Implementing Continuation Semantics with Monadic Effects

Joshua Gancher

May 19, 2018

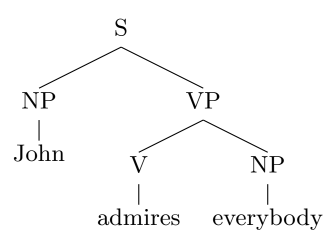
Abstract

We present a Haskell library for computing semantic derivations using a monadic framework of continuations, with support for additional monadic effects.

1 Overview of Theory

Our starting-off point is the semantic framework of *monadic continuations*, as described by Charlow [1]. In this framework, we do not work with base types (typically $\mathsf{e}$and t) directly, but instead work with *computations returning base types*. Our computations will be of the form $(\sigma \to N\ \mathsf{t}) \to N\ \mathsf{t}$, where N is a monad which may encode various side-effects. This type will be abbreviated $M\sigma$. It turns out that M has the structure of a *monad transformer*, so is itself a monad.

In our library, verbs are functions from computations to computations. Thus, in the following situation:



"John" and "everybody" *both* have semantic type $M\mathsf{e}$, and "admires" has semantic type $M\mathsf{e} \to M\mathsf{e} \to M\mathsf{e}$. By arranging our types this way, we may compute semantic values just as before, by using functional application. All non-local phenomena is handled within the monad M.

Our semantic framework is parameterized by the choice of inner monad N. Recall that any monad N supports the following operations:

- return: $\forall\alpha.\ \alpha \to N\alpha$
- bind: $\forall\alpha\beta.\ N\alpha \to (\alpha \to N\beta) \to N\beta$.

An example inner monad would be the *Reader* monad, which models intensionality. The Reader monad maps a type σ to the type $\mathsf{s} \to \sigma$. The return function for the monad, on input x, returns the constant function $\lambda w.\ x$. The bind function, on input c and f, returns the function $\lambda w.\ (f\ (c\ w))\ w$; that is, it applies the world variable to c, applies the result to f, and applies the world variable again to f's result. Note that the reader monad induces a nontrivial operation, get of type $\mathsf{s} \to \mathsf{s}$, which returns the current world variable.

Given an inner monad, we derive a monad structure on M, the outer monad, defined to be:

- $\mathsf{return}\ x\ k := kx$ and
- $\mathsf{bind}\ c\ f\ k := c\ (\lambda x.\ (f\ x)k)$.

That is, return takes a value and passes it to the continuation, while bind runs the first computation with the continuation that, on input x, feeds x into f and runs that computation on the present continuation k. Since M is a monad transformer, it also admits a *lifting* operator $\cdot^\sharp$, which turns a computation in the inner monad, N, to the outer monad, M. This is done by use of the bind operation of the underlying monad. Thus, when N is the reader monad, we receive the operation $\mathsf{get}^\sharp$ of type $M\mathsf{s}$.

Given the above setup, we can give a semantic interpretation to the above sentence, using the Reader monad:

$$
\begin{aligned}
[\![\text{John}]\!] &:= \mathsf{bind}_M\ \mathsf{get}^\sharp(\lambda w.\ \mathsf{return}\ (\mathbf{j}\ w)) \\
[\![\text{everybody}]\!] &:= \lambda k.\ \mathsf{bind}_N\ \mathsf{get}(\lambda w.\ \forall x.\ \mathbf{person}\ w\ x \implies (k\ x)w)) \\
[\![\text{admires}]\!] &:= \lambda c_1 c_2.\ \mathsf{bind}_M\ \mathsf{get}^\sharp(\lambda w.\ \mathsf{bind}_M\ c_1(\lambda x.\ \mathsf{bind}_M\ c_2\ (\lambda y.\ \mathsf{return}\ (\mathbf{admires}\ w\ x\ y))))
\end{aligned}
$$

Above, $[\![\text{John}]\!]$ receives the current world, w, and returns $\mathbf{j}\ w$ to the current continuation. Thus, the semantics of names can be world dependent. The semantic value $[\![\text{everybody}]\!]$ takes in the current continuation explicitly as well as the current world variable w, and first applies x to k. This results in a value of type $N\mathsf{t}$, which means it is waiting for a world variable. Thus, we apply w to the result. Furthermore, we may restrict the domain of quanification using w, to require that x is a person in the current world. Finally, $[\![\text{admires}]\!]$ takes in the current world w, binds the two argument continuations, and returns the result of applying the current world and the two results to **admires**.

Once we have the above three parts, we can now construct the semantics of the total sentence

$$S = [\![\text{John admires everybody}]\!] = [\![\text{admires}]\!]\ [\![\text{John}]\!]\ [\![\text{everybody}]\!]$$

which has type $M\mathsf{t} = (\mathsf{t} \to N\mathsf{t}) \to N\mathsf{t}$. At this point, we may apply return to S to receive a value of type $N\mathsf{t}$. This value we can then interpret as an ordinary lambda term, which will calculate to

$$[\![\text{John admires everybody}]\!] = \lambda w.\, \forall x.\, \mathbf{person}\, w\, x \implies \mathbf{admire}\, w\, (\mathbf{j}\, w)\, x.$$

In the next section, we outline the implementation of the computational system, which is able to automatically derivations such as the one above. The computational system is currently extended to use a monad

N which handles intensionality as well as discourse referents. (A refined implementation would use a system of algebraic effects for N, which would mean the innner monad can be completely parameterized throughout the implementation.)

2 Overview of Implementation

We begin with an embedded lambda calculus, given in higher order abstract syntax:

```
data Exp (tp :: Ty) where
    Var :: String -> TyRepr tp -> Exp tp
    Const :: String -> TyRepr tp -> Exp tp

    -- tuple constructors / destructors
    Tup :: Exp tp -> Exp tp2 -> Exp (tp ** tp2)
    PiL :: Exp (tp ** tp2) -> Exp tp
    PiR :: Exp (tp ** tp2) -> Exp tp2

    Lam :: KnownTy tp => (Exp tp -> Exp tp2) -> Exp (tp ->> tp2)
    App :: Exp (tp ->> tp2) -> Exp tp -> Exp tp2

    -- logical operators
    Forall :: KnownTy t => Exp (t ->> T) -> Exp T
    Exists :: KnownTy t => Exp (t ->> T) -> Exp T
    Not :: Exp T -> Exp T
    And :: Exp T -> Exp T -> Exp T
    Or :: Exp T -> Exp T -> Exp T
    Implies :: Exp T -> Exp T -> Exp T

    -- list constructors / destructors
    ListNil :: KnownTy t => Exp (List t)
    ListCons :: Exp t -> Exp (List t) -> Exp (List t)
```

Our entire computational system will be based on the above core language. Above, `Ty` comes from the grammar

$$\tau := \mathsf{s} \mid \mathsf{e} \mid \mathsf{t} \mid \tau ** \tau \mid \tau \text{ ->> } \tau \mid \mathsf{List}\ \tau,$$

where $\tau ** \tau$ is the type constructor for tuples, and τ ->> τ is the type constructor for functions.

Note above that we do *not* embed any monadic notions in our core language. Instead, we use the monadic constructs of Haskell directly. For instance, the Reader monad from Section 1 is modeled as `Reader (Exp T)`, where `Reader` is the Reader monad in Haskell. By doing so, we can implement the monad M from above as follows:

```
data MS = MS {
    _erefs :: [Exp E],
    _etrefs :: [Exp E -> M T] }

type M a = ContT (Exp T) (ReaderT (Exp S) (State MS)) (Exp a)
```

Here, the inner monad N is `ReaderT (Exp S) (State MS)`, and the outer monad is `ContT (Exp T) N`. This inner monad models three things:

1. Intensionality, using a reader monad,
2. Anaphora, using a state monad with the field `_erefs`, and
3. Higher order discourse referents, using the same state monad with the field `_etrefs`.

Anaphora and drefs are modeled by a stack of values, represented as a list. Higher order discourse referents can be used to model sentences such as "John wishes [himself] to be an actor, and so does Keisha [for herself]", which can be computed to have semantic value

$$\lambda w.\, (\forall v.\, \mathbf{want}\; w\; (\mathbf{j}\; w)\; v \implies \mathbf{actor}\; v\; (\mathbf{j}\; w)) \wedge (\forall v.\, \mathbf{want}\; w\; (\mathbf{k}\; w)\; v \implies \mathbf{actor}\; v\; (\mathbf{k}\; w)).$$

(TODO: should I change it so I get **actor** v ($\mathbf{j}$ v)?)

References

[1] Simon Charlow. On the semantics of exceptional scope. 2014.