NJPLS, Sept 2016

Executable Categorical Models of Type Theory

Gershom Bazerman /
S&P Global Market Intelligence

The Starting Point

Curry-Howard : Type Theoretic/Computational Semantics of Logic

———

Lawvere-Lambek : Categorical Semantics of Logic

Conversely

Programming Languages/Type Theories give rise to Logics

—>

Categories give rise to Logics

flavor of type theory	equivalent to	flavor of category theory	
intuitionistic propositional logic/simply-typed lambda calculus		cartesian closed category	
multiplicative intuitionistic linear logic		symmetric closed monoidal category	(various authors since ~68)
first-order logic		hyperdoctrine	(Seely 1984a)
classical linear logic		star-autonomous category	(Seely 89)
extensional dependent type theory		locally cartesian closed category	(Seely 1984b)
homotopy type theory without univalence (intensional M-L dependent type theory)		locally cartesian closed (∞,1)-category	(Cisinski 12-(Shulman 12)
homotopy type theory with higher inductive types and univalence		elementary (∞,1)-topos	see here
dependent linear type theory		indexed monoidal category (with comprehension)	(Vákár 14)

The Research Program

- Start with a Computational Encoding of Category Theory
- Directly Produce Embedded Programming Languages
- Study Relationships to Fully Typed Embeddings, Variable Binding Representation, Domain Specific Languages, etc.

Cartesian Closed Categories and the STLC

In Haskell / Agda we often have indexed terms of the form

```
data Term ctx typ where…
```

(where context need not only be free variables, but region markers, resource quantifiers, etc).

Infix that gives us

```
ctx :- typ
```

A full term in the object language has the type *precisely* of a typing judgement —
$\Gamma \vdash A$
(and an inhabitant of this type — a term in our host language that is also a term in our embedded language, is the computation that bears witness to this judgement).

Cartesian Closed Categories and the STLC

$\Gamma \vdash A$

Weakening on the left allows strengthening on the right.

The turnstile has mixed variance.

$\Gamma \rightarrow A$

$\mathrm{Hom}(\Gamma, A)$

New challenge: in what class of categories do contexts and terms live side by side as objects.

Cartesian Closed Categories and the STLC

Challenge: in what class of categories do contexts and terms live side by side as objects.

Approach: Study the structure necessitated by contexts, and then pick a category in which all objects have this structure.

1) Contexts have monoidal structure. You can append to them, you can drop from them, you can project from them.
2) Contexts have exponential structure. From A, A -> B we can conclude B.

Result: we take typing judgments to be given as homs of a cartesian closed category.

Cartesian Closed Categories and the STLC

We take typing judgments to be given as homs of a cartesian closed category.

In a natural deduction system we look particularly at those homs into a one element context.

Given a category C and a particular fixed element A, this yields a slice category C/A. In our simple setting, such categories themselves will not necessarily be cartesian closed.

This also yields a functor from each element A of our category of types to the set of all its inhabitants. Hence terms are fibers of presheaves.

Code

```
{-# LANGUAGE
DataKinds,
TypeOperators,
MultiParamTypeClasses,
TypeFamilies,
GADTs,
ScopedTypeVariables,
RankNTypes,
PolyKinds,
FlexibleContexts,
UndecidableInstances #-}
```

Code

```
-- We begin with objects of cartesian closed
categories over some base index of types.
data TCart b = TUnit
             | TPair (TCart b) (TCart b)
             | TExp (TCart b) (TCart b)
             | TBase b

-- Base indices are mapped to types via Repr
type family Repr a :: *

-- Cartesian objects over the base are mapped to
types via CartRepr
type family CartRepr a :: *
type instance CartRepr (Ty TUnit) = ()
```

Code

```
-- Ty is used to wrap polykinded things up in kind *
data Ty a

type instance CartRepr (Ty (TBase a))    =
    Repr (Ty a)
type instance CartRepr (Ty (TPair a b)) =
    (CartRepr (Ty a), CartRepr (Ty b))
type instance CartRepr (Ty (TExp a b))  =
    CartRepr (Ty a) -> CartRepr (Ty b)

data ABase = AInt | AString | ADouble

type instance Repr (Ty AInt) = Int
type instance Repr (Ty AString) = String
type instance Repr (Ty ADouble) = Double
```

Code

```
-- A Context b is a list of cartesian objects over
base index b
data Cxt b = CCons (TCart b) (Cxt b) | CNil

-- CxtArr a b is a judgment a |- b
-- when b contains multiple terms this is a sequent
--
-- CxtArr a b -> CxtArr c d is an inference rule
--    a |- b
-- ---------
--    c |- d

data CxtArr :: Cxt a -> Cxt a -> *  where
  -- To be a category we must have id and composition
  CXAId  :: CxtArr a a
  CXACompose :: CxtArr b c -> CxtArr a b -> CxtArr a
c
```

Code

```
  -- We have a terminal object
  CXANil :: CxtArr a CNil

  -- We have face maps
  CXAWeaken :: CxtArr (CCons a cxt) cxt

  -- We have degeneracy maps
  CXADiag :: CxtArr (CCons a cxt) (CCons a (CCons a
cxt))

  -- We have additional "degeneracy" maps given by
every inhabitant of our underlying terms
  CXAAtom :: CartRepr (Ty a) -> CxtArr cxt (CCons a
cxt)
```

Code

```
  -- We also have a cartesian structure
  CXAPair  :: CxtArr cxt (CCons a c2) -> CxtArr cxt
(CCons b c2) -> CxtArr cxt (CCons (TPair a b) c2)
  CXAPairProj1 :: CxtArr (CCons (TPair a b) cxt)
(CCons a cxt)
  CXAPairProj2 :: CxtArr (CCons (TPair a b) cxt)
(CCons b cxt)

  -- And a closed structure (aka uncurry and eval)
  CXAEval  :: CxtArr (CCons (TPair (TExp a b) a) cxt)
(CCons b cxt)
  CXAAbs   :: CxtArr (CCons a cxt) (CCons b c) ->
CxtArr cxt (CCons (TExp a b) c)
```

Code

```
-- We give axioms on our category as conditions on
coherence of composition
cxaCompose :: CxtArr b c -> CxtArr a b -> CxtArr a c
cxaCompose CXAId f = f
cxaCompose f CXAId = f
cxaCompose CXANil _ = CXANil
cxaCompose CXAPairProj1 (CXAPair a b) = a
cxaCompose CXAPairProj2 (CXAPair a b) = b
cxaCompose CXAWeaken CXADiag = CXAId
cxaCompose h (CXACompose g f) =
             CXACompose (cxaCompose h g) f
— this can get stuck
cxaCompose f g = CXACompose f g
```

Code

```
instance Category CxtArr where
    id = CXAId
    (.) = cxaCompose

data Term cxt a =
    Term {unTerm :: CxtArr cxt (CCons a CNil)}
```

This Yields de Bruijn

```
varTerm :: Term (CCons a CNil) a
varTerm = Term CXAId

absTerm :: Term (CCons a cxt) b -> Term cxt (TExp a
b)
absTerm = Term . CXAAbs . unTerm

appTerm :: Term cxt (TExp a b) -> Term cxt a -> Term
cxt b
appTerm f x = Term (CXAEval . (CXAPair (unTerm f)
(unTerm x)))

tm_id :: Term CNil (TExp a a)
tm_id = Term (CXAAbs CXAId)
-- tm_id = absTerm varTerm

tm_k :: Term CNil (TExp b (TExp a b))
tm_k = Term . CXAAbs . CXAAbs $ (CXAWeaken . CXAId)
```

There's Another Exponential

We're in a category of presheaves: Context^op -> Set
This category is cartesian closed by definition, with an exponential given for P, Q at an obje
as

Hom(y(C)xP,Q)
—>
Nat(y(C)xP,Q)
—>
forall D. y(C)(D) -> P(D) -> Q(D)
—>
forall D. Hom(C,D) -> P(D) -> Q(D)

```
CXALam :: (forall c.
           CxtArr c cxt ->
           CxtArr c (CCons a c2) ->
           CxtArr c (CCons b c2))
         -> CxtArr cxt (CCons (TExp a b) c2)
```

There's Another Exponential

```
  CXALam :: (forall c.
               CxtArr c cxt ->
               CxtArr c (CCons a c2) ->
               CxtArr c (CCons b c2))
            -> CxtArr cxt (CCons (TExp a b) c2)

cxaCompose CXAEval (CXAPair (CXALam f) g) = f CXAId g

lamt ::
  (forall c. CxtArr c cxt -> Term c a -> Term c b) ->
  Term cxt (TExp a b)
lamt f = Term (CXALam (\m x -> unTerm (f m (Term
x))))
```

Now we can interpret

```
-- Interpretation does the obvious thing
interp :: Term CNil a -> CartRepr (Ty a)
interp (Term (CXAAtom x)) = x
interp (Term (CXAPair f g)) = (interp (Term f),
                               interp (Term g))
interp (Term (CXALam f)) =
     interp . Term . f CXAId . unTerm . abst
interp (Term (CXAAbs f)) =
     interp (Term (CXALam $ \_ x -> f . x))

subst :: Term (CCons a cxt) t -> Term cxt a -> Term
cxt t
subst = appTerm . absTerm

nbe :: Term CNil a -> Term CNil a
nbe = abst . interp
```

Variable binding and HOAS

```
lam :: (forall c. Term c a -> Term c b) -> Term cxt (TExp a
lam f = lamTerm $ \ h -> f

tm_id = lam $ \x -> x

-- errr
tm_k = lam $ \x -> lamt $ \g y -> appArrow g x

-- cripes!
tm_s = lamt $ \h f ->
       lamt $ \h1 g ->
       lamt $ \h2 x ->
         appTerm (appTerm (appArrow (h1 . h2) f) x)
                 (appTerm (appArrow h2 g) x)
```

Variable binding and HOAS

A refresher:

“Plain” HOAS admits ‘exotic’ terms that can case on the value they are given.

We can recover a tight representation by forcing our HOAS terms to be polymorphic over t
type of the variable representation. (Weirich/Washburn)

However: as as discussed by Dan Licata in his thesis, “plain HOAS” isn’t logically bad, it ju
corresponds to something else — terms from the host language which are admissible into t
logic as axioms. (As opposed to terms in the host language which are derivable in the logic
tautologies).

Variable binding and HOAS

Claim/conjecture:

Terms written with our "Categorical Abstract Syntax" that do not inspect their arguments a
parametric (free) in the context they range over. This is precisely the statement that they are
derivable in any context.

Terms written in the same fashion that do inspect their arguments can only do so by fixing
type of the context. This is the statement that they are admissible in a particular context.

```
e.g.:
addOne :: Term CNil (TBase AInt) -> Term CNil (TBase AInt)
addOne = abst . (+(1::Int)) . interp
```

Note: the lattice structure of derivability and admissibility of terms should itself yield a realization in the internal hom of our category, a la PShf(C/j) =~ PShf(C)/y(j).

Variable binding and HOAS

```
but:
oops :: Term c (TBase AInt) -> Term c (TBase AInt)
oops (Term x) = case x of
    (CXAAtom x) -> Term (CXAAtom (1::Int))
    (CXACompose _ _) -> liftTerm $ Term (CXAAtom (5::Int))
```

We need to eliminate CXACompose or prove it never occurs or we're not in a genuinely fre
CCC. Conjecture: this is the same condition that determines parametric terms are genuinel
derivable terms.

One binder for the price of two

```
tm_s = lamt $ \h f ->
       lamt $ \h1 g ->
       lamt $ \h2 x ->
         appTerm (appTerm (appArrow (h1 . h2) f) x)
                 (appTerm (appArrow h2 g) x)
```

The reindexing term (morphisms in the slice) “forgets” to de Bruijn indexing, and induces t
bound term.

Forgetting the reindexing term results in traditional HOAS.

Yoneda: The Ultimate Lambda

$$\begin{tikzcd} & \text{Categorical Abstract Syntax} \arrow[dl] \arrow[dr] & \\ \text{Parametric HOAS} & & \text{de Bruijn} \end{tikzcd}$$

Related Work

“Introduction to Higher Order Categorical Logic,“ J. Lambek and P.J. Scott
“The Maximality of the Typed Lambda Calculus, and of Cartesian Closed Categories,” K. D
and Z. Petric
“Unembedding Domain-Specific Languages,” R. Atkey
“Embedding F,” S. Lindley
“Type Theory in Type Theory using Quotient Inductive Types,” A. Kaposi and T. Altenkirc

Future Work

- Linear Logics
- Dependent Theories (Contextual Categories, CwA, CwF).
- Parametric Theories (System F).
- Effectful theories (relation to coeffects).
- Formalization
- Translation of techniques to practical use
- Down with the bureaucracy of reindexing!

Thanks Due

This project is especially inspired by many conversations with Atze van der Ploeg. Additional valuable discussions particularly with Stephanie Weirich, Ambrus Kaposi, and Peter LeFanu Lumsdaine. Thanks also to all members of the NY Topos Theory Reading Group/Category Theory Seminar.