Simple λ_Π-interpreter for matrix computation

Nicholas Huang
Department of Computer Science, Stanford
nykh@stanford.edu

ABSTRACT

This project to extend the simply-typed Lambda calculus $\lambda_\to$ to have types dependent on constant value. In particular this language will be used to check the validity of matrix computation.

KEYWORDS

dependent type, matrix computation, numeric computation

1 BACKGROUND

Historically, Matrix computation and linear algebra support were not implemented as core language feature but usually with third-party library support, except in mathematics-oriented language like Matlab, Mathematica, or Octave. Most of the languages used for matrix computation are scripting languages in the framework used in our course. In those languages, dimension check is performed at run-time, by checking the dimension information contained in the matrix struct. `numpy` a python numerical computation library, is a good example, because it is widely used in engineering and machine learning research [6]. When we check the type of an arbitrary vector or matrix (`numpy` differentiates between vector and matrix as two different classes), they are shown as simply `numpy.ndarray` and `numpy.matrixlib.defmatrix.matrix`. Their types do not encode the dimension information.

Intuitively, a compiled language is beneficial for matrix computation because potentially it can perform optimization, parallelization, as well as catching errors at compile time, including dimension checking. One of the popular matrix computation library for rust is called `nalgebra` [1]. This language does (optionally) encode matrix dimension in its types. But it does so very clumsily. Because in Rust, type cannot include any term, the library defines types `U1` through `U127` so as to allow the user to write matrix with type information that include dimension, for example

```
Matrix<N, U2, U4, S>
```

ecti which means a 2×4-Matrix of scalar type. The type level integer representation is in something similar to Church encoding, but instead of unary representation it uses a binary representation [2]. This can be seen as a work-around to the lack of dependent type in Rust. And at least since 2016, there has been an on-going discussion about the possibility of adding (constant) dependent type into Rust [7]. This speaks to the demand for such type constructs.

In this project my goal is to extend the simply-typed lambda calculus $\lambda_{\to!}$ to also support a simple version of dependent type that depends on constant integer, and allow simple linear algebra program written in it to be dimension-checked at compile time. The actual goal of choosing to do this project, however, is to understand the dependent type. This is a logical extension on our discussion on Type Theory, following lambda calculus, and polymorphism. However as it turns out, introducing true dependent type to the type system does not make the implementation more complicated.

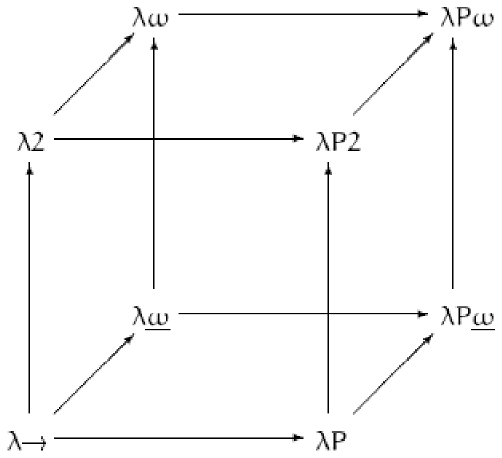

Figure 1: Lambda Cube diagram. Source: https://en.wikipedia.org/wiki/Lambda_cube

Now we must discuss the central topic of the project.

1.1 Dependent Type

Dependent type is a well-studied concept in type theory. To illustrate the bigger picture, refer to the **Lambda Cube diagram** shown in Figure-1. The Lambda cube is a framework to understand the relationship different different type systems, proposed by Barendregt [3]. In this cube diagram, each vertex corresponds to one type system. At the origin in the bottom left is the simply typed lambda calculus ($\lambda_\to$). The three axes refer to the three ways $\lambda_\to$can be extended. The second axis going in the vertical direction introduces polymorphism, i.e. value depending on type. The second axis projecting into the page leads to weak lambda omega $\lambda_{\underline{\omega}}$, which introduces type functions, i.e. **types** that depend on other **types**. These concepts are in fact all present in the `Lam2` interpreter we wrote for Assignment 4. The new axis is the horizontal axis in the cube diagram. The new dependency that was allowed was for **type** to depend on **value**, i.e. dependent type. A lambda calculus including dependent type is denoted λP on the diagram, but we will continue using λ_Π in the rest of the text.

There is a vast literature about dependent type and type theories in general. I however focused on the implementation aspect of the type system. Coq is a well developed proof assistant language that contains dependent type [4]. A more programming-oriented example is Agda, a programming language first developed as a PhD thesis by Ulf Norell with proof support. It has syntax similar to Haskell. Another dependent type language with a Haskell-like syntax is Idris [5]. Dependent type allows Idris to define type that depends on value as shown in Figure-2. (The `total` keyword invokes the totality checker which is less relevant to the topic).

```
infixr 5 :

data Vect : Nat -> Type -> Type where
  Nil  : Vect 0 a
  (::) : (x : a) -> (xs : Vect n a) -> Vect (n + 1) a

total
app : Vect n a -> Vect m a -> Vect (n + m) a
app Nil       ys = ys
app (x :: xs) ys = x :: app xs ys
```

Figure 2: Vector example in Idris

Theoretically, adding dependent type to the simply typed lambda calculus is actually very simple. In the original lambda with type functions, we had pairs of Abstraction and Application for term (`Lam`, `App`) and for type (`TLam`. `TApp`), to reflect the fact that value can be computed from value and types, and type can be computed from types. But as types can now be computed with value as well, we will not differentiate functions on the value and type levels anymore but instead define a single **dependent function space** that can take either type or value and map to either type or value. In fact, in this system, types and values are now unified.

2 APPROACH

I extended the simply typed Lambda calculus interpreter from Assignment 4 to include dependent type. To do this I had to

(1) Add a new type called `Type`, which is the type of types.
(2) Merge the `Lam` type and `TLam` term to a new type called `Pi`, with the signature of that of the `Fn` term. The idea is the old `TLam` will now be expressed with its argument taking the `Type` type defined above
(3) Merge the `Forall` type with the `Fn` type.

3 RESULT

Overall I acknowledge my implementation was very imcomplete. I have not implemented the matrix computation portion in the system. I wish to do so before the presentation.

But in terms of understanding dependent type system, I feel I have made good progress.

REFERENCES

[1] 2017. nalgebra Quick Reference: matrices and vectors. (2017). Retrieved Dec 15, 2017 from http://nalgebra.org/quick_reference/#matrices-and-vectors

[2] 2017. Struct typenum::uint::UInt. (2017). Retrieved Dec 15, 2017 from https://docs.rs/typenum/1.5.1/typenum/uint/struct.UInt.html

[3] Henk Barendregt. 1991. Introduction to Generalized Type Systems. *J. Funct. Program.* 1, 2 (1991), 125–154.

[4] Yves Bertot and Pierre Castéran. 2004. Interactive Theorem Proving and Program Development. CoqArt: The Calculus of Inductive Constructions. Springer Verlag.

[5] Edwin Brady. 2017. Idris: A Language with Dependent Types. (2017). Retrieved Dec 15, 2017 from https://www.idris-lang.org/

[6] NUMPY 2017. Numpy. (2017). Retrieved Dec 15, 2017 from http://www.numpy.org/

[7] ticki. 2016. RFC: const-dependent type system. (June 2016). Retrieved Dec 15, 2017 from https://github.com/rust-lang/rfcs/pull/1657