Modal type theory based on the intuitionistic epistemic logic.

Rogozin Daniel

MSU, Slam Labs

November, 2017

Motivation. Calculations with side-effects.

- Motivation lies within a pure functional programming in languages like Haskell, Purescript, Elm or Idris.
- Without loss of generality we divide the types in Haskell (or Purescript, Elm or Idris) into two parts: simple types and parametrized types.
- simple types: `Int`, `String`, `Char`, `Double`, `Unit`, etc.
  simple type is an usual data type as in the rest functional or imperative languages.
- Parametrized types: `List Int`, `Maybe Char`, `IO String`, etc.
  Parametrized types are used for calculations with some side effect (programs with input-output, partial functions, etc).
- By the same way, we divide the function into two parts: simple functions and parametrized functions:
- Simple functions: `Int -> Int`, `String -> Int`, etc.
- Parametrized function: `Maybe Int -> IO Int`, `[String] -> [Int]`, etc.

Motivation. Functor.

Type class `Functor` is a general interface for "uniform action over a parameterized type, generalizing the map function on lists":

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b.
```

Motivation. Monad.

According to Hackage: "From the perspective of a Haskell programmer, however, it is best to think of a monad as an abstract datatype of actions. " For instance, calculation in IO-world is the special case of modadic calculations.

Monad definition:

```
class Functor m => Monad m where
 return : a -> m a
 (>>=) :: m a -> (a -> m b) -> m b.
```

Sequence of actions as monadic composition:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
```

Final point: applicative functor and its use.

Applicative functor is stronger than functors, but weaker then monads (similar to strong lax monoidal functors with additional natural transformation):

```
class Functor f => Applicative f where
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

Using applicative functor, we may inject a value into the functor `f` with `pure` and perform application inside the `f` using `<*>`.

Uses:

- Generalize of `fmap` for an arbitrary sequence of embedded arguments into the functor: `pure f <*> a1 <*> ... <*> an`
- Parsing (example: Parsec library);
- Type composition.

Monadic calculations in theory.

1) *Eugenio Moggi.* "Notions of computation and monads." Inf. Comput., 93(1): 55–92, 1991.

2) *Frank Pfenning and Rowan Davies.* "A judgmental reconstruction of modal logic." Mathematical. Structures in Comp. Sci. 11, 4 (August 2001), 511—540.

3) *Bierman, G., and De Paiva, V.*. On an Intuitionistic Modal Logic. Studia Logica: An International Journal for Symbolic Logic, 65(3), 2000. 383–416.
etc...

Applicative functors in theory.

Unfortunately, applicative functor is unknown concept outside the community of haskellers. Possible reason: applicative functors are considered just from a programming point of view, i.e. without proof-theoretical consideration:

1) *Conor McBride and Ross Paterson.* "Applicative Programming with Effects." Journal of Functional Programming 18:1 (2008), pages 1–13.
2) *Ross Paterson.* "Constructing Applicative Functors." Mathematics of Program Construction, Madrid, 2012, Lecture Notes in Computer Science vol. 7342, pp. 300–323, Springer, 2012.

Large white spot: consider modal typed lambda calculus for axiomatizing of the applicative functor calculations.

Intuitionistic epistemic logic for (applicative) applications.

It is convenient to solve such a problem using some intuitionistic modal logic and building lambda calculus which would isomophic to this logic:

Definition

Intuitionistic epistemic logic IEL$^-$:
1) *IPC axioms;*
2) $\boldsymbol{K}(A \to B) \to (\boldsymbol{K}A \to \boldsymbol{K}B)$ *(normality);*
3) $A \to \boldsymbol{K}A$ *(co-reflection);*
Rule: MP.

1) *Artemov S., Protopopescu T.* (2014, June). Intuitionistic epistemic logic. ArXiv, math.LO 1406.1582v1.
2) *Krupski V. N., Alexey Y.* "Sequent calculus for intuitionistic epistemic logic IEL" // Logical Foundations of Computer Science – International Symposium, LFCS 2016, Deerfield Beach, FL, USA, January 4-7, 2016. Proceedings. – Vol. 9537 of Lecture Notes in Computer Science. – Springer, 2016. – P. 187–201.

Modal type theory based on the IEL$^-$.

We should get the natural deduction for IEL$^-$ and obtain modal lambda calculus by proof-assignment. In other words we add to STT the next typing rules for modality:

$$\frac{\Gamma \vdash x : \alpha}{\Gamma \vdash \texttt{pure}\ \ x : \mathbf{K}\alpha}\ \mathbf{K}_I$$

$$\frac{\Gamma \vdash f : \mathbf{K}(\alpha \to \beta) \qquad \Gamma \vdash x : \mathbf{K}\alpha}{\Gamma \vdash f\ \ \texttt{<*>}\ \ x : \mathbf{K}\beta}\ \mathbf{K}_{app}$$

Examples of derivation trees.

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{x : \alpha \vdash \texttt{pure x} : \mathbf{K}\alpha}\,\mathbf{K}_I}{\vdash (\lambda x.\texttt{pure } x) : \alpha \to \mathbf{K}\alpha}\,{\to_i}$$

$$\dfrac{\dfrac{\dfrac{f : \mathbf{K}(\alpha \to \beta) \vdash f : \mathbf{K}(\alpha \to \beta) \qquad x : \mathbf{K}\alpha \vdash x : \mathbf{K}\alpha}{f : \mathbf{K}(\alpha \to \beta), x : \mathbf{K}\alpha \vdash f \texttt{ <*> } x : \mathbf{K}\beta}\,\mathbf{K}_{app}}{f : \mathbf{K}(\alpha \to \beta) \vdash \lambda x.f \texttt{ <*> } x : \mathbf{K}\alpha \to \mathbf{K}\beta}\,{\to_i}}{\vdash \lambda f.\lambda x.f \texttt{ <*> } x : \mathbf{K}(\alpha \to \beta) \to \mathbf{K}\alpha \to \mathbf{K}\beta}\,{\to_i}$$

Substitution.

1) $x[x := N] = N$, $x[y := N] = x$;
2) $(MN)[x := N] = M[x := N]N[x := N]$;
3) $(\lambda x.M)[x := N] = \lambda x.M[x := N]$;
4) $(M, N)[x := P] = (M[x := P], N[x := P])$;
5) $(\pi_i M)[x := P] = \pi_i(M[x := P])$, $i \in \{1, 2\}$;
6) $(\texttt{pure}\ \ M)[x := P] = \texttt{pure}\ \ (M[x := P])$;
7) $(M\ \texttt{<*>}\ N)[x := P] = (M[x := P])\ \texttt{<*>}\ (N[x := P])$.

Reduction (for pure and <*>).

In Haskell any implementation of the Applicative instance should preserve the next laws:
1) (identity law) `pure id <*> v = v`;
2) (composition law) `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
3) (homomorphism law) `pure f <*> pure x = pure (f x)`
4) (interchange law) `u <*> pure y = pure ( \ f -> f y) <*> u`.

In $\lambda_{\mathbf{K}}$ we set this laws as reduction rules:
1) `pure` $(\lambda x.x)$ `<*>` $M \to_\beta M$;
2) `pure` $(\lambda f.\lambda g.\lambda x.f(gx))$ `<*>` M `<*>` N `<*>` $P \to_\beta M$ `<*>` $(N$ `<*>` $P)$;
3) (`pure` M) `<*>` (`pure` `N`) $\to_\beta$ `pure` (MN);
4) M `<*>` `pure` $N \to_\beta$ (`pure` $(\lambda f.fN)$) `<*>` M.

Additionaly:
1) $M_1 \to_\beta M_2 \Leftrightarrow$ `pure` $M_1 \to_\beta$ `pure` M_2;
2) $M_1 \to_\beta M_2 \Rightarrow M_1$ `<*>` $N \to_\beta M_2$ `<*>` N;
3) $N_1 \to_\beta N_2 \Rightarrow M$ `<*>` $N_1 \to_\beta M$ `<*>` N_2;

Metatheoretical properties. Non-interesting part (but important).

Lemma
If $\Gamma \vdash M : \alpha$, *then* $FV(M) \subseteq dom(\Gamma)$.

Lemma
1) $\Gamma \vdash \texttt{pure}\ \ M : \boldsymbol{K}\alpha$ *implies that* $\Gamma \vdash M : \alpha$;
2) $\Gamma \vdash M\ \texttt{<*>}\ N : \boldsymbol{K}\alpha$ *implies that* $\Gamma \vdash M : \boldsymbol{K}(\alpha \to \beta)$ *and* $\Gamma \vdash N : \boldsymbol{K}\beta$.

Lemma
Let $\Gamma \vdash M : \alpha$, *then, if* $\Gamma \subseteq \Delta$, *the* $\Delta \vdash M : \alpha$.

Metatheoretical properties. Subject reduction.

Lemma

If $\Gamma \vdash M : \alpha$ *and* $M \to_\beta N$, *then* $\Gamma \vdash N : \alpha$.

We may consider case with homomorphism law: Let $\Gamma \vdash (\texttt{pure}\ \ M)\ \texttt{<*>}\ (\texttt{pure}\ \ N) : \mathbf{K}\beta$, then $\Gamma \vdash \texttt{pure}\ \ (MN) : \mathbf{K}\beta$.
From the one hand:

$$\dfrac{\dfrac{\Gamma \vdash M : \alpha \to \beta}{\Gamma \vdash \texttt{pure}\ \ M : \mathbf{K}(\alpha \to \beta)} \qquad \dfrac{\Gamma \vdash N : \alpha}{\Gamma \vdash \texttt{pure}\ \ N : \mathbf{K}\alpha}}{\Gamma \vdash (\texttt{pure}\ \ M)\ \texttt{<*>}\ (\texttt{pure}\ \ N) : \mathbf{K}\beta}$$

From the other hand:

$$\dfrac{\dfrac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta}}{\Gamma \vdash \texttt{pure}\ \ (MN) : \mathbf{K}\beta}$$

Metatheoretical properties. Strong normalization and confluence.

Theorem

Any sequence of reductions terminates.

Proof.

By modifying and applying William Tait's method of logical relations.

□

Theorem

Let $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$, then there exists term D, such as $B \twoheadrightarrow_\beta D$ and $C \twoheadrightarrow_\beta D$.

Proof.

By checking local confluence (weak Church-Rosser property) and applying Newman's lemma.

□

Categorical model. Definitions from category theory

Definition

Monoidal category.
A monoidal category $\mathcal{C}$ is a category with:
1) A bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ called the tensor product;
2) An object $\mathbb{1} \in Ob(\mathcal{C})$ called the unit;
3) Forall $A, B, C \in Ob_{\mathcal{C}}$:
$A \otimes (B \otimes C) \cong (A \otimes B) \otimes C$;
$A \otimes \mathbb{1} \cong \mathbb{1} \otimes A \cong A$.
4) Monoidal category $\langle \mathcal{C}, \otimes, \mathbb{1} \rangle$ is a symmetrical if forall $A, B \in Ob(\mathcal{C}), A \otimes B \cong B \otimes A$

Definition

A lax monoidal functor between monoinal categories $\langle \mathcal{C}, \otimes, \mathbb{1} \rangle$ and $\langle \mathcal{D}, \otimes, \mathbb{1} \rangle$ is a functor $F : \langle \mathcal{C}, \otimes, \mathbb{1} \rangle \to \langle \mathcal{D}, \otimes, \mathbb{1} \rangle$ with the next natural transformations.
1) $u : \mathbb{1} \to F\mathbb{1}$ (unit property);
2) $* : FA \otimes FB \to F(A \otimes B)$ (application property).

Categorical model. Definition of Applicative functor.

Definition

Let $\langle \mathcal{C}, \otimes, \mathbb{1} \rangle$ is a symmetrical monoidal category. Then applicative functor $\mathcal{K} : \langle \mathcal{C}, \otimes, \mathbb{1} \rangle \to \langle \mathcal{C}, \otimes, \mathbb{1} \rangle$ is a lax monoidal endofunctor with natural transformation $p : Id \Rightarrow \mathcal{K}$, such that forall $A, B \in Ob(\mathcal{C})$ the next diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle p}\downarrow & & \downarrow{\scriptstyle p} \\ \mathcal{K}A & \xrightarrow[p_f]{} & \mathcal{K}B \end{array}$$

i.e. $p_f \circ p = p \circ f$.

By default we will consider applicative functor $\mathcal{K}$ on some CCC $\mathcal{C}$, that's not hard to check, that finite product and terminal object produce some (symmetric) monoidal structure for an arbitrary CCC $\mathcal{C}$.

Soundness. Semantic translation for typing rules.

Theorem

Soundness

If $M =_{\beta\eta} N$, *then* $[\![M]\!] = [\![N]\!]$

Let us remember the semantic translation from simply typed lambda calculus to CCC:

1) Translation for types: $[\![A]\!] := \hat{A}, A \in \mathbb{T}$, $[\![A \to B]\!] := [\![A]\!] \to [\![B]\!]$, $[\![A \times B]\!] := [\![A]\!] \times [\![B]\!]$;.

2) Translation for contexts: $[\![\Gamma = \{x_1 : A_1, ..., x_n : A_n\}]\!] := [\![\Gamma]\!] = [\![A_1]\!] \times ... \times [\![A_n]\!]$;

3) $[\![\Gamma \vdash M : A]\!] := [\![M]\!] : [\![\Gamma]\!] \to [\![A]\!]$;

4) Translation rules for typing rules with $\to$ and $\times$ are definined with exponential and products (slide is too small for them);

5) Interpretation for modal types: $[\![\mathbf{K}A]\!] = \mathcal{K}[\![A]\!]$, where $\mathcal{K}$ is an applicative functor.

5) Check substitution in a model.

Soundness. Semantic translation for typing rules.

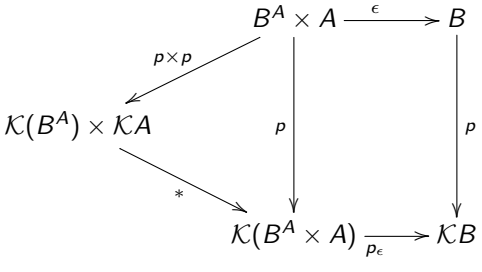
Semantic translation for $\mathbf{K}_{app}$ and $\mathbf{K}_{I}$:

$$\frac{[\![\Gamma \vdash M : A]\!] := [\![M]\!] : [\![\Gamma]\!] \to [\![A]\!]}{[\![\Gamma \vdash \texttt{pure}\ \ M : \mathbf{K}A]\!] := [\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![A]\!] \xrightarrow{p_A} \mathcal{K}[\![A]\!]}$$

$$\frac{[\![\Gamma \vdash M : \mathbf{K}(A \to B)]\!] := [\![M]\!] : [\![\Gamma]\!] \to \mathcal{K}([\![B]\!]^{[\![A]\!]}) \qquad [\![\Gamma \vdash N : \mathbf{K}A]\!] := [\![N]\!] : [\![\Gamma]\!] \to \mathcal{K}[\![A]\!]}{[\![\Gamma \vdash M\ \texttt{<*>}\ N : \mathbf{K}B]\!] := [\![\Gamma]\!] \xrightarrow{p_\epsilon \circ * \circ \langle [\![M]\!], [\![N]\!] \rangle} \mathcal{K}B}$$

Soundness. Verifying of the reduction rule in categorical model.

$[\![(\texttt{pure}\ M)\ \texttt{<*>}\ (\texttt{pure}\ N)]\!] = [\![\texttt{pure}\ (MN)]\!]$



It is easy to see, that
$[\![(\texttt{pure}\ M)\ \texttt{<*>}\ (\texttt{pure}\ N)]\!] = p_\epsilon \circ * \circ \langle [\![\texttt{pure}\ M]\!], [\![\texttt{pure}\ N]\!] \rangle = p_\epsilon \circ * \circ \langle p \circ [\![M]\!], p \circ [\![N]\!] \rangle =$
$p_\epsilon \circ * \circ (p \times p) \circ \langle [\![M]\!], [\![N]\!] \rangle = p_\epsilon \circ (* \circ (p \times p)) \circ \langle [\![M]\!], [\![N]\!] \rangle = p_\epsilon \circ p \circ \langle [\![M]\!], [\![N]\!] \rangle =$
$(p_\epsilon \circ p) \circ \langle [\![M]\!], [\![N]\!] \rangle = p \circ \epsilon \circ \langle [\![M]\!], [\![N]\!] \rangle = [\![\texttt{pure}\ (MN)]\!]$.

Compeleteness.

Not proved yet.

Further research.

Monad class in modern Haskell (or Idris) is obtained by inheritance from Applicative class, in other words:

```
class Applicative m => Monad (m :: * -> *) where
(>>=) :: m a -> (a -> m b) -> m b
return :: a -> m a
```

Further research.

From a logical point of view, we have some system with the following modal axioms:
Some system $= IEL^- + \mathbf{K}\mathbf{K}A \to \mathbf{K}A$ or Some system $= IEL^- + \mathbf{K}A \to ((A \to \mathbf{K}B) \to \mathbf{K}B)$.

We may extend our type system by adding some rule for the third axiom and consider type system for monadic calculations based on some applicative functor (examples of possible rules):

$$\frac{\Gamma \vdash M : \mathbf{K}\mathbf{K}A}{\Gamma \vdash \texttt{join}\ M : \mathbf{K}A}$$

$$\frac{\Gamma \vdash M : \mathbf{K}A \qquad \Gamma, x : A \vdash N : \mathbf{K}B}{\Gamma \vdash \texttt{let}\ M = x\ \texttt{in}\ N : \mathbf{K}B}$$

Hypothesis 1: Moggi's metalanguage should be embedded in our system with `join`-rule.
Hypothesis 2: Our system with monadic binding rule is a non-conservative extension of Moggi's metalanguage.
Hypothesis 3: extension of this kind should be sound and complete for some complication of monoidal monad (for example, applicative functor with natural transformation $\eta : \mathcal{K}^2 \to \mathcal{K}$).

Thank you!