Halftime Report

Qufei Wang

June 28, 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, any deviation from the planning report and what remains to be done; Second, the major part of the report, is a draft of the final report. And the third, adjustment on the time planning for the remaining of the project.

2 Project Progress

2.1 What Has Been Done

A Haskell implementation of our target language, a command line REPL (read-evaluate-print-loop) tool where a source file of the target language could be loaded and type checked. For the ease of use, we also provide various other commands including those used to experiment with the locking/unlocking mechanism. The first part of the thesis project is to study an extension of lambda calculus with definitions, where constants could be locked/unlocked in the process of evaluation (or reduction). For this part, we could say that we have completed almost all of its work.

2.2 Deviation From the Time Plan

This halftime report comes about half month later than what has been scheduled in the planning report. The reasons for the delay are more of psychological than technical. One reason relevant with the project itself is that, I was too obsessed with the idea of getting a bigger picture of how the project might fit into the spectrum of the knowledge of functional programming or logic, rather than getting down to the practical work. Nonetheless, I'm still confident that the remaining work could be finished within the planned time frame, with the shift of my mindset. For this, I would like to thank Thierry for his patience and support.

2.3 Remaining Work

The second part of this thesis project is to add a module mechanism with the notion of *segment*. Such a mechanism could be seen as an extension to our target language and the idea of 'segment' comes from the system AUTOMATH [2], which is conceived and developed by N.G. de Bruijn. The whole picture of AUTOMATH may not be easy to grasp in a short time, but as the example given below shows, we may still be to borrow the idea of 'segment' and incorporate it into our language without a fully understanding of the system.

Example 2.1 (The notion of 'segment'). The idea of *segment* is to add a new form of declaration $x = ds$ **Seg**, where x is the name of the 'segment' or 'modules with parameters' and ds a list of declaration ('Seg' is a keyword reserved by the language). Here is an example:

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

This is a module which contains one declaration and one definition. The declaration $A : *$ ($*$ here is the type of all small types, for a detailed description of the syntax, please refer to TODO-REF) acts as a parameter whereas the definition with name id represents the identity function.

With this segment declaration, if we can form another type $A0 : *$, we can then write the expression $(s\ \ A0)\ .\ id$, which has type $A0 \to A0$ and value $([x : A]\ x)(A = A0)$ (the value here is a closure).

With this example taken in mind, we see what still lacks are extensions to the syntax and type checking rules for our language to accommodate this *module* concept.

3 Draft of the Final Report

3.1 Abstract

In this paper, we present a dependently typed language which could be seen as a simplified version of Mini-TT [1]. The main difference 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 add a locking/unlocking mechanism into our type checking algorithm, from which we find a way to calculate the minimum set of constants that need to be unlocked, such that a constant declaration could be type checked; 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 limited subset of syntax is its reduced capability in expressiveness (e.g., one can not build data types in our language, which could be expressed as *Labeled Sum* in Mini-TT). However, starting out with a simple syntax allows us to focus on the study of a definition mechanism in dependent type theory, which is the main aim of this thesis project.

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 here the key terminology and explain 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 use *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.

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

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

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* 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 type system built into a programming language is 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 type system where no type polymorphism or dependent type is used, 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, since types in Java are not computable [1], there's no way for other terms

[1] Technically speaking, the type of an object in Java can be retrieved by means of the *reflections* and presented in the form of another object, thus subject to computation. But it

in the language to be reduced to the term *int*. In a dependent type system, however, the situation is more complex since a type may contain any value as its component, deciding the equality 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 the canonical representations 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.

- 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 and explore a *definition* mechanism, where definitions of constants could be expanded as few times as possible during the type checking process. What this means is that, we want to build a locking/unlocking operation on the constants, such that we can indicate certain constants to be locked (or unlocked) 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. Without providing a rigorous proof, we will take the example above later to illustrate the idea behind the claim. Before that, we should at fist make it clearer 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] (with some variance we will come about later in section TODO). 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

is not computable on the syntactic level, like being passed as arguments to functions.

x in M with N everywhere [2]. 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.

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.

[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 detailed articles about λ-*calculus*.

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 3.1 (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 3.2 (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 3.3 (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

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

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 a 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 of a constant x separately.

Principle 2. Forbid the shadowing of variable names.

Principle 3. Rename variable during convertibility checking or ‘read back‘ a value to its normal form.

3.4.3 Syntax of the Language

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	$::=$	$x : A \mid x : A = B$
syntactic sugar	$A \to B$	$::=$	$[\_ : A]B$

Table 1: Language Syntax

Expressions are defined as follows

- U : the type of a universe of small types.
- x, y, z : variables(constants) with names, as opposed to the variables denoted by *De Bruijn* indices.
- MN : function application.
- $[D]M$: abstraction.

A declaration has either of the two forms

- $x : A$: variable x has type A.
- $x : A = B$: variable x has type A and is defined as B.

An abstraction of the form $[x : A]\,B$ can be used to represent

- $\Pi x : A\,.\,B$: dependent product, meaning that for any element $x \in A$, there's a type B which may depend on x.
- $\lambda\,(x : A) \to B$: λ abstraction.
- A non-dependent function $A \to B$ is desugared as $[\_ : A]\,B$, with the dummy variable ‘_’ meaning that there's no variable introduced.

An abstraction of 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* $x : A = B$.

3.4.4 Operational Semantics

Expressions can be evaluated to *values*, which are defined in table 2. Note that in the implementation, we did not differentiate in syntax between *expressions* and *values*, since the syntax is simple and we can use the same syntax for both.

$$\text{values} \quad u, v \quad ::= \quad U \mid x \mid u\, v \mid \langle e, \rho \rangle$$

Table 2: Values of the Language

Another two important concepts that are used widely in expression evaluation and type checking are environment (*Env*, ρ) and context (*Cont*, Γ). An environment relates variables to their values and a context relates variables to their types. An environment is defined as

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

and a context is defined as

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

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

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

The operation *appVal* is defined as follows:

$$\text{appVal} \quad \langle [x : A]\, B, \rho \rangle \quad v \quad = \quad [\![B]\!](\rho, x = v)$$

otherwise

$$\text{appVal} \quad v1 \quad v2 \quad = \quad v1\, v2$$

We also define lookup operations on environment and context

- $\rho(x)$: find the value of variable x in the environment ρ.
- $\Gamma(x)$: find the type of variable x in the context Γ.

with

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

and

$$\begin{array}{rcl} ()(x) & = & \text{error: variable not declared} \\ (\Gamma, x : v)(x) & = & v \\ (\Gamma, y = v)(x) & = & \Gamma(x)(y \neq x) \\ (\Gamma, x : A = \_)(x) & = & [\![A]\!](\text{envCont}\,\Gamma) \\ (\Gamma, y : A = B)(x) & = & \Gamma(x)(y \neq x) \end{array}$$

Note that the type check algorithm ensures that each variable is bound with a type, such that the error condition never happens.

We can get an environment out of a context by using the function *envCont*

$$\begin{array}{rrcl} \text{envCont} & () & = & () \\ \text{envCont} & (\Gamma, x : v) & = & \text{envCont}\,\Gamma \\ \text{envCont} & (\Gamma, x : A = B) & = & (\text{envCont}\,\Gamma,\ x : A = B) \end{array}$$

3.4.5 Typing Rules

The type checking algorithm is implemented as a state monad in Haskell, where the state is a context(Γ) starting from an empty context and getting updated by checking each declaration from the source file.

There are four forms of judgments:

checkD	$\Gamma \vdash D \Rightarrow \Gamma'$	D is a correct declaration and extends Γ to Γ'
checkT	$\Gamma \vdash M \Leftarrow t$	M is a correct expression given type t
checkI	$\Gamma \vdash M \Rightarrow t$	M is a correct expression and its type is inferred to be t
checkC	$\Gamma \vdash u,\ v$	the two terms u, v are convertible

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.