Halftime Report

Qufei Wang

June 2, 2021

1 Introduction

1.1 Background

Dependent type theory is widely used in proof-assistant systems (e.g. Coq [1], Agda [2], Lean [3], Idris [4], etc.) and contributed much to their success. In any typed system, deciding whether one type is equal to another is important. 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, whereas in dependently typed systems the problem becomes more complicate, since a type may contain any value as its component, making it unavoidable to do computations on types.

One common approach to deciding equality of terms in dependent type theory is *normalization by evaluation* (NbE) [5], 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.
- 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. We hope to find a common reduct before going into expensive reductions, and in the light of this, we find that a proper study of a *locking/unlocking* mechanism on the *definitions* of a dependent type system may provide us hints to the solution of the problem. Here, a *definition* has

the conceptual form $c : A = M$, which means that a *constant* c has type A and is defined as M, A and M are just common terms of the target language. A constant is *locked* when we explicitly hide its definition from the context, making only the type information accessible. It becomes *unlocked* when we bring its definition back into the context again.

1.2 Aim

The aim of this project is to find a mechanism where *definitions* in a dependently typed system could be handled more efficiently. What this means is that, we aim to develop a *locking/unlocking* mechanism into a simple dependent type language, which allows us to do computations on terms with the ability to lock/unlock some of the *constants*, with the hope that a proper design of the mechanism could make the type checking process (in particular, deciding the convertibility of two terms) more efficient. The target language is a simplified version of Mini-TT [6], whose syntax and semantics will be described in the following sections. On top of that, we may wish to extend the language with one or two language features, such as a module system or implicit argument handling, to make it more practical.

1.3 Limitations

We will not try to establish a universal mechanism that is applicable in different dependent type systems. Instead, we will only focus on the system introduced by the simple language with its specific syntax and semantics. Also, there's no guarantee that a rigorously describable mechanism could be found, and what we finally establish may only work under certain preconditions. However, just by studying the problem and giving suggestions to the possible directions to a better solution may also deemed as a valuable work.

2 Current Progress

Currently there has been a Haskell implementation of the language, with a basic locking/unlocking mechanism. We give a detailed description of the language here, with regard to its syntax, semantics, evaluation operations and type checking algorithms. We also present the locking/unlocking mechanism in a brief manner.

2.1 Syntax of the Language

A summary of the syntax can be found in table 1.

Expressions are defined as follows

- U : the type of a universe of small types.

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

- 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* $x : A = B$.

2.2 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.

values	u, v	$::=$	$U \mid x \mid u\,v \mid \langle e, \rho \rangle$

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

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

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*

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

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

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

[3] 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.

[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] 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.