AUTOMATH [2] which is conceived and developed by N.G. de Bruijn. We illustrate the idea with the following example.

Example 1. 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.

Here is an example

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

This 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.

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

Suppose we have another type $(A0 : *)$, then the expression $(s\ A0)\ .\ id$ has $A0 \to A0$ as its type and closure $([x : A]\ x)(A = A0)$ as its value.

For the module mechanism, what remains is to add the syntax to the language and the corresponding type checking rules to the type checker. A theoretical description of the implementation is also needed in the final report.

2.3 Deviation From The Time Plan

This halftime report comes more than 20 days later than what has been scheduled in the planning report. The reasons for the delay are more of psychological than technical, of which the details will not be brought up here. Nonetheless, I have gradually come to comprehend more and appreciate the ideas behind this project, thus getting more motivated. Besides, the draft of the final report included here also provides the context and framework for the remaining work. These two facts together make me feel confident that the project could still be delivered as scheduled. For this, I would like to thank Thierry for his patience and support.

2.4 Time Plan for The Remaining Work

The planning report states that from June 28 to July 25, the coding work of the AUTOMATH system should be done. Before that, there should be 20 days time spent on reading and understanding the literature about AUTOMATH. Now it is no longer realistic to spend ample amount of time on reading before the coding work. Instead, as the example 1 above suggests, a complete and deep understanding of AUTOMATH may not be a prerequisite to the implementation of the module mechanism.

So the time plan for the rest of the work remains the same: I will first try to have an implementation of the module system done before July 25. Then come back to the literature study and try to improve and complete the final report by the end of August.

3 Draft of the Final Report

3.1 Abstract

In this paper, we present a dependently typed language which is a simplified version of Mini-TT [1]. The differences between our language and Mini-TT are threefold: First, the syntax of our language is much simpler than that of Mini-TT. Particularly, we use the same syntax for both dependent product $(\Pi x : A.B(x))$ and λ abstraction $(\lambda x.M)$; Second, we build a locking/unlocking mechanism to the system and find a method to calculate the minimum set of constants to be unlocked, such that a new constant could be type check valid;

Third, as an extension to Mini-TT, we build a module system based on the notion of *segments* borrowed from the system AUTOMATH [2].

The disadvantage of having a substantial limited syntax lies in its reduced capability in expressiveness: there is no syntax to create data types in our language, which could be expressed as *Labeled Sum* in Mini-TT. However, starting out with a minimalized syntax allows us to focus more on the study of the definition mechanism, which is the main aim of this project. The outcome of the project is a REPL[2] implemented in Haskell, with commands to type check a source file and experiment with the locking/unlocking mechanism.

3.2 Terminology

In order to make clear of the potential ambiguity or unnecessary confusion over the words we choose to use in the following sections, we list below the terminology we use together with their meaning:

- **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 talk about the components of a specific definition, we also use the word 'definition' specifically to indicate the part of term B.

- **Constant:** A *constant* is the name or identifier used in a declaration, like the x in $x : A$, $x : A = B$.

- **Variable:** A synonym of the word *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.

- **Value:** When talk about the components of a definition $x : A = B$, we also use the word 'value' to indicate the part of term B, meaning the value of x. However, in the semantics of our language, a 'value' is more precisely used to refer to the result of expression evaluation.

3.3 Introduction

3.3.1 Some Background About Dependent Types

Dependent type theory has lent much of its power to the proof-assistant systems like Coq [3], Lean [4], and functional programming languages like Agda [5] and Idris [6], and contributed much to their success. Essentially, *dependent types*

[2]REPL stands for a read-evaluate-print-loop program

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 or java, there are array types which depend on the type of their elements, but no types that depend on values of some other type. 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 equipped with a dependently typed system are guaranteed with the highest level of correctness and precision, which makes them a natural option in building proof-assistant systems.

3.3.2 Issues with Dependently Typed Systems

The downside of introducing a dependent type system lies in its difficulties of implementation, one of which is checking the **convertibility** of terms. More precisely, in any typed system, it is crucial for the type checker to decide whether a type denoted by a term A is equal with another type denoted by a term B. In a simple typed system, this 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 [3]: there's no way for other terms in the language to be reduced to the term *int*. In a dependently typed system, however, the situation is more complex since a type may contain any value as its component, deciding the convertibility of types entails doing reduction on values, which requires much more computation.

One common approach to deciding the equality of terms in dependent type theory, whenever the property of confluence holds, is *normalization by evaluation* (NbE) [7], 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 using the definition of exponentiation.

[3] 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.

- As the number of definitions using previous definitions grows, the size of terms by expanding definitions 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 system.

3.3.3 Aim of the Project

The first aim of the project is to study how to present *definitions* properly in the dependent type theory. 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 proof assistant that is based on dependent type theory. We will analyze the example above later to give a support to our claim. Before that, we shall at fist make it clear for the reader this question: What exactly is the problem of definition and why is it important?

A *definition* in the context of dependent type theory is a term of the form $x : A = B$, meaning that x is a constant of type A, defined as B. The problem with definition is not about how a constant should be introduced, but how it should be **evaluated**. *Evaluation*, or *reduction*, in dependent type theory has its concept rooted in λ-*calculus* [8]. There, a term in the form $(\lambda x.M)\ N$ can be **evaluated** (or **reduced**) to the form $M[x := N]$, meaning that replacing the appearance of x in M with N everywhere [4]. In dependent type theory, however, different evaluation strategies can have huge difference when it comes to the efficiency of evaluation.

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

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

where Nat represents type of natural number and $*$ is the definition 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 exp multiple times, we keep the constant exp **locked** and only reduce both sides to the term $(exp\ 2\ 10)$. Then by showing that they can be reduced to a common term, we prove their equality with much less computation. Here, a **locked** constant has only its type information exposed, such that a type checker can still use it to do as much type checking work as possible, whereas its definition is erased so that we can not do any function application on it.

[4] There is a problem of the capture of free variables which we will not elaborate here. Curious and uninformed readers are encouraged to read detailed articles about λ-*calculus*.

The second aim of the project is to add a module system with the locking/unlocking capability. The module system is based on the idea 'segments' borrowed from the work of AUTOMATH [2]. (this paragraph could be expanded later when we have finished the module system)

3.3.4 Limitations

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

1. **Expressiveness:** We try to keep the syntax of our language as simple as possible in order to focus on the study of a proper definition mechanism, which inevitably affects the expressiveness of our language. As has been mentioned, there is no syntax for self-defined data types, nor for the pattern matches on case analysis functions. Besides, because we track the names of constants in a linear manner as an approach to the name collision problem, any constant declaration can not collide with that of top levels, there is no *variable shadowing* in our language.

2. **Scope:** For the study of definition, we do not try to establish a universal mechanism that is applicable in different systems. What we present here is only **one** alternative for doing type checking in the presence of definitions in a dependent type theory. Thus, the result of our work applies only in a very limited scope.

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

3.4 Theory

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

3.4.1 Subtleties in a Dependent Type Theory

We present the subtleties by giving the following examples.

Example 2 (Definitions in dependent type theory cannot be reduced to λ-*calculus*). Suppose we have

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

then

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

is not well typed because the type of y is $(P\ x)$ not $(P\ a)$. However, if we modify it to

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

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

Example 3 (Names should be handled carefully). Suppose we have

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

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

Example 4 (Problem with capture of variables). 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.

$$(\lambda u\ .\ u\ (u\ b))(\lambda z\ y\ x\ .\ a\ (z\ x)\ y) \tag{1}$$

If we do the reduction on (1) naively, we get

$$\begin{aligned}
(\lambda u\ .\ u\ (u\ b))(\lambda z\ y\ x\ .\ a\ (z\ x)\ y) &\implies\\
(\lambda z\ y\ x\ .\ a\ (z\ x)\ y)((\lambda z\ y\ x\ .\ a\ (z\ x)\ y)\ b) &\implies\\
(\lambda z\ y\ x\ .\ a\ (z\ x)\ y)(\lambda y\ x\ .\ a\ (b\ x)\ y) &\implies\\
\lambda y\ x\ .\ a\ ((\lambda y\ x\ .\ a\ (b\ x)\ y)\ x)\ y & \qquad (2)
\end{aligned}$$

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

(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) 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.

3.4.2 Definitions in a Dependent Type Theory

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

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

Principle 2. Forbid the shadowing of variable names.

Principle 3. Rename variable whenever necessary.

Principle 1 relates to example 2. As has been suggested in the example, the definition of a constant can be important to ensure the type safety of a term. In other cases, however, the definition is not needed, like in this term $\lambda(f : A \to B)(a : A)\ .\ f\ a$: f could be any function from A to B, a could be any value of A, regardless of their specific values, we know for sure that term $f\ a$ has type B. These facts indicate that type and definition take unequal roles in dependent type theory: one can declare a constant without a definition, but cannot declare a constant without a type.

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

Principle 2 comes as a simple strategy to avoid the pitfall revealed by example 3. During the type checking process, each declaration, including the declarations from λ-abstraction, is checked with the top level context (actually, we have only one level context) to ensure no naming clash occurs. Using *De Bruijn index* is another, maybe better, from the point of view of the user, way to avoid the name clashing issue. However, having to maintain the relationship between names and

indices may unnecessarily complicate our implementation and obscure the main aim of the project.

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

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

Principle 4. Deferred evaluation.

In order to reduce unnecessary reduction steps during the type checking process by exploiting a locking mechanism on variables, we need to defer the computation on function application as late as possible. We do this by

1. Using *closure* to carry functions around. A closure is a function (or λ abstraction) extended with an environment.

2. Applying β-reduction on multi-variable functions in an incremental manner.

3. Only unlocking a constant when the type checking procedure can not proceed. We use this technique to find the minimum set of unlocked constants to type check that a declaration is valid.

With all the 4 principles introduced, we are ready to describe in detail the syntax and semantics of our language, and the operations we build upon it.

3.4.3 Syntax of the Language

A program of our language consists of a list of declarations. A declaration has either the form $x : A$ or $x : A = B$, where A, B are expressions. A summary of the syntax can be found in table 1.

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

Table 1: Language Syntax

The meaning of each expression constructor is explained as follows:

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

- **Dependent Product:** $\Pi\, x : A\, .\, M$ - the type of functions which take an arbitrary object x of type A into an object of type M (M may dependent

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

Table 2: Expressions

on x).

- **λ-abstraction:** $\lambda\,(x : A)\ .\ M$ - a function that takes a variable x of type A into an expression M.

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

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

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

The syntax of our language is a substantial subset of that of Mini-TT. Moreover, we use the same syntax for both dependent product and λ abstraction as an effort to maintain simplicity. This practice causes ambiguity only when an expression in the form $[x : A]M$ is viewed in isolation: it can be seen both as a dependent type and a function abstraction. In the former case, M represents a type, whereas in the latter case a value. Our type checking rules ensure that it could be used in a consistent way.

3.4.4 Operational Semantics

Expressions are evaluated to *values* under a given environment. The definitions for *values* are given in table 3.

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

Table 3: Values of the Language

The meaning of each value constructor is explained as follows:

Note that in our Haskell implementation, we use the same syntax for both expressions and values, since our syntax is simple.

U	:	The value of *Universe*, U
x	:	*Neutral value*, the value of a undefined variable
$u\ v$	:	Value of an application when the first value u cannot be reduced to a closure
$\langle [x : A]M, \rho \rangle$	:	Closure, a function extended with an environment

Table 4: Definition of Value

An environment is defined as

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

meaning that an environment is either empty, or another environment extended with a variable paired with its value, or extended with a definition.

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

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

Table 5: Semantics of Language

The operation *appVal* is defined as:

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

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

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

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

3.4.5 Type Checking Rules

3.4.5.1 Type Checking Context

The type checking procedure is performed under a context Γ,

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

meaning that a type checking context is either empty, or another context extended with a variable paired with its type, or extended with a definition.

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

$$\begin{aligned}
()(x) &= \text{error} \\
(\Gamma, x : A)(x) &= A \\
(\Gamma, y : A)(x) &= \Gamma(x)(y \neq x) \\
(\Gamma, x : A = B)(x) &= A \\
(\Gamma, y : A = B)(x) &= \Gamma(x)(y \neq x)
\end{aligned}$$

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

In our implementation, when parsing the source file into the abstract syntax of our language, we make sure that each variable must be declared with a type and the name of the variable has never been used. By doing so, we ensure that the error condition in the lookup operation will never occur during the type checking process and each variable's name is unique.

We also defined a function `freshVar` that given the name a variable and a type checking context, return a new name.

$$\text{freshVar} :: \text{String} \to \Gamma \to \text{String}$$

This function is used whenever a variable renaming is needed.

The locking/unlocking mechanism in our system is implemented via a concept called *lock strategy* plus a function called `getEnv`. For a lock strategy s, we have

$$\text{getEnv} :: s \to \Gamma \to \rho$$

The idea is that when we lock a constant, we need to remove its definition from the environment ρ, such that when evaluated, this constant becomes a neutral value, cutting off all the possibility for further evaluation; When we unlock the constant later, we need to restore its definition to ρ. During the type checking process, the context Γ is always extended with all the definitions declared so far. By the function *getEnv* and a lock strategy s that represents our intention about the locking/unlocking of each variable, we can conveniently get the environment ρ that effectuates our locking strategy.

In our current implementation, we have 4 lock strategies: *LockAll, LockNone, LockList vs, UnLockList vs*, where vs is a list of variables. We give their definitions in a Haskell like pseudo-code in table 6.

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

Table 6: Lock Strategies

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

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

Table 7 lists out the judgments used during the type checking process. There, Γ is the type checking context and s is the lock strategy. Note that the name collision check is performed before the type checking process, so we do not need to check the name uniqueness of each constant in the declarations anymore.

3.4.5.2 checkDecl

$$\frac{\Gamma, s \vdash A \Leftarrow U}{\Gamma, s \vdash x : A \Rightarrow \Gamma_1} \tag{3}$$

$$\frac{\Gamma, s \vdash A \Leftarrow U \quad \Gamma, s \vdash B \Leftarrow t}{\Gamma, s \vdash x : A = B \Rightarrow \Gamma_1} \left(\begin{array}{lcl} t & = & [\![A]\!]\rho \\ \rho & = & \mathrm{getEnv}(s, \Gamma) \end{array} \right) \tag{4}$$

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

Table 7: Type Checking Judgments

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

3.4.5.3 checkInferT

$$\frac{}{\Gamma, s \vdash U \Rightarrow U} \tag{5}$$

$$\frac{}{\Gamma, s \vdash x \Rightarrow t} \begin{pmatrix} t & = & [\![A]\!]\rho \\ A & = & \Gamma(x) \\ \rho & = & \text{getEnv}(s, \Gamma) \end{pmatrix} \tag{6}$$

U has itself as its type; A variable x is well typed and its type is inferred to be the value evaluated from its bound type in Γ.

$$\frac{\Gamma, s \vdash M \Rightarrow \langle [x : A]B, \rho \rangle \qquad \Gamma, s \vdash N \Leftarrow va}{\Gamma, s \vdash M\ N \Rightarrow v^*} \begin{pmatrix} v^* & = & [\![B]\!]\rho_2 \\ va & = & [\![A]\!]\rho \\ vn & = & [\![N]\!]\rho_1 \\ \rho_2 & = & (\rho, x = vn) \\ \rho_1 & = & \text{getEnv}(s, \Gamma) \end{pmatrix} \tag{7}$$

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

1. Check M is a function, namely, it has type in form $\langle [x : A]B, \rho \rangle$
2. Check N has the right type to be applied to M
3. Return the value of B evaluated in ρ extended by binding x to the value of N

$$\frac{\Gamma, s \vdash x : A = B \Rightarrow \Gamma_1 \qquad \Gamma_1, s \vdash M \Rightarrow t}{\Gamma, s \vdash [x : A = B]\, M \Rightarrow t} \tag{8}$$

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

3.4.5.4 checkWithT

$$\frac{}{\Gamma, s \vdash U \Leftarrow U} \tag{9}$$

$$\frac{\Gamma, s \vdash v \equiv vt \Rightarrow v^*}{\Gamma, s \vdash x \Leftarrow v} \left(\begin{array}{lcl} vt & = & [\![A]\!]\rho \\ A & = & \Gamma(x) \\ \rho & = & \mathrm{getEnv}(s, \Gamma) \end{array} \right) \tag{10}$$

As we have already known, U has U as its type; To check a variable x has type v, we first get the value vt of the type bound to x from the context Γ, then we check that vt and v are convertible.

$$\frac{\Gamma, s \vdash M\ N \Rightarrow v' \qquad \Gamma, s \vdash v' \equiv v \Rightarrow v^*}{\Gamma, s \vdash M\ N \Leftarrow v} \tag{11}$$

$$\frac{\Gamma, s \vdash A \Leftarrow U \qquad \Gamma_1, s \vdash B \Leftarrow U}{\Gamma, s \vdash [x : A]\, B \Leftarrow U} (\Gamma_1 = (\Gamma, x : A)) \tag{12}$$

To check an application $M\ N$ has type v, we first infer its type v', then we check that v and v' are convertible; To check an abstraction $[x : A]\, B$ has type U, we first check that A has type U, then we check that B also has type U in an extended context.

$$\frac{\Gamma, s \vdash A \Leftarrow U \qquad \Gamma, s \vdash va \equiv va' \Rightarrow t \qquad \Gamma_1, s \vdash B \Leftarrow vb'}{\Gamma, s \vdash [x : A]\, B \Leftarrow \langle [x' : A']\, B', \rho \rangle} \left(\begin{array}{lcl} va & = & [\![A]\!]\rho_1 \\ va' & = & [\![A']\!]\rho \\ vb' & = & [\![B']\!]\rho_2 \\ \Gamma_1 & = & (\Gamma, x : A) \\ \rho_1 & = & \mathrm{getEnv}(s, \Gamma) \\ \rho_2 & = & (\rho, x' = x) \end{array} \right) \tag{13}$$

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

1. Check A has type U
2. Evaluate the value of A in the environment extracted from the current context, denote it as va
3. Evaluate the value of A' in the environment from the closure, denote it as va'
4. Check that va and va' are convertible
5. Extend ρ to ρ_2, with x' bound to x
6. Evaluate B' in ρ_2, denote the value as vb'
7. Extend Γ to Γ_1, with x having type A
8. Check B has type vb' in the new context Γ_1

This is the rule used to check an abstraction has another abstraction as its type.

$$\frac{\Gamma, s \vdash x : A = B \Rightarrow \Gamma_1 \qquad \Gamma_1, s \vdash M \Leftarrow t}{\Gamma, s \vdash [x : A = B]\, M \Leftarrow t} \tag{14}$$

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

3.4.5.5 checkEqualInferT

$$\frac{}{\Gamma, s \vdash U \equiv U \Rightarrow U} \tag{15}$$

$$\frac{x =:= y}{\Gamma, s \vdash x \equiv y \Rightarrow v} \left(\begin{array}{lcl} v & = & [\![A]\!]\rho,\ A = \Gamma(x) \\ \rho & = & \mathrm{getEnv}(s, \Gamma) \end{array} \right) \tag{16}$$

The first rule states that U is equal to itself and has type U; For two variables to be equal, they must have the same name and their type is inferred to be the value of the type bound to the name in Γ.

$$\frac{\Gamma, s \vdash M_1 \equiv M_2 \Rightarrow \langle [x : A]\, B, \rho \rangle \qquad \Gamma, s \vdash N_1 \equiv N_2 \Leftarrow va}{\Gamma, s \vdash (M_1\ N_1) \equiv (M_2\ N_2) \Rightarrow v} \left(\begin{array}{lcl} va & = & [\![A]\!]\rho \\ v & = & [\![B]\!]\rho_2 \\ vn & = & [\![N_1]\!]\rho_1 \\ \rho_2 & = & (\rho, x = vn) \\ \rho_1 & = & \mathrm{getEnv}(s, \Gamma) \end{array} \right) \tag{17}$$

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

1. Check M_1 and M_2 are convertible and has type in the form of a closure $\langle [x : A]B, \rho \rangle$
2. Get the value of A evaluated in the environment ρ, denoted as va
3. Check N_1 and N_2 are convertible given va as their type.
4. Get the value of N_1 evaluated in the environment extracted from the current type checking context, denoted as vn
5. Extend ρ with variable x bound to vn to ρ_2
6. Return the value of B evaluated in ρ_2 as the inferred type

$$\frac{\Gamma, s \vdash \langle [x : A]\, B, \rho \rangle \equiv \langle [y : A']\, B', \rho' \rangle \Leftarrow U}{\Gamma, s \vdash \langle [x : A]\, B, \rho \rangle \equiv \langle [y : A']\, B', \rho' \rangle \Rightarrow U} \tag{18}$$

We check the convertibility of two closures by checking that they are convertible given type U. This inference rule is only used when two values representing **types** are checked for convertibility[5]. In this case, the abstractions from the closures are always seen as elements of the type U, not as elements of types in the form of some other closures. This reflects a 'two-tier' type structure of our system: Only U and elements of U (in the form of an abstraction, as indicated by rule 12) are eligible to be used as types.

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

3.4.5.6 CheckEqualWithT

$$\frac{\Gamma_1, s \vdash m \equiv n \Leftarrow vb}{\Gamma, s \vdash v1 \equiv v2 \Leftarrow \langle [x : A]\, B, \rho \rangle} \left(\begin{array}{lcl} y & = & \text{freshVar}(x, \Gamma) \\ \rho_1 & = & (\rho, x = y) \\ vb & = & [\![B]\!]\rho_1 \\ \rho_0 & = & \text{getEnv}(s, \Gamma) \\ m & = & [\![v1\ y]\!]\rho_0 \\ n & = & [\![v2\ y]\!]\rho_0 \\ va & = & [\![A]\!]\rho \\ \Gamma_1 & = & (\Gamma, y : va) \end{array} \right) \tag{19}$$

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

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

This rule accommodates for η-conversion, where $\lambda x.f\, x$ and f can be checked convertible. This is the reason why we apply $v1$ and $v2$ with the new variable, and check the convertibility of the result. We generate a new variable and do variable renaming as a respect to principle 3 in section 3.4.2. Note that we do not replace each x in B to y manually, but add the binding (x, y) to the environment in the closure and rely on the evaluation operation to achieve the desired effect.

$$\frac{\Gamma, s \vdash va_1 \equiv va_2 \Leftarrow U \qquad \Gamma_1, s \vdash vb_1 \equiv vb_2 \Leftarrow U}{\Gamma, s \vdash \langle [x_1 : A_1]\, B_1, \rho_1 \rangle \equiv \langle [x_2 : A_2]\, B_2, \rho_2 \rangle \Leftarrow U} \left(\begin{array}{lcl} va_1 & = & [\![A_1]\!]\rho_1 \\ va_2 & = & [\![A_2]\!]\rho_2 \\ y & = & \text{freshVar}(x_1, \Gamma) \\ \rho_{21} & = & (\rho_1, x_1 = y) \\ \rho_{22} & = & (\rho_2, x_2 = y) \\ vb_1 & = & [\![B_1]\!]\rho_{21} \\ vb_2 & = & [\![B_2]\!]\rho_{22} \\ \Gamma_1 & = & (\Gamma, y : va_1) \end{array} \right) \tag{20}$$

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

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

$$\frac{\Gamma, s \vdash v1 \equiv v2 \Rightarrow t' \qquad \Gamma, s \vdash t \equiv t' \Rightarrow t^*}{\Gamma, s \vdash v1 \equiv v2 \Leftarrow t} \tag{21}$$

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

3.4.6 Locking Mechanism

As has been introduced, a locking mechanism in our system is realized by setting up a *lock strategy*, and use it to extract an environment from the underlying type

checking context. The *environment* is the place where a variable is bound to its definition and the context in which the evaluation of an expression takes place. A variable without a value bound evaluates to itself, a *neutral value* about which we know nothing, which cannot be reduced to any other form, either by itself or applied with another value. We adjust the lock strategy so that the definition of a constant could be erased or restored from the environment. In this way, we effectively lock/unlock a variable.

This is a locking mechanism applied to definitions where a constant acts as a *locking unit*. The lock status of variables are independent of each other, meaning that lock/unlock a constant does not entail other constants in its definition being locked/unlocked. An alternative is to apply locking on expressions, where we define a metric of computation such that during evaluation, only certain 'steps' of reductions are performed. We did not build this alternative in our system but will elaborate the idea more in section 3.4.8 when we talk about *head reduction*.

One application of our locking mechanism is to find the minimum set of constants unfolded in a well typed context, such that another constant could be checked valid. The method is straightforward and largely based on the type checking inference rules. We only sketch the idea here, readers who are curious can refer to appendix A.1 for a complete view of the Haskell source code.

Say we have a text file which contains a valid program (a list of declarations) of our language. We want to find the minimum set of constants unfolded for another constant, say x, to be valid. We do as follows:

1. Load and type check the program with no variables locked. This will give us the complete type checking context Γ^* .

2. Find the definition of x in $\Gamma*$, denote it as D.

3. Denote the context consisting of all the declarations which appear before D as Γ.

4. Create a list `vs` that is used to keep track of all the variables unfolded and change the lock strategy to `UnLockList vs`. The initial value of `vs` is `[]` and the initial lock strategy is actually `LockAll`.

5. Type check the declaration D. Whenever the type checking process halts because of an unmatched pattern (e.g. checking $[x : A]\, B$ with a neutral value x) or fails because of a violation to one of the inference rules, find the first variable locked that causes the exception, add it to `vs` and update the lock strategy.

6. Repeat 5 until the type checking process succeed.

7. Return the intersection of the variables in `vs` and the variables in Γ as the

result.

Note that because the program is well typed, this algorithm is guaranteed to terminate. We take the intersection in the final step because the renaming operation introduces new variables that are not originally declared.

3.4.7 Read Back to Normal Form

3.4.8 Head Reduction

3.5 Methods

3.6 Results

3.7 Conclusion

References

[1] 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.

[2] 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.

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

[4] 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.

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

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

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

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

A Appendix

A.1 Haskell Source Code