Categorical Semantics of Dependent Type Theory for Cubical Syntax

Maksym Sokhatskyi [1] and Pavlo Maslianko [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute

July 27, 2018

Abstract

Categorical Semantics of Dependent Type Theory.

Keywords: Formal Methods, Type Theory, Programming Languages, Theoretical Computer Science, Applied Mathematics, Cubical Type Theory, Martin-Löf Type Theory

Contents

1 Intro

Here is a short informal description of categorical semantics of dependent type theory given by Peter Dybjer. The code is by Thierry Coquand ported to cubical by 5HT [1].

Definition (Fam). Fam is the category of families of sets where objects are dependent function spaces $(x : A) \to B(x)$ and morphisms with domain $Pi(A, B)$ and codomain $Pi(A', B')$ are pairs of functions $< f : A \to A', g(x : A) : B(x) \to B'(f(x)) >$.

Definition (Derivability). $\Gamma \vdash A = (\gamma : \Gamma) \to A(\gamma)$.

Definition (Comprehension). $\Gamma; A = (\gamma : \Gamma) * A(\gamma)$. Statement. Comprehension is not assoc.

$\Gamma; A; B \neq \Gamma; B; A$

Definition (Context). The C is context category where objects are contexts and morphisms are substitutions. Terminal object $\Gamma = 0$ in C is called empty context. Context comprehension operation $\Gamma; A = (x : \Gamma) * A(x)$ and its eliminators: $p : \Gamma; A \vdash \Gamma$, $q : \Gamma; A \vdash A(p)$ such that universal property holds: for any $\Delta : ob(C)$, morphism $\gamma : \Delta \to \Gamma$, and term $a : \Delta \to A$ there is a unique morphism $\theta =< \gamma, a >: \Delta \to \Gamma; A$ such that $p \circ \theta = \gamma$ and $q(\theta) = a$. Statement. Subst is assoc.

$\gamma(\gamma(\Gamma, x, a), y, b) = \gamma(\gamma(\Gamma, y, b), x, a)$

Definition (CwF-object). A CwF-object is a $Sigma(C, C \to Fam)$ of context category C with contexts as objects and substitutions as morphisms and functor $T : C \to Fam$ where object part is a map from a context Γ of C to famility of sets of terms $\Gamma \vdash A$ and morphism part is a map from substitution $\gamma : \Delta \to \Gamma$ to a pair of functions which perform substitutions of γ in terms and types respectively.

Definition (CwF-morphism). Let $(C, T) : ob(C)$ where $T : C \to Fam$. A CwF-morphism $m : (C, T) \to (C', T')$ is a pair $< F : C \to C', \sigma : T \to T'(F) >$ where F is a functor and σ is a natural transformation.

Definition (Category of Types). Let we have CwF with (C,T) objects and $(C, T) \to (C', T')$ mophisms. For a given context Γ in Ob(C) we can construct a $Type(\Gamma)$ – the category of types in context Γ with set of types in contexts as objects as and functions $f : \Gamma; A \to B(p)$ as morphisms.

Definition (Local Cartesian Closed Category).

$LCCC(C) = Sigma(C, (A : obC) \to CCC(C/A))$.

Usually, from the mathematical point of view, there are no differences between different syntactic proofs of the same theorem. However, from the programming point of view, we can think of code reuse and precisely defined math libraries that reduce the overall code size and provide simplicity for operating complex structures (most heavy one is the categorical library). So in this work, we will focus on proper decoupling and more programming friendly base library still usable for mathematicians.

```
data Exp = Star (_:    nat)
         | Var  (_:    nat)
         | Pi   (_ _: Exp)
         | Lam  (_:    Exp)
         | App  (_ _: Exp)
```

[1] http://www.cse.chalmers.se/ peterd/papers/Ise2008.pdf

```
Ty:     U = Exp
Ctx:    U = list Ty
Subst: U = list Exp
seq (start: nat): list Exp = cons (Var start) (seq (suc start))

mutual

p:    Subst = seq one
q:    Exp   = Var zero
ide: Subst = seq zero
cmp: Subst -> Subst -> Subst = split
     nil -> \(ts: Subst) -> nil
     cons x xs -> \(ts: Subst) -> cons (sub ts x) (cmp xs ts)

lift (ts: Subst): Subst = cons q (cmp ts p)
unwrap: maybe Exp -> Exp = split { nothing -> q ; just x -> x }
shift (t: Exp) (i: nat): Exp = sub (seq i) t

isCo: Ctx -> bool = split
  nil -> true
  cons x xs -> and_ (isCo xs) (isTy xs x)

isU (c:Ctx)(a b:Exp): Exp -> bool = split
  Star  i   -> (and_ (isTm c (Star i) a) (isTm (cons a c) (Star i) b))
  Var   i   -> false
  Pi    a b -> false
  Lam     x -> false
  App   a b -> false

isPi (c:Ctx)(e:Exp): Exp -> bool = split
  Star  i   -> false
  Var   i   -> false
  Pi    a b -> isTm (cons a c) b e
  Lam     x -> false
  App   a b -> false

isTy (c:Ctx): Ty -> bool = split
  Star  i   -> true
  Var   i   -> isTm c (Star zero) (Var i)
  Pi    a b -> and_ (isTy c a) (isTy (cons a c) b)
  Lam     x -> isTm c (Star zero) (Lam x)
  App   a b -> isTm c (Star zero) (App a b)

isTm (c:Ctx)(e:Exp): Ty -> bool = split
  Star  i   -> false
  Var   i   -> false
  Pi    a b -> isU c a b e
  Lam     x -> isPi c x e
  App   a b -> false
```

```
app (s: Exp): Exp -> Exp = split
  Star  i   -> App (Star i) s
  Var   i   -> App (Var i) s
  Pi    a b -> App (Pi a b) s
  Lam   x   -> sub (cons s ide) x
  App   a b -> App (App a b) s

sub (ts: Subst): Exp -> Exp = split
  Star  i   -> Star i
  Var   i   -> unwrap (nth Exp i ts)
  Pi    a b -> Pi  (sub ts a) (sub (lift ts) b)
  Lam   x   -> Lam (sub (lift ts) x)
  App   s t -> app (sub ts t) (sub ts s)

inferTy (c: Ctx): Exp -> maybe Ty = split
  Star  i   -> just (Star i)
  Var   i   -> just (shift (unwrap (nth Exp i c)) (suc i))
  Pi    a b -> just (Star one)   -- implement
  Lam   x   -> just (Star zero)  -- implement
  App   s t -> just (Star zero)  -- implement
```