Planning Report

Qufei Wang

April 22, 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]. On top of that, we wish to extend the language with a module system, which is based on the concept of 'segment' borrowed form AutoMath [7]. The outcome of the work should be an implementation of type checker in Haskell, that illustrates the concepts above.

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.

1.4 Specification of Issue under Investigation

These are the questions that we'd like to address with regard to the aim we claimed.

- How to design the type checking algorithm in the presence of locked variables ?
- How to find the minimum set of constants to be unlocked in a non-exhaustive way, for a term to be checked as valid ?
- Is there a way to use the knowledge about the minimum set of constants needed for each constant in the context, to make type checking a new term more efficient ?

2 Methodology

In the work already done so far, we've implemented an evaluation operation on terms with the ability to hide some constants. In an unpublished draft (which we will refer to as "transparency" and will be submitted together with this report as a reference), the author proposed a mechanism with transparency/opacity configuration which may provide us some inspiration on a further development of our locking/unlocking mechanism.

Also, since the problem of *definition* has close relation with *evaluation*, and the evaluation mechanism in a dependent type language has its root in typed λ-calculus, a familiarity with the literatures in this field may also be helpful. For example, consider the simple problem of checking the convertibility of two terms $id\ A\ M$ and M, where

$$id : [A : U]\, A \to A = [A : U]\, [x : A]\, x$$
$$M : A = \alpha$$

Suppose M is type correct and α is a constant whose value requires enormous computation. Without going into the details of α, we can instantly know that these two terms are equal. However, an evaluation mechanism may go as far as computing the value of M, which makes the problem unnecessarily complicate (or even undecidable!).

Laziness, as a strategy of evaluation used in Haskell, can help to avoid the over-computed situation. Another way to think about this is if we allow the user to specify certain algebraic properties of the constants (e.g. $\forall A \in U, x \in A \Rightarrow id\ A\ x = x$) and suggest the properties to the type checker, then the problem could be solved more efficiently (just like in Agda, one can prove properties of data types and use the properties as functions for further proof). The ideal solution would be that the type checker can deduce the properties itself and choose the properties intelligently from the context for the further reasoning.

3 Schedule

The schedule of the remaining work is shown in figure 1. The details of each task is shown in table 1. The schedule covers time from April 19, 2021 to September 12, 2021, in a total of 20 weeks (when taking into consideration the extension).

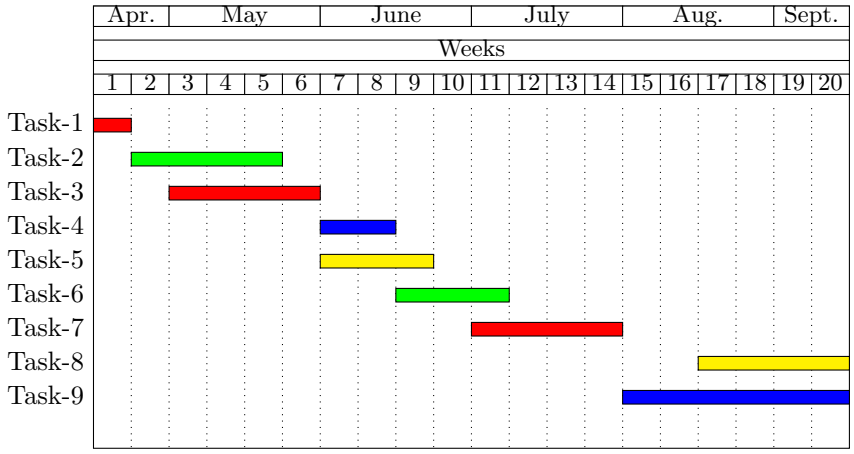

Figure 1: Task Schedule

	Tasks	Duration(weeks)	Start Date	End Date
1	**Coding** η-conversion	1	Apr. 19	Apr. 25
2	**Literature Study** evaluation mechanism in typed λ-calculus	4	Apr. 26	May 23
3	**Coding** development on the *definition* mechanism	4	May 3	May 30
4	**Writing** Half-time report	2	May 31	June 13
5	**Presentation** attend opposition presentation	3	May 31	June 20
6	**Literature Study** AUTOMATH system	3	June 14	July 4
7	**Coding** a module mechanism based on the notion 'segment' from AUTOMATH	4	June 28	July 25
8	**Presentation** final presentation	4	Aug. 16	Sep. 12
9	**Writing** Final report	6	July 26	Sep. 12

Table 1: Task Detail

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.

[7] N. G. de Bruijn, "Generalizing automath by means of a lambda-typed lambda calculus," *Mathematicallogic and theoretical computer science*, pp. 71–92, 1987.