Design of the lambda cube interpreter

Thomas Tan

18 March 2019

1 Basic syntax and object representation

- Prop ($*$, denoted by $*$ or Prop in language, CocSyntaxProp)
- Type ($\Box$, denoted by @ or Type in language, CocSyntaxType)
- variables (alphanumeric names, CocSyntaxVariable label)
- application ((a b) where a and b are terms, CocSyntaxApply function argument)
- lambda abstraction ((\a:B.c) where a is a variable, B is the type, and c is the body, CocSyntaxLambda param inType body)
- forall abstraction aka product type ({\a:B.c} where a is a variable, B is the type, and c is the body, CocSyntaxForall param inType body)

We should first convert this to an internal representation using de Brujin indices, but we keep the labels around for fun (readability):

- "Objects" (CocObject)
  - Prop (CocProp)
  - Type (CocType)
  - variables (CocVariable label index index being the deBrujin index, 0 if it's used as the param in an abstraction)
  - application (CocApply function argument)
  - lambda abstraction (CocLambda param inType body)
  - forall abstraction aka product type (CocForall param inType body)
- With typing judgements
  - CocValue { cocObject :: CocObject, cocType :: Maybe CocObject }

2 Theory

2.1 Reduction rules

Firstly we write the reduction rules:

$$\frac{}{(\lambda x : A.B)C \mapsto B[C/x]}\ \textsc{BetaReduce}$$

$$\frac{B \mapsto B'}{(\lambda x : A.B) \mapsto (\lambda x : A.B')}\ \textsc{LambdaBodyReduce}$$

$$\frac{A \mapsto A'}{(\lambda x : A.B) \mapsto (\lambda x : A'.B)}\ \textsc{LambdaTypeReduce}$$

$$\frac{B \mapsto B'}{(\Pi x : A.B) \mapsto (\Pi x : A.B')}\ \textsc{ForallBodyReduce}$$

$$\frac{A \mapsto A'}{(\Pi x : A.B) \mapsto (\Pi x : A'.B)}\ \textsc{ForallTypeReduce}$$

2.2 Construction rules

And now the typing judgements. These rules double as rules to build up a set of (valid) terms in the language. Here s_1 and s_2 represent sorts which are either $*$ or $\Box$ as mentioned before. Calculus of constructions allow all combinations of s_1 and s_2 but weaker systems are restricted.

$$\frac{}{\vdash * : \Box}\ *C$$

$$\frac{}{\vdash *}\ CtxC$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash b : B \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)}\ \lambda C$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2}\ \Pi C$$

$$\frac{\Gamma \vdash C : (\Pi x : A.B) \qquad \Gamma \vdash D : A}{\Gamma \vdash C\ D : B[D/x]}\ AppC$$

$$\frac{\Gamma_L, x : A, \Gamma_R \vdash *}{\Gamma_L, x : A, \Gamma_R \vdash x : A}\ VarC$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash B : C}{\Gamma, x : A \vdash B : C}\ Weak$$

The $\varnothing C$ rule may seem out of place but it's purpose is to allow the construction of variables via $VarC$. In the original paper, it was used to identify which contexts were "valid". It used the following rules as well

$$\frac{\Gamma \vdash \Delta}{\Gamma, x : \Delta \vdash *}\ CtxExt1$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash *}\ CtxExt2$$

which builds up valid contexts. But these rules can be replicated with the $Weak$ rule, I believe.

The $Weak$ rule is necessary to allow construction of sub-expressions which do not use every variable in the context, and it's obvious to see why it's necessary for even tiny expressions.

2.3 Examples

Here's a derivation for a simple λ term.

$$\dfrac{\dfrac{}{\vdash * : \Box}\ *C \qquad \dfrac{\dfrac{}{\vdash * : \Box}\ *C \quad \dfrac{}{\vdash * : \Box}\ *C}{x : * \vdash * : \Box}\ Weak}{\vdash (\Pi x : *.*) : \Box}\ \Pi C$$

Here's a derivation for a Π term.

$$\dfrac{\dfrac{}{\vdash * : \Box}\ *C \qquad \dfrac{\dfrac{\dfrac{}{\vdash * : \Box}\ *C}{x : * \vdash *}\ CtxExt2}{x : * \vdash x : *}\ VarC \qquad \dfrac{\dfrac{}{\vdash * : \Box}\ *C \quad \dfrac{}{\vdash * : \Box}\ *C}{x : * \vdash * : \Box}\ Weak}{\vdash (\lambda x : *.x) : (\Pi x : *.*)}\ \lambda C$$

For apply terms we run into a small hurdle... Notice that there are unknowns in the precedents that have to be solved.

$$\dfrac{\dfrac{}{\vdash (\lambda x : *.x) : (\Pi x : ??.*)} \qquad \dfrac{}{\vdash (\Pi y : *.*) : ??}}{\vdash ((\lambda x : *.x)\ (\Pi y : *.*)) : *[(\Pi y : *.*)/x]}\ AppC$$

However, it is plain to see that if the function term is known, as it is here $(\lambda x : *.x)$, then we can see that because the only way to derive a λ term is through λC, we know that the ?? must be equal to the type of the lambda's parameter, namely $*$.

We can write a slightly modified *NAppC* rule that takes handles application where the function term starts with a lambda constructor.

$$\frac{\vdash (\lambda x : A.b) : (\Pi x : A.B) \qquad \vdash D : A}{\vdash ((\lambda x : A.b)\ D) : B[D/x]}\ NAppC$$

Now we can do the original derivation:

$$\frac{\dfrac{\vdots}{\vdash (\lambda x : *.x) : (\Pi x : *.*)}\ \lambda C \qquad \dfrac{\vdots}{\vdash (\Pi y : *.y) : *}\ \Pi C}{\vdash ((\lambda x : *.x)\ (\Pi y : *.y)) : *[(\Pi y : *.y)/x]}\ NAppC$$

We can see that naively we already need some basic form of type inference just to determine if terms are derivable.

More issues arise when for example, we only write that

$$\vdash ((\lambda x : *.x)\ (\Pi y : *.y)) : *$$

and not the pre-substitution form written above.

The result is still true, but how do we run the rules backwards to determine its validity?

This motivates the introduction of conversion rules which bake the idea of equality containing beta-reduction into the type system.

The equality rules take up a lot of space so I won't bother showing it here. Instead let's just do a quick derivation for the above problematic judgement.

$$\frac{\dfrac{\vdots}{\vdash (\Pi y : *.y) : *} \qquad \dfrac{\begin{array}{c}\dfrac{\vdots}{\vdash ((\lambda x : *.x)\ (\Pi y : *.y)) \cong x[(\Pi y : *.y)/x]} \\ \vdash x[(\Pi y : *.y)/x] \cong (\Pi y : *.y)\end{array}}{\vdash ((\lambda x : *.x)\ (\Pi y : *.y)) \cong (\Pi y : *.y)}}{\vdash ((\lambda x : *.x)\ (\Pi y : *.y)) : *}\ \beta \cong$$

However just because they are now derivable judgementally doesn't mean it's decidable or efficient. The original paper says that there exists an efficient (polytime) algorithm to determine if a term is well typed, but doesn't describe it.

I haven't actually found any article that describes the algorithm proper, so I've written it down below.

2.4 Find type

We'll define a function $(\mathit{ft}\ \Gamma A) = B$ that takes in a context Γ and an expression A, and its result will be its type B.

The hope is that the following rule is admissible,

$$\frac{(\mathit{ft}\ \Gamma\ A) = B}{\Gamma \vdash A : B}$$

which would let us use ft as a subroutine or preprocessing step to check that terms are well typed.

$$\frac{\Gamma \vdash *}{(\mathit{ft}\ \Gamma\ *) = \Box}\ \mathit{ft}*$$

$$\frac{\Gamma_L, x : A, \Gamma_R \vdash *}{(\mathit{ft}\ \Gamma_L, x : A, \Gamma_R\ x) = A}\ \mathit{ftVar}$$

$$\frac{(\mathit{ft}\ \Gamma\ A) = s_1 \qquad (\mathit{ft}\ \Gamma, x : A\ b) = B \qquad (\mathit{ft}\ \Gamma, x : A\ B) = s_2}{(\mathit{ft}\ \Gamma\ (\lambda x : A.b)) = (\Pi x : A.B)}\ \mathit{ft}\lambda$$

$$\frac{(\mathit{ft}\ \Gamma\ A) = s_1 \qquad (\mathit{ft}\ \Gamma, x : A\ B) = s_2}{(\mathit{ft}\ \Gamma\ (\Pi x : A.B)) = s_2}\ \mathit{ft}\Pi$$

$$\frac{(\mathit{ft}\ \Gamma\ C) = (\Pi x : A.B) \qquad (\mathit{ft}\ \Gamma\ a) = A' \qquad A = A'}{(\mathit{ft}\ \Gamma\ (C\ a)) = B}\ \mathit{ftApp}$$

Here's an trace of the algorithm on a simple term. I'll denote in superscripts a step number which relates where the value comes from.

$$\dfrac{\dfrac{\dfrac{}{(ft\ \varnothing\ *) = \Box} \qquad \dfrac{x : * \vdash *^{1}}{(ft\ x : *\ x) = *^{1}}}{(ft\ \varnothing\ (\lambda x : *.x)) = (\Pi x : *^{2}.*^{1})} \qquad \dfrac{\dfrac{}{(ft\ \varnothing\ *) = \Box} \qquad \dfrac{x : * \vdash *}{(ft\ x : *\ x) = *^{3}}}{(ft\ \varnothing\ (\Pi x : *.x)) = *^{3}} \qquad *^{2} = *^{3}}{(ft\ \varnothing\ ((\lambda x : *.x)\ (\Pi x : *.x))) = *^{1}}$$