Master thesis project proposal

A Haskell Implementation of Mini-TT: A Dependent Type Functional Language

Qufei Wang
qufei@student.chalmers.se

Suggested Supervisor at CSE: Thierry Coquand
Thierry.Coquand@cse.gu.se

Supervisor at Company:

Relevant completed courses student:

DAT060/DIT201, Logic in Computer Science
TDA342/DIT260, Advanced Functional Programming
TDA283/DIT300, Compiler construction
DAT151/DIT231, Programming language technology
DAT350/DIT233, Types for programs and proofs

January 21, 2021

1 Introduction

Dependent type theory has lent much of its power to the proof-assistant systems like Coq [3], Lean [11], and functional programming languages like Agda [4] and Idris [5], 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. One can say that the system of types brings order to the programming language in the sense that well typed programs exclude a large subset 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]. In the light of this, dependently typed languages are those on the language spectrum with the strongest type system that ensures the highest level of correctness and precision, which makes it a natural option in the building of proof-assistant systems.

The downside of introducing a strong type system into a language lies in its difficulty of implementation. In a paper about modal dependent type theory [12], the authors argued that in any typed language, one (particularly the type checker) needs to decide whether one type is equal to another. In a dependent type theory, since a type may contain any value, deciding the equality of types entails deciding the equality of any terms, which requires much more computation. Besides, one should be careful in choosing the semantics of the language to allow it to be flexible enough in expressing dependent types, yet not too flexible to make the type checking undecidable. The main aim of this project is to explore a good *definition mechanism*, such that the work of deciding the equality of two terms in a dependently typed language could be alleviated. The meaning of the definition mechanism will be explained in the following sections and our work will be conducted upon a published functional language called Mini-TT [10].

Mini-TT is a small functional language with dependent types. It has typed lambda calculus as its core, extended with language features such as dependent products (Π), dependent sums (Σ), a unit type (U), labelled sums (used both in pattern matched function and data type construction), recursive definitions and pattern abstractions and bindings. Particularly noted is its type checking algorithm, which reduces the problem of type checking to checking convertibility of terms, which is further reduced to checking syntactic identity of the normal forms. This is done by first evaluating an expression to its value and then applying a readback function taking the value to a normal expression. Values in Mini-TT are expressed in weak head normal form [1]. The type checking algorithm in Mini-TT is decidable (means termination is ensured), the proof is not given but the conclusion is justified in the paper. We rephrase theses features here because our definition mechanism be built on the type inference

rules of Mini-TT, thus a good understanding of its mechanism is important.

2 Problem

Mini-TT was published in 2009 (more than 10 years from now), when it was intended to be a description of core Agda [4]. The design ideas behind Mini-TT have been used in a project called *cubicaltt* [13] [17]. A more recent work related with Mini-TT could be found at [6], which is about a proposal on developing an embedded core for Agda [4]. There are two problems in Mini-TT that we'd like to address in this project. One, the approach taken by Mini-TT of reducing proof checking to type checking is based on a definition mechanism which is not clearly described at the theoretical level, and it could be implemented in a more efficient way. Two, the current definition of Mini-TT lacks a mechanism for handling implicit arguments. Implicit argument handling is a quite desirable feature in a proof-assistant system, for it allows users to declare certain arguments as being implicit and ignore them in the subsequent proof, which usually results in a more concise and elegant style of writing.

3 Context

So, what is exactly the problem with a definition mechanism and why is it important? A *definition* in the context of functional programming is just a *declaration* of the form $p : A = M$, which means that a constant p has type A with a definition M. A *definition mechanism* is not about how constants (or variables) should be declared, but how they should be evaluated. Different ways of evaluation will have big difference in terms of the efficiency on the implementation of type checking. The question of a definition mechanism could be rephrased more explicitly as "How to do type checking in presence of definitions?", which could be further reduced as "How to do conversion checking in presence of definitions?". In essence, the definition problem is about optimising the amount of computation to achieve higher level system efficiency. This problem is important because deciding efficiently the conversion of λ-terms has arguably become one of the most significant bottleneck in the implementation of proof checker of dependent type theories. There are some discussions in one unpublished draft [2] which I find suitable to be cited in the following sections for further elaboration on this point.

In systems where the property of confluence holds, deciding the convertibility of two terms amounts to finding a common reduct. When strong normalisation holds, the simplest algorithm is by comparing the normal forms of these two input terms (as does by Mini-TT). However, this algorithm does not scale to large theories, for

- First, producing the normal form may require much more reduction steps than necessary. For example, consider a system where addition and mul-

tiplication of natural numbers are defined. To prove $(1+1)^{10} = 2^{(5+5)}$ by normalisation, we need to reduce the term on each side to 1024. However, much less computation is needed if we can prove it by just showing $1+1=2$ and $10=5+5$.

- Second, the size of terms by expanding may grow exponentially, e.g. $x_n := (x_{n-1}, x_{n-1})$.

Also, there are times when computation can be reasoned by using already proved algebraic properties, without a fully expansion of definition. For example, we can reason about the equality of two terms $1+(2+3)$ and $(1+2)+3$ by the associativity property of addition operation, without computing the final result of each term. These examples above suggest an natural approach for the definition problem by introducing a locking/unlocking mechanism for constants. The ideal effect is that, during type checking, whenever possible, the definitions of certain constants could be locked from expansion without imposing any negative impact on the proof, such that computations could be saved. This is more similar to how humans reason in mathematics, where after some steps, one may forget the definitions of some notions, but is able to continue reasoning as if these notions were primitives.

For the second concern of our project, an implicit argument handling mechanism for Mini-TT, it is enough to illustrate this feature by giving a simple example in Agda:

```
id : {A : Set} -> A -> A
id x = x
```

Here in the example, A is declared as an implicit argument. This enables the user to give a definition of *id* without providing the first argument. As mentioned above, implicit argument gives users access to a cleaner way of writing.

4 Goals and Challenges

The goal of the project is to give extension and simplification for the work presented in the Mini-TT paper [10]. Specifically, it includes:

1. A simplification of Mini-TT based on a presentation [9] given by Thierry.

2. An extension of definition mechanism with locking/unlocking features.

3. An extension of implicit argument handling mechanism, depending on the progress of the two goals above (optional).

4. Keep simplicity, which means that the overall implementation should be as simple as possible, that it should not contort the type inference rules of

the original work unnecessarily or in a confused way, nor should it increase the order of time complexity of the type checking algorithm dramatically.

The goals above should lead to an actual implementation of type checker in Haskell, which should be able to illustrate the notions of the goals above and handle suitable examples. Particularly, for the purpose of testing the locking/unlocking mechanism, Thierry provided a small example in Agda (a representation of a variation of Hurkens paradox [14]), which is included in the appendix. It is expected to use the locking mechanism to prove the last theorem.

For the definition mechanism part, the main challenge comes from a lack of published works that have described the definition problem on a theoretical level. As Thierry puts it, "even for 'ordinary' dependent type theory, what is almost never presented is how to implement 'definitions' in type theory", and "there is no clear answer even after 50 years of study of this problem".

For the implicit argument handling part, Thierry has suggested using the idea of *proof-carrying code* [18]. The challenge comes from my unfamiliarity in this area, which means it may take some time to understand the mechanism of the proof-carrying code, and some extra time to get inspiration to apply the idea into implicit argument handling of dependent type theory.

The final challenge comes to the justification of the simplicity of the implementation. One should give clear evidence to support the claim that the goal of simplicity is indeed satisfied.

5 Approach

A basic Haskell implementation based on the presentation [9] would be quite straightforward, as it already contains the simplified syntax and type inference rules. Some work with regard to this could be found at [7] [8]. Besides, the original paper of Mini-TT [10] contains a complete implementation of the type checker in Haskell, which provides a suitable code template.

As for the definition mechanism, although published work describing this problem in a theoretical level is scarce, some pioneering work on this problem has already been done. In the unpublished draft [2] we mentioned above, the author introduced a locking/unlocking mechanism based on the notion of *transparency annotations*, with detailed descriptions on the properties of this mechanism and how type checking judgements should be derived. There's also a definition mechanism implemented in work *cubicaltt*, although less documented.

Since the problem of definition is essentially the problem of evaluation, as we have explained above, a good understanding of the evaluation mechanism should be a prerequisite. For this aspect, a work named "smalltt" [15] is worth noted,

as in a presentation titled "Fast Elaboration for Dependent Type Theories" of this project, the author classified values into so-called "glued values" and "local values", based on different purposes of evaluation (conversion checking or meta solution generation).

Our approach for the definition problem is to study carefully these pioneering work, try to combine the ideas behind these work and incorporate them into the implementation of our system. It seems that the focus of the work "transparency" [2] is on building a locking/unlocking mechanism consistently, whereas the focus of "smalltt" is on introducing a hierarchy mechanism, such that different extent of constant expansion during evaluation could be achieved. A preferable result would be a combination of the two.

The approach for handling hidden arguments is to use the technique of *proof-carrying code*. Since this aim is optional, we may or may not get this far, however if we did get to this point, a comparison between our approach and Agda should be made to illustrate the pros and cons of our approach.

6 References

References

[1] See definition of weak head normal form. `https://wiki.haskell.org/Weak_head_normal_form`. Accessed: 2021-01-10.

[2] Bruno Barras. A module system based on opacity. June 21, 2016.

[3] Yves Bertot. A short presentation of coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 12–16. Springer, 2008.

[4] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[5] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552, 2013.

[6] Jesper Cockx. A trustworthy and extensible core language for agda. `https://jesper.sikanda.be/posts/veni-announcement.html`. Accessed: 2021-01-11.

[7] Thierry Coquand. Related work: Code. `http://www.cse.chalmers.se/~coquand/conv1.hs`. Accessed: 2021-01-17.

[8] Thierry Coquand. Related work: Presentation. `http://www.cse.chalmers.se/~coquand/mTT10.pdf`. Accessed: 2021-01-17.

[9] Thierry Coquand. A simple programming language. http://www.cse.chalmers.se/~coquand/bengt.pdf, 2009. Accessed: 2021-01-12.

[10] Thierry Coquand, Yoshiki Kinoshita, Bengt Nordström, and Makoto Takeyama. A simple type-theoretic language: Mini-tt. *From Semantics to Computer Science; Essays in Honour of Gilles Kahn*, pages 139–164, 2009.

[11] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[12] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[13] Simon Huber. A cubical type theory. https://pdfs.semanticscholar.org/6c03/8772b1ead12dc1de8e4440945f351d66eac2.pdf, 2015.

[14] Antonius JC Hurkens. A simplification of girard's paradox. In *International Conference on Typed Lambda Calculi and Applications*, pages 266–278. Springer, 1995.

[15] Andras Kovacs. Smalltt. https://github.com/AndrasKovacs/smalltt/blob/master/krakow-pres.pdf. Accessed: 2021-01-17.

[16] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[17] Anders Mörtberg. Cubicaltt on github. https://github.com/mortberg/cubicaltt. Accessed: 2021-01-11.

[18] George Necula. Resources of proof-carrying code. https://people.eecs.berkeley.edu/~necula/papers.html. Accessed: 2021-01-17.

7 Appendix: Hurkens Paradox

```
{-# OPTIONS --type-in-type #-}

module loop2 where

Pow : Set -> Set
Pow X = X -> Set

T : Set -> Set
T X = Pow (Pow X)
```

```
funT : (X Y : Set) -> (X -> Y) -> T X -> T Y
funT X Y f t g = t (λ x -> g (f x))


postulate ⊥ : Set
postulate foo : ⊥ -> ⊥

¬ : Set -> Set
¬ X = X -> ⊥

U : Set
U = (X : Set) -> (T X -> X) -> X

tau : T U -> U
tau t X f = f (λ g -> t (λ z -> g (z X f)))

sigma : U -> T U
sigma z = z (T U) (λ t g -> t (λ x -> g (tau x)))

Q : T U
Q p = (z : U) -> sigma z p -> p z

B : Pow U
B z = ¬ ((p : Pow U) -> sigma z p -> p (tau (sigma z)))

C : U
C = tau Q

lem1 : Q B
lem1 z k l = l B k (λ p -> l (λ z -> p (tau (sigma z))))

A : Set
A = (p : Pow U) -> Q p -> p C

lem2 : ¬ A
lem2 h = h B lem1 (λ p -> h (λ z -> p (tau (sigma z))))

lem3 : A
lem3 p h = h C (λ x -> h (tau (sigma x)))

loop : ⊥
loop = lem2 lem3
```