1 Motivating example

I will begin with an example from functional programming that will help to motivate the ideas behind my approach. Functional programming shares a lot of similarities with the formal work a semanticist has to do.

The key principle of functional programming is purity: the results of functions depend exclusively on their arguments, they cannot effect or be effected by any other elements. In formal semantics, the analogous principle is that of compositionality: the meaning of a phrase is a function of the meanings of its constituents (and of the way those constituents are combined).

We can rephrase purity in terms of compositionality and vice versa. Pure procedures are procedures such that the meaning of a procedure call (its result) is a function of the meaning of the procedure (its definition) and of the argument (its value). A syntactic operator has a compositional semantics if its meaning can be derived as a pure function of its constituents meanings.

However, both in the case of functional programming and formal semantics, we run into problems when we need to handle phenomena which are "seemingly non-compositional". An example of this in formal semantics is the treatment of context and the capability of phrases to both depend on it and modify it.

1.1 The example

We consider the problem of having to write a function which (re)labels the nodes of a binary tree with natural numbers in an increasing order depth-first left-to-right.

Let us start first with a straightforward solution (`relabel0`). We will transform the tree recursively. Since in order to relabel some subtree, we need to know the number at which to start indexing, the recursive function needs an extra argument. We also need to find the amount by which the index was incremented while transforming the subtree so that we know at which number to continue when we process the next subtree. We solve this by having the recursive function return a pair of the transformed tree and the new index value.

```
relabel0 :: Tree a -> Tree Int
relabel0 tree = fst (relabel' tree 0) where

  relabel' :: Tree a -> Int -> (Tree Int, Int)
  relabel' (Leaf value)                n = (Leaf n, n + 1)
  relabel' (Branch left value right) n =
    let (newLeft, newValue) = relabel' left n
        (newRight, newN)    = relabel' right (newValue + 1) in
    (Branch newLeft newValue newRight, newN)
```

We can refactor our solution a bit into `relabel1`. This makes the passing around of the integer state more systematic. We see that the `relabel'` function performs a sequence of steps. The sequentiality is expressed by the fact that we use some state of our computation (the current index) as a parameter at each step and we produce some value and the new state.

```
relabel1 :: Tree a -> Tree Int
relabel1 tree = fst (relabel' tree 0) where

  relabel' :: Tree a -> Int -> (Tree Int, Int)
  relabel' (Leaf value)              n0 =
    let (newValue, n1) = (n0, n0 + 1) in
        (Leaf newValue, n1)
  relabel' (Branch left value right) n0 =
    let (newLeft,  n1) = relabel' left n0
        (newValue, n2) = (n1, n1 + 1)
        (newRight, n3) = relabel' right n2 in
    (Branch newLeft newValue newRight, n3)
```

Now that we are passing around the state systematically, we can define a function to do the tedium for us. In this case, we are actually asking for the `bind` method (`>>=`) of the state monad. We can therefore abstract away from passing around the state manually and use Haskell's handy `do`-notation, which is syntactic sugar for the `bind` method of a monad (`relabel2`).

```
relabel2 :: Tree a -> Tree Int
relabel2 tree = evalState (relabel' tree) 0 where

  relabel' :: Tree a -> State Int (Tree Int)
  relabel' (Leaf value)              =
    do newValue <- relabelNode value
       return (Leaf newValue)
  relabel' (Branch left value right) =
    do newLeft  <- relabel' left
       newValue <- relabelNode value
       newRight <- relabel' right
       return (Branch newLeft newValue newRight)

  relabelNode :: a -> State Int Int
  relabelNode old = do n <- get
                       put (n + 1)
                       return n
```

The type constructor `State Int a` represents a stateful computation of type `a` where the state being accessed is of type `Int`. It is merely an abstraction for `Int -> (a, Int)`. The signature of `relabel'` is now clearer. It tells us that `relabel'` is a function that maps some tree into an integer tree while accessing some state of type `Int`. The final function, `relabel2`, however, is not stateful because we apply `evalState` to the stateful computation and supply some initial value for the state. `evalState` just passes the initial state to the stateful computation and then retrieves the result of the computation, i.e. the first element of the returned pair. The body of the function is pretty much the same, only `let` and `=` have been replaced by `do` and `<-`, respectively. This frees us from having to manage state manually ourselves.

We also extract the relabeling transformation that is applied to the values in the tree themselves into the function `relabelNode`. Inside its definition, we use the “effectful” operations `get` and `put`, whose types are `State a a` and `a -> State a ()` respectively, to access and modify the state.

If we look closer at the function definition, it becomes easy now to see that `relabel'` is just a recursive traversal of the tree which applies the function `relabelNode` to each value in the tree. Furthermore, it is a generalization of the common `map` function since it accepts functions that map the values of the tree to values in some monad. By instantiating the monad to the identity monad, we get the familiar `map` function. By instantiating it to the state monad, we get a depth-first left-to-right traversal procedure.

We can pull out the traversal logic into its separate function. As it happens, this kind of function can be derived automatically by Haskell just from the data type's declaration. Our relabeling function then ends up looking like this (`relabel3`):

```
relabel3 :: Tree a -> Tree Int
relabel3 tree = evalState (traverse relabelNode tree) 0 where

  relabelNode :: a -> State Int Int
  relabelNode old = do n <- get
                       put (n + 1)
                       return n
```

We finally end up with something which looks very simple, we simply traverse the tree while applying a function. Inside the function, there is an introduction of state because of the use of the `get` and `put` operations. At the top, we then have the corresponding elimination of state by the `evalState` function (of type `State a b -> a -> b`). Furthermore, all of this is done in a pure setting. Haskell provides us with merely a thin layer of syntactic sugar over the lambda calculus, even though it feels like we are modifying some state variables as in more expressive languages.

This example has relevance to the plight of a formal semanticist trying to handle discourse dynamics compositionally. The tree transformation that we set out to describe here is interesting because the translation of a subtree is not simply a function of the subtree, but it also depends on some context that was established before, context which is also updated by the process of translating the subtree. This is very similar to the basic problem we encounter in dynamic semantics. The meaning of some syntactic phrase is expressed not only in terms of its constituents' meanings but also in terms of some context which it can update as well.

We used the pure/compositional formalism of Haskell to describe the transformation. This has led us to a solution in a “denotational” style. We have then presented a framework commonly adopted in Haskell, the notion of a monad, which allowed us to reach a more succinct and intelligible solution. This new solution was expressed in “operational” terms. We had a procedure with virtual side-effects called `relabelNode` which was mapped over the tree. In the end, we had been using a pure programming language but the most elegant solution to our problem had proven to be the implementation of state and then expressing the transformation as an operation with effect on the state.

In this way, we got to work in a pure metalanguage which is amenable to denotational reasoning and analysis, but we still had the possibility of working in an operational style if it suited our problem. Another approach similar to this might be the introduction of a language which is effectful, but has both an operational and a denotational semantics.

2 Dynamic logic as effectful computation

We will demonstrate our idea of adopting effects in computational semantics by rephrasing the continuation-based dynamic logic of Philippe DeGroote in effect-like terms.

2.1 Our monad

We define our monad to be the combination of state and continuations that was used by Oleg Kiselyov in his Applicative ACG examples. The type $[\![\alpha]\!]_\gamma^\omega$ shall be synonymous with $\gamma \to (\alpha \to \gamma \to \omega) \to \omega$ and stands for a computation of a value of type α that manipulates state of type γ and has access to its continuation whose result type is ω. The continuation expects both the computed value of type α and the new state of type γ. This differs from, and generalizes, the approach used in DeGroote, where both α and ω are fixed to the type o of propositions and where the continuation only produces the future proposition and does not need access to the current proposition (the extra argument of type α in our formulation).

We give below the definitions of `return` and `>>=` for this monad.

$$\begin{aligned}
&[\![\alpha]\!]_\gamma^\omega = \gamma \to (\alpha \to \gamma \to \omega) \to \omega \\
&\texttt{return} :: \alpha \to [\![\alpha]\!]_\gamma^\omega \\
&\texttt{return}\ x \equiv \lambda e\phi.\phi x e \\
&(\texttt{>>=}) :: [\![\alpha]\!]_\gamma^\omega \to (\alpha \to [\![\beta]\!]_\gamma^\omega) \to [\![\beta]\!]_\gamma^\omega \\
&m \texttt{ >>= } f \equiv \lambda e\phi.me(\lambda v e'.f v e' \phi)
\end{aligned}$$

We also give the definition of a state "elimination" procedure. `runState` takes as arguments some initial state e of type γ and a value m in our monad, having type $[\![\alpha]\!]_\gamma^\alpha$, and converts the "dynamic" value (stateful computation) into a pure value of type α by providing it with the initial state and a trivial terminal continuation.

$$\begin{aligned}
&\texttt{runState} :: \gamma \to [\![\alpha]\!]_\gamma^\alpha \to \alpha \\
&\texttt{runState}\ e\ m \equiv me(\lambda x e'.x)
\end{aligned}$$

2.2 Logical connectives

When using a logic, we prefer to write in terms of logical connectives. We will now proceed to define conjunction, negation and existential quantification on dynamic formulas (stateful computations of propositions, type $[\![o]\!]^o_\gamma$).[1]

2.2.1 Conjunction

Conjunction is the most straight-forward. We order the effects of the two formulas ϕ and ψ in conjunction $\phi \mathbin{\bar{\wedge}} \psi$ such that the effects of ϕ happen before the effects of ψ by first binding ϕ and then ψ.

$$\begin{aligned}(\bar{\wedge}) &:: [\![o]\!]^o_\gamma \to [\![o]\!]^o_\gamma \to [\![o]\!]^o_\gamma \\ \phi \mathbin{\bar{\wedge}} \psi &\equiv \texttt{do}\ p \leftarrow \phi \\ &\qquad\ \ q \leftarrow \psi \\ &\qquad\ \ \texttt{return}\ (p \wedge q)\end{aligned}$$

This syntactic sugar is translated into the following lambda calculus:

$$\phi \mathbin{\bar{\wedge}} \psi \equiv \phi \texttt{ >>= } (\lambda p.\psi \texttt{ >>= } (\lambda q.\texttt{return}\ (p \wedge q)))$$

There is also a combinator, `liftM2`, commonly used in Haskell which fits our definition of dynamic conjunction. `liftM2` lifts a binary function over simple values into a generalized binary function which operates on values in some monad and which evaluates its arguments left-to-right.

$$\bar{\wedge} \equiv \texttt{liftM2}\ \wedge$$

This states more clearly our intent that dynamic conjunction is just plain conjunction which evaluates its two arguments for effects on discourse state in a left-to-right order.

2.2.2 Negation

If we follow the definition given in DeGroote's proposal, we arrive at the following restatement in our monadic framework.

$$\begin{aligned}(\bar{\neg}) &:: [\![o]\!]^o_\gamma \to [\![o]\!]^o_\gamma \\ \bar{\neg}\,\psi &\equiv \lambda e\phi.\phi(\neg(\psi e(\lambda p e'.p)))e\end{aligned}$$

We notice that inside the negation, we are applying a dynamic proposition to some initial state and the trivial terminal continuation, which was exactly the definition of our state eliminator, `runState`.

$$\bar{\neg}\,\psi \equiv \lambda e\phi.\phi(\neg(\texttt{runState}\ e\ \psi))e$$

[1]The other logical connectives are to be derived from these three using the usual equations.

Here we see that dynamic negation can be seen as recursively interpreting the negated content in a fresh instance of the state interpreter, all effects of ψ will be contained inside the negation. In our case of using our state monad, this reinterpretation can be more easily understood as dynamic (re)binding. Wrapping `runState` over a stateful computation is like dynamically binding a new value to the state variable. Using `runState` in our extended negation thus means that the state that is modified by ψ is different from the global state. Since we initialize this new state to the same value as the global state, this can be seen as saying that the negated content only has access to a copy of the state, changes to which will be discarded.

We can think of `runState` as of a handler which interprets the effects in the negated formula ψ locally. Note that access to state is not the only effect supported by our monad, we also give our computations access to their continuations. This means that `runState` not only binds a fresh state variable but it also delimits the continuations inside ψ.

Before we finish with negation, we will present another way of restating it. The previous definition was given in a continuation-passing style. We could not just use `return` to raise the negated formula into a dynamic formula since we were accessing the current state e which we used as our new initial state. We can define an effectful operation (a function, 0-ary in this case, whose output values belong to the monad) that retrieves the state for us.

$$\begin{aligned}&\texttt{get} :: [\![\gamma]\!]^{\omega}_{\gamma}\\&\texttt{get} \equiv \lambda e\phi.\phi e e\end{aligned}$$

Having introduced `get`, we can now reformulate negation using `do`-notation and not having to abstract manually over the state e and the continuation ϕ.

$$\begin{aligned}\bar{\neg}\,\psi \equiv \texttt{do}\ &e \leftarrow \texttt{get}\\&\texttt{return}\ \neg(\texttt{runState}\ e\ \psi)\end{aligned}$$

As a final remark, we notice that wrapping a dynamic formula in `runState` has a similar effect as wrapping it in a DRT box. The argument of `runState` gets a fresh copy of the state which it can use as a sandbox for introducing local state changes which will not escape the negation.

2.2.3 Existential quantification

For the existential quantifier, we start with the definition from DeGroote's proposal, since it is compatible with our formalization.

$$\begin{aligned}&(\bar{\exists}) :: (\iota \to [\![o]\!]^{o}_{\gamma}) \to [\![o]\!]^{o}_{\gamma}\\&\bar{\exists}\,P \equiv \lambda e\phi.\exists(\lambda x.Pxe\phi)\end{aligned}$$

Here we witness for the first time the body is not an application of the continuation to some return value. Instead, we are quantifying over a term built using the continuation.

We can decompose this term into smaller parts. First, we notice that the type of the existential quantifier is $(\iota \to o) \to o$. We note that values having the type $(\alpha \to \omega) \to \omega$, which we will write as $[\![\alpha]\!]^{\omega}$, form a monad of their own. This monad, dubbed the continuation monad, represents computations that have access to their continuations of some result type ω.

However, we operate with a notion of a computation that has access both to its continuation and to some mutable state. We will define a lifting operation which takes a value from the continuation monad having type $[\![\alpha]\!]^{\omega} = (\alpha \to \omega) \to \omega$ and injects it into the more general state-and-continuation monad, producing a value of type $[\![\alpha]\!]^{\omega}_{\gamma} = \gamma \to (\alpha \to \gamma \to \omega) \to \omega$ for some γ.

$$\begin{aligned} \texttt{cont} &:: [\![\alpha]\!]^{\omega} \to [\![\alpha]\!]^{\omega}_{\gamma} \\ \texttt{cont}\ c &\equiv \lambda e\phi.c(\lambda x.\phi x e) \end{aligned}$$

If we are reasoning within a calculus with continuations, we can think of the existential quantifier as a computation that produces some existentially quantified variable. By lifting the existential quantifier using the function defined above, we can therefore introduce a new effectful operation, `fresh`.

$$\begin{aligned} \texttt{fresh} &:: [\![\iota]\!]^{o}_{\gamma} \\ \texttt{fresh} &\equiv \texttt{cont}\ \exists \end{aligned}$$

By making the existential quantifier into a function whose value belongs to our monad, we can think of it as an effectful operation which generates fresh, existentially quantified discourse referents. We can then use this operation to express the dynamic existential quantifier.

$$\begin{aligned} (\bar{\exists}) &:: (\iota \to [\![o]\!]^{o}_{\gamma}) \to [\![o]\!]^{o}_{\gamma} \\ \bar{\exists}\, P &\equiv \texttt{do}\ x \leftarrow \texttt{fresh} \\ &\qquad\quad Px \end{aligned}$$

This can be rewritten without the `do`-notation sugar...

$$\bar{\exists}\, P \equiv \texttt{fresh} \ \texttt{>>=}\ P$$

...or in a more applicative style by flipping the direction of the bind.

$$\bar{\exists}\, P \equiv P \ \texttt{=<<}\ \texttt{fresh}$$

Thus we can see that saying $\bar{\exists}\, P$ is the same as stating P about some new existentially-quantified entity. The reason we need to turn to the effect of scoping over our continuation is that we would like to capture not only P in the scope of the quantifier, but also anything else that comes after. This way, the predicate does not check only P, but also everything else that is stated about the introduced individual in later discourse.

2.3 Translating the lexical entries

Now we can turn our attention to adapting the grammar itself.

The lexical entries that do not make use of the state and continuations will be very similar. Minor changes are made to fit our slightly varied notion of a continuation.

$$\begin{aligned}
[\![farmer]\!] &\equiv \lambda x.\texttt{return}\ (\mathbf{farmer}\ x)\\
[\![donkey]\!] &\equiv \lambda x.\texttt{return}\ (\mathbf{donkey}\ x)\\
[\![owns]\!] &\equiv \lambda os.s(\lambda x.o(\lambda y.\texttt{return}\ (\mathbf{own}\ x\ y)))\\
[\![beats]\!] &\equiv \lambda os.s(\lambda x.o(\lambda y.\texttt{return}\ (\mathbf{beat}\ x\ y)))\\
[\![who]\!] &\equiv \lambda rnx.nx \mathbin{\bar{\wedge}} r(\lambda\psi.\psi x)
\end{aligned}$$

Now for the interesting parts.

$$\begin{aligned}
[\![a]\!] &\equiv \lambda n\psi.\,\bar{\exists}x.nx \mathbin{\bar{\wedge}} (\lambda e\phi.\psi x(x :: e)\phi)\\
[\![every]\!] &\equiv \lambda n\psi.\,\bar{\forall}x.nx \mathbin{\bar{\rightarrow}} (\lambda e\phi.\psi x(x :: e)\phi)\\
[\![it]\!] &\equiv \lambda\psi e\phi.\psi(\texttt{sel}_{\texttt{it}}e)e\phi
\end{aligned}$$

Here we have the parts that interact with the state being passed through the computation. As before, we might try to hide the abstractions over the states e and continuations ϕ and replace them with some more abstract operations.

$$\begin{aligned}
&\texttt{push} :: \iota \to [\![()]\!]^{\omega}_{\gamma}\\
&\texttt{push}\ x \equiv \lambda e\phi.\phi()(x :: e)^2
\end{aligned}$$

push is just a stateful operations that adds a new discourse referent on to the list of current discourse referents in the state. We can now rewrite the recipes for *a* and *every*.

$$\begin{aligned}
[\![a]\!] &\equiv \lambda n\psi.\,\bar{\exists}x.nx \mathbin{\bar{\wedge}} (\texttt{do push}\ x; \psi x)\\
[\![every]\!] &\equiv \lambda n\psi.\,\bar{\forall}x.nx \mathbin{\bar{\rightarrow}} (\texttt{do push}\ x; \psi x)
\end{aligned}$$

We can pull out the syntactic sugar of the do-notation and reach the following ($A \mathbin{\texttt{>>}} B$ is defined as $A \mathbin{\texttt{>>=}} (\lambda\_.B)$).

$$\begin{aligned}
[\![a]\!] &\equiv \lambda n\psi.\,\bar{\exists}x.nx \mathbin{\bar{\wedge}} (\texttt{push}\ x \mathbin{\texttt{>>}} \psi x)\\
[\![every]\!] &\equiv \lambda n\psi.\,\bar{\forall}x.nx \mathbin{\bar{\rightarrow}} (\texttt{push}\ x \mathbin{\texttt{>>}} \psi x)
\end{aligned}$$

We can do the same for *it* by wrapping the selection operator in an effectful operation.

$$\begin{aligned}&\texttt{select}_{\texttt{it}} :: [\![\iota]\!]^{\omega}_{\gamma}\\&\texttt{select}_{\texttt{it}} \equiv \lambda e\phi.\phi(\texttt{sel}_{\texttt{it}}e)e\end{aligned}$$

With this in hand, we can now rewrite the recipe for *it* by using effects instead of direct manipulation of continuations.

$$\begin{aligned}[\![it]\!] \equiv \lambda\psi.\texttt{do}\ & x \leftarrow \texttt{select}_{\texttt{it}}\\ & \psi x\end{aligned}$$

This arguably reads nicer with the syntactic sugar removed...

$$[\![it]\!] \equiv \lambda\psi.\texttt{select}_{\texttt{it}} \ \texttt{>>=}\ \psi$$

...and in an applicative order.

$$[\![it]\!] \equiv \lambda\psi.\psi \ \texttt{=<<}\ \texttt{select}_{\texttt{it}}$$

The applicative style betrays the similarity of the denotation of *it* to the denotations of other referential expressions, such as proper names.

$$\begin{aligned}[\![John]\!] &= \lambda\psi.\psi\ \texttt{j}\\ &= \lambda\psi.\psi \ \texttt{=<<}\ \texttt{return}\ \texttt{j}\end{aligned}$$

Here we have just replaced the $\texttt{select}_{\texttt{it}}$ operation by the trivial computation `return j`.

2.4 Recapitulation

To sum up what we have done, we have introduced a monad which is capable of treating computations that have access to their continuations and to some mutable state.

In terms of this monad, we have then redefined three of the basic predicate logic connectives. Conjunction was trivially lifted such that it chained the effects from left to right. Negation introduced a sort of modality which acts as a containing island for any state and continuation effects. The existential quantifier profited from the presence of continuations to extend its scope to anything that will be yielded by the continuing computation.

Finally, we have restated the donkey grammar fragment using this formalism. In the final grammar, we can distinguish three kinds of lexical items.

We can see the universal/neutral items which do not use any of the introduced effects, such as the common nouns, the verbs and the relativizer. These items would work just as well in other grammars by taking their definitions of the logical connectors and their monad, e.g. in a grammar about intensionality (by using the world reader monad), about event arguments (by using the event reader monad) or just in a basic toy grammar (by using the identity monad).

Next, we have the decorated items. These are lexical items that are not at the focal point of our grammar but which interact with the phenomenon it models, or at least its formalization. In our example, the items in focus are anaphoric expressions such as pronouns. These interact with quantified noun phrases, even though a satisfying account of quantified noun phrases can be given while omitting pronouns. The structural difference between the treatment of these items in a simple grammar and an expanded one is usually just a single adjunction, e.g. adjoining ($\texttt{push}\ x \gg *$) in the case of the quantified noun phrases. It is usually desirable to avoid the ad-hoc sprinkling of such effects in a grammar by folding them inside the basic logical connectives to arrive at a more principled solution. In our running example, this could be achieved by removing the push applications from the lexicon and using the following definition of dynamic quantification which automatically pushes the new discourse referent (as was done by Lebedeva).

$$\bar{\exists} P \equiv \texttt{fresh} \gg= (\lambda x.\texttt{push}\ x \gg P x)$$

Finally, we have the focal items which could not be expressed without the mechanisms that were introduced. In the case of dynamicity, these are the anaphoric expressions, which cannot be reconciled with the basic stateless view of semantics.

3 From monads to effects

The point of introducing monads and using them to present dynamic logic was not to convince others to adopt monads in their work. That would be a moot point since there is hardly any difference between the original proposal and our monadic revision and preferring one over the other is merely a question of style.

On the contrary, the reason for our exposition was to convince the reader that the continuation-based style of dynamic logic already is a monad, or at least something akin to one in the relevant aspects. What we talk about is the fact that meanings are generalized from their plain form (e.g. a truth value) into the more expressive form (e.g. a function from left and right contexts into a truth value) that allows us to describe some novel phenomenon.

The problem with this approach is that as we try to account for multiple phenomena in a single theory, we end up with a layered structure of these monadic generalizations. This layering manifests itself in several inconveniences. Consider the case of combining intensionality and dynamicity. Applying an intensional operator to a dynamic intension necessitates that we lift the operator to the dynamic domain so that it can pass through the dynamic layer, operate on the intension and return another dynamic intension. A similar problem arises when applying the same (static intensional) operator to the intension of a dynamic meaning. We will demonstrate this with an example.

3.0.1 Example of lifting operators

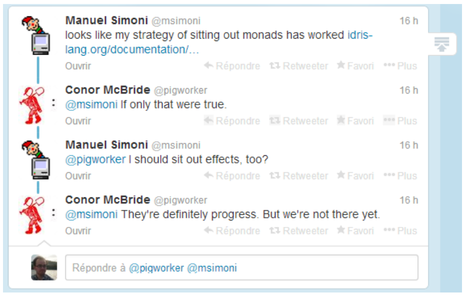
Imagine we have an intensional grammar fragment. Denotations of linguistic expressions have types which are the result of applying the intensionalization operation described in de Groote and Kanazawa [2013] to the usual types. To present the notion of an intensional type, we introduce the notation $\overline{\alpha}$ which will stand for the type $\sigma \to \alpha$, where σ is the type of possible worlds. The type of an intensional denotation is then derived from the type of an extensional denotation using the homomorphism given below.

$$\begin{aligned} Int(\iota) &= \overline{\iota} \\ Int(o) &= \overline{o} \\ Int(\alpha \to \beta) &= Int(\alpha) \to Int(\beta) \end{aligned}$$

See the example below with the intensional operator $fake$ being lifted to $fake'$ so that it can be applied to dynamic intensions such as $student'$ and to $fake''$ so that it is applicable to intensions of dynamic meanings such as $student''$.

$$\begin{aligned} \overline{\alpha} &\equiv \sigma \to \alpha \\ [\![student]\!] &:: \overline{\iota} \to \overline{o} \\ [\![fake]\!] &:: (\overline{\iota} \to \overline{o}) \to (\overline{\iota} \to \overline{o}) \\ [\![student']\!] &:: \overline{\iota} \to [\![\overline{o}]\!]_{\gamma}^{\overline{o}} \\ [\![fake']\!] &:: (\overline{\iota} \to [\![\overline{o}]\!]_{\gamma}^{\overline{o}}) \to (\overline{\iota} \to [\![\overline{o}]\!]_{\gamma}^{\overline{o}}) \\ [\![fake']\!] &\equiv \lambda Pxe\phi.\phi([\![fake]\!](\lambda y.Pye(\lambda\psi e'.\psi))x)e \\ [\![student'']\!] &:: \overline{\iota} \to \overline{[\![o]\!]_{\gamma}^{o}} \\ [\![fake'']\!] &:: (\overline{\iota} \to \overline{[\![o]\!]_{\gamma}^{o}}) \to (\overline{\iota} \to \overline{[\![o]\!]_{\gamma}^{o}}) \\ [\![fake'']\!] &\equiv \lambda Pxse\phi.\phi([\![fake]\!](\lambda yt.Pyte(\lambda\psi e'.\psi))xs)e \end{aligned}$$

$$\begin{aligned} Dyn(\iota) &= \iota \\ Dyn(o) &= [\![o]\!]_{\gamma}^{o} = \gamma \to (o \to \gamma \to o) \to o \\ Dyn(\alpha \to \beta) &= Dyn(\alpha) \to Dyn(\beta) \\ [\![fake']\!] &:: Int(Dyn((\iota \to o) \to (\iota \to o))) \\ [\![fake'']\!] &:: Dyn(Int((\iota \to o) \to (\iota \to o))) \end{aligned}$$



Manuel Simoni @msimoni 16 h
looks like my strategy of sitting out monads has worked idris-lang.org/documentation/...

Conor McBride @pigworker 16 h
@msimoni If only that were true.

Manuel Simoni @msimoni 16 h
@pigworker I should sit out effects, too?

Conor McBride @pigworker 16 h
@msimoni They're definitely progress. But we're not there yet.

Figure 1: Anecdotal evidence of the transition from monads to effects.

References

Philippe de Groote and Makoto Kanazawa. A note on intensionalization. *Journal of Logic, Language and Information*, pages 1–22, 2013.