Computational LCC

Towards DEMO *v. 2.0*

Matthew P. Wampler-Doty

1 Introduction

This document encompasses several ideas regarding the Logic of Communication and Change, and its computational implementation. We have rewritten model checking in DEMO essentially from scratch here, with an attempt to employ Haskell's class system as well as certain structural observations regarding modal logic and Haskell programming. Our work culminates with a computational implementation of intersection free PDL.

```
{-# OPTIONS_GHC -fglasgow-exts #-}

module LCC

where
import Data.Map (Map)
import qualified Data.Map as Map
```

Our presentation is broken up into 4 parts:

(*1*) Introduction

(*2*) Language

(*3*) Semantics

(*4*) Conclusion

2 Language

2.1 Introduction & Theory

The idea for the languages employed in this computational implementation of LCC is that languages extend one another in a hierarchical manner. The concept here is summarized in the following

slogan, which comes directly from [1]:

> *Modal languages are Boolean algebras with operators (BAOs), which have certain behaviors.*

Apart from this basic observation, we will try to suspend, as much as possible, every actual design choice as long as possible. This includes saying what predicate letters are, what modalities are, what PDL is, and even what the truth predicate is.

Instead, we give an assortment of classes, with the promise of later instantiating them. When we do program data structures for instantiation, we will try to be as polymorphic as possible. This is all done with the hope that

2.2 Classes for Languages

2.2.1 Boolean Algebras

We start by giving a basic class for Boolean algebras:

```
class BA form where
  infixr 5 \/
  top :: form                       -- Verum
  bot :: form                       -- Falsum
  neg ::  form -> form              -- Negation
  (\/) ::  form -> form -> form     -- Disjunction
```

There were of course many, many ways to accomplish the above definition; one way of note is that we could have defined a *ring* and then assumed that the user would enforce Boolean algebra laws where appropriate (as is done with *monads* in Haskell).

Here are some natural polymorphic logical connectives we might imagine after declaring this class:

```
infixr 5 /\, -->, <->, |^
(/\) ::  BA form => form -> form -> form     -- Conjunction
x /\ y = neg (neg x \/ neg y)

(-->) :: BA form => form -> form -> form     -- Implication
x --> y = (neg x \/ y)

(<->) :: BA form => form -> form -> form     -- Bi-implication
x <-> y = (x --> y) /\ (y --> x)

(|^) :: BA form => form -> form -> form      -- Scheffer Stroke
x |^ y = neg (x /\ y)
```

A small extension to `BA` is the introduction of named letters p, q, r, etc. which refer to particular predicates. We could probably do this outside of a class instance but this would break language independence.

```
class Preds x where
  a :: x; b :: x; c :: x; d :: x; e :: x; f :: x; g :: x;
  p :: x; q :: x; r :: x; s :: x; t :: x; u :: x; v :: x;
```

2.2.2 Modal Logic

We now turn to giving a class for (monadic) modal logic. Now, while basic modal logic has only one modality, we can think of plenty that have many more (for example, temporal logic, arrow logic, epistemic logic, and PDL). Furthermore, our system should support *ad-hoc* modalities, as one important activity for a modal logician is to experiment with extensions to a modal logic at hand in the hopes of finding reduction axioms. So our class for modal logic is maximally vague; it is a multi-parameter class on "modalities", and a Boolean algebra.

```
class BA form => ML modality form where
  nec :: modality -> form -> form
```

One important thing for modal logic is to have boxed and dual variants of every modality. Now, we may remark that if `K` is a modality, then `(nec K)` behaves, in Haskell syntax, just like a $\Box_K$ operation. The modal dual is naturally defined as follows:

```
pos :: (BA form, ML modality form) => modality -> form -> form
pos mod = neg . (nec mod) . neg
```

...and we can see that `(pos K)` behaves just like $\Diamond_K$.

2.2.3 PDL

We finish our presentation of languages with `PDL`, which is a modal logic with composable modalities. PDL is essentially a grammar on atomic relations with the following constructions: $\cup$, $\sim$ (dynamic negation), ?, ;, and $*$. They form a class that supervenes on `ML`:

```
class (ML modality form) => PDL modality form | modality -> form where
  infixr 4 -|-, -$-
  (-|-) :: modality -> modality -> modality        -- Union
  (-$-) :: modality -> modality -> modality        -- Composition
  test :: form -> modality                         -- Test
  kl :: modality -> modality                       -- Kleene Star
  nl :: modality                                   -- The Empty Program
```

We made a slightly unusual choice in incorporating the empty program, denoted $\varnothing$, into our class specification. We will return to this choice in our computational implementation of PDL in §3.4.3. It could just as easily be left out, since $\varnothing = ?\bot$.

We now turn to some naturally definable modalities. It should be noted that dynamic negation ($\sim$) can in fact be defined in terms of test (?), since $\sim\pi = ?[\pi]\bot$. Two natural PDL modalities are `until` and `while`. But these can in fact be easily defined in terms of other operations. We use the names `pdluntil` and `pdlwhile`, since `until` is already defined in Haskell (and we might naturally want to define `while`):

```
dneg :: PDL modality form => modality -> modality
dneg pi = test (nec pi bot)

pdluntil :: PDL modality form => form -> modality -> modality
pdluntil phi pi = kl (test (neg phi) -$- pi)

pdlwhile :: PDL modality form => form -> modality -> modality
pdlwhile phi pi = kl (test phi -$- pi)
```

... in symbols, these are given as:

$$[\texttt{pdluntil}(\varphi,\pi)]\psi := [(?\neg\varphi;\pi)^*]\psi$$
$$[\texttt{pdlwhile}(\varphi,\pi)]\psi := [(?\varphi;\pi)^*]\psi$$

As a final remark, in studying `pdlwhile` and `pdluntil` we see that `PDL` has a highly unintuitive functional dependency associated with it, namely `modality` $\rightarrow$ `form`. This is due to a certain amount of level confusion inherent in PDL – there is a modality for every formula, so given a specification for modalities one must be able to infer what the formulae of this logic is. We return to this peculiar nature of PDL in §2.3.3.

2.3 Instances

2.3.1 Predicate Logic

The most important thing about our implementation of predicate logic is that it is polymorphic. Predicate letters can be characters, strings, numbers, or even potentially things that take on types $\mathsf{pre}(\varphi)$ and $\mathsf{subst}(p,\varphi)$, where perhaps φ belongs to some logical language itself.

We may observe that the Backus-Naur style CFG declaration for predicate logic is in fact grammar for predicate logic in Polish notation. Our instance declaration of it as a Boolean Algebra is essentially just a translation scheme from infix notation to Polish notation.

```
data PL a =
     PLPred a              -- Predicate Letters
   | PLTop                 -- Top
```

```
   | PLNeg (PL a)          -- Negation
   | PLDisj (PL a) (PL a)  -- Disjunction

instance BA (PL a) where
  neg = PLNeg
  (\/) = PLDisj
  top = PLTop
  bot = neg top
```

We may observe that `PL` is actually a *monad* (this is because logical formulae are *trees*):

```
instance Monad PL where
  return = PLPred
  PLPred a >>= f = f a
  PLNeg x >>= f = PLNeg (x >>= f)
  PLDisj x y >>= f = PLDisj (x >>= f) (y >>= f)
  PLTop >>= f = PLTop
```

Similar monad instances can be given for just about all the data-types we will be introducing. Anyhow, a more useful declaration is an instantiation of `Preds` for predicate logic on characters.

```
instance Preds (PL Char) where
  a = PLPred 'a'; b = PLPred 'b'; c = PLPred 'c'; d = PLPred 'd'
  e = PLPred 'e'; f = PLPred 'f'; g = PLPred 'g'
  p = PLPred 'p'; q = PLPred 'q'; r = PLPred 'r'; s = PLPred 's'
  t = PLPred 't'; u = PLPred 'u'; v = PLPred 'v'
```

2.3.2 Epistemic Logic

We now turn to giving epistemic logic, and give its obvious instantiations. `EL` is a polymorphic datatype for with two parameters, one for predicate logic and the other for agents.

```
data EL agent a =
     ELPred a                              -- Predicate Letters
   | ELTop                                 -- Top
   | ELNeg (EL agent a)                    -- Negation
   | ELConj (EL agent a) (EL agent a)      -- Conjunction
   | ELK agent (EL agent a)                -- Modality

instance BA (EL agent a) where
  neg = ELNeg
  x \/ y = neg( neg x `ELConj` neg y)
  top = ELTop
  bot = neg top
```

```
instance ML agents (EL agents a) where
  nec a = ELK a

instance Preds (EL agents Char) where
  a = ELPred 'a'; b = ELPred 'b'; c = ELPred 'c'; d = ELPred 'd'
  e = ELPred 'e'; f = ELPred 'f'; g = ELPred 'g'
  p = ELPred 'p'; q = ELPred 'q'; r = ELPred 'r'; s = ELPred 's'
  t = ELPred 't'; u = ELPred 'u'; v = ELPred 'v'
```

We use conjunction since it is better for *ad hoc* modalities than disjunction, as we will observe shortly.

We note that the instantiation of `ML` here appears somewhat redundant. The reason for its peculiar declaration is that each agent in an *epistemic logic* corresponds to a modality (namely, their belief modality).

We next define a special case for Preds, namely on strings:

```
instance Preds String where
  a = "Abelard"; b = "Brauwer"; c = "Curry"; d = "De Morgan"
  e = "Eloise"; f = "Frege"; g = "Godel"
  p = "Prior"; q = "Quine"; r = "Russell"; s = "Skolem"
  t = "Turing"; u = "Urysohn"; v = "Venn"
```

Hence, if we force `((pos p :: String) p) :: EL String Char`, this will be exactly the same as `(pos "Prior") (PL (PLPred 'p')) :: EL String Char`.

We now turn to defining *ad hoc* modalities. Most of the ad-hoc modalities we have encountered so far have been easily expressible as λ expressions. The first non-trivial ad hoc modality is `PAL`:

```
data PAL = PAL

instance ML (PAL, EL agents a) (EL agents a) where
  nec (_, phi) (ELPred p) = phi --> ELPred p
  nec (_, phi) ELTop = top
  nec (_, phi) (ELNeg psi) = phi --> neg (nec (PAL, phi) psi)
  nec (_, phi) (psi `ELConj` chi) =
      (nec (PAL, phi) psi) /\ (nec (PAL, phi) chi)
  nec (_, phi) (ELK agent psi) = phi --> (nec agent) (nec (PAL, phi) psi)
```

We may observe that this definition resembles the reduction axiom definition of PAL rather closely. Another ad hoc modality close to our own research is *belief update*, $[BU\ \varphi]\psi$. Semantically, the idea is to take the current world ω under consideration, and change $R_a(\omega)$ to $R_a(\omega) \cap [\![\varphi]\!]$ for every agent a. On the one hand, this is a much more shallow change than a PAL update, but on the other hand it has more complicated reduction axioms:

$$
\begin{aligned}
[BU\ \varphi]p &\leftrightarrow p \\
[BU\ \varphi]\neg p &\leftrightarrow \neg p \\
[BU\ \varphi](\psi \wedge \chi) &\leftrightarrow [BU\ \varphi]\psi \wedge [BU\ \varphi]\chi \\
[BU\ \varphi]\neg(\psi \wedge \chi) &\leftrightarrow ([BU\ \varphi]\neg\psi) \vee ([BU\ \varphi]\neg\chi) \\
[BU\ \varphi]\neg\neg\psi &\leftrightarrow [BU\ \varphi]\psi \\
[BU\ \varphi]\Box_a\psi &\leftrightarrow \Box_a(\varphi \rightarrow \psi) \\
[BU\ \varphi]\neg\Box_a\psi &\leftrightarrow \Diamond_a(\varphi \wedge \neg\psi)
\end{aligned}
$$

Despite the complexity of these reduction axioms, they easily give rise to an ad hoc modality, albeit a rather bizarre one. This is because the way we write reduction axioms, and the way we write Haskell, is actually just pattern matching:

```
data BU = BU

instance ML (BU, EL agents a) (EL agents a) where
  nec (_, phi) (ELPred p) = ELPred p
  nec (_, phi) (ELNeg (ELPred p)) = neg (ELPred p)
  nec (_, phi) ELTop = top
  nec (_, phi) (ELNeg ELTop) = bot
  nec (_, phi) (psi `ELConj` chi) =
      (nec (BU, phi) psi) /\ (nec (BU, phi) chi)
  nec (_, phi) (ELNeg (psi `ELConj` chi)) =
      (nec (BU, phi) (neg psi)) \/ (nec (BU, phi) (neg chi))
  nec (_, phi) (ELNeg (ELNeg psi )) =(nec (BU, phi) psi)
  nec (_, phi) (ELK a chi) = nec a (phi --> chi)
  nec (_, phi) (ELNeg (ELK a psi)) = pos a (phi /\ (neg psi))
```

2.3.3 PDL

We now turn to PDL. PDL represents a challenge, since as alluded to earlier, every formula gives rise to a modality. However, this is clear by just looking at the way the grammars are defined:

$$
\begin{aligned}
\varphi &::= \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid [\pi]\varphi \\
\pi &::= a \mid ?\varphi \mid \pi_1;\pi_2 \mid \pi_1 \cup \pi_2
\end{aligned}
$$

But luckily, Haskell data-type declaration is a close relative to Backus-Naur notation for grammars, so we just need to define these data-types in parallel

```
data EPDL agent a =
```

```
     EPDLPred a                                  -- Predicate Letters
   | EPDLTop                                     -- Top
   | EPDLNeg (EPDL agent a)                      -- Negation
   | EPDLDisj (EPDL agent a) (EPDL agent a)      -- Disjunction
   | EPDLK (EPDLM agent a) (EPDL agent a)        -- Modality

data EPDLM agent a =
     EPDLA agent                                 -- Atomic Modalities
   | EPDLU (EPDLM agent a) (EPDLM agent a)       -- Union
   | EPDLC (EPDLM agent a) (EPDLM agent a)       -- Composition
   | EPDLT (EPDL agent a)                        -- Test
   | EPDLS (EPDLM agent a)                       -- Kleene Star
   | EPDLE                                       -- Everyone Believes
   | EPDLN                                       -- Empty Program
```

We now proceed to the tower of class instantiations.

```
instance BA (EPDL agent a) where
     neg = EPDLNeg
     (\/) = EPDLDisj
     top = EPDLTop
     bot = neg top

instance Preds (EPDL agents Char) where
  a = EPDLPred 'a'; b = EPDLPred 'b'; c = EPDLPred 'c'; d = EPDLPred 'd'
  e = EPDLPred 'e'; f = EPDLPred 'f'; g = EPDLPred 'g'
  p = EPDLPred 'p'; q = EPDLPred 'q'; r = EPDLPred 'r'; s = EPDLPred 's'
  t = EPDLPred 't'; u = EPDLPred 'u'; v = EPDLPred 'v'

instance Preds (EPDLM String a) where
  a = EPDLA "Abelard"; b = EPDLA "Brauwer"; c = EPDLA "Curry";
  d = EPDLA "De Morgan"; e = EPDLA "Eloise"; f = EPDLA "Frege";
  g = EPDLA "Godel"; p = EPDLA "Prior"; q = EPDLA "Quine";
  r = EPDLA "Russell"; s = EPDLA "Skolem" ;t = EPDLA "Turing";
  u = EPDLA "Urysohn"; v = EPDLA "Venn"

instance ML (EPDLM agents a) (EPDL agents a) where
     nec pi = EPDLK pi

instance PDL (EPDLM agents a) (EPDL agents a) where
    (-|-) = EPDLU
    (-$-) = EPDLC
    test = EPDLT
    kl = EPDLS
    nl = EPDLN
```

It should be remarked that since we shall be reasoning about common belief, we have introduced the *Everyone Believes* atom as `EPDLE`. This gives rise to the celebrated ad hoc modality *common belief*, and the modified version $C_B(\varphi, \cdot)$ from [2]. We give a class for everyone believes and common belief, and a polymorphic function `cbm` representing the LCC extension:

```
class (PDL modality form) => EpPDL modality form | modality -> form where
   eb :: modality                          -- Everybody Believes
   cb :: modality                          -- Common Belief

cbm :: (EpPDL modality form) => form -> modality
cbm x = pdlwhile x eb

instance EpPDL (EPDLM agents a) (EPDL agents a) where
   eb = EPDLE
   cb = kl eb
```

This concludes our treatment of language. We turn now to semantics.

3 Semantics

3.1 Introduction to the `Model` class

The basic approach we take here is to introduce a class for models. There appear to be two essential ingredients in a model - an object and a language.

```
class Model obj lang | obj -> lang where
  infix 3 |=
  (|=) :: obj -> lang -> Bool
```

Here, we set the fixity of the double turnstile to be *lower than* the fixity of our logical infix operators and PDL operators.

Also note that we are assuming functional dependence of the language defined on the object. We would ideally like to define multiple languages on the same object, but without this restriction type inference is simply too challenging, so we are regrettably forced to make a design choice here.

3.2 Example $\mathbb{Z}_2$

Our first example is somewhat esoteric, but it is intended to illustrate the principles behind the methods we are developing here. We will study the integers modulo 2. We define our semantics in the following manner:

```
data Z2 = Z2
```

```
instance Model Z2 Int where
  _ |= n = odd n
```

We can naturally think of a Boolean algebra on the integers with these as semantics. We can think of $(1+)$ as a negation operation, and $(*)$ is conjunction (on the other hand, $(+)$ is in fact a `nexor` operation). Unfortunately, since our Boolean algebras is somewhat tethered to using disjunction, we will have to make due with defining disjunction using DeMorgan's law.

```
instance BA Int where
  neg = (1+)
  n \/ m = neg (neg n * neg m)
  top = 1
  bot = 0
```

In fact, we might even imagine modal operators on integers in $\mathbb{Z}_2$. For instance, it is easy to check that $(\lambda x.x^x)$ is a modal operator (it behaves as if it were an identity operator). We can in fact instantiate it:

```
instance ML Z2 Int where
  nec _ n = n^n
```

3.3 Epistemic Logic

We now turn to our treatment of epistemic logic. Before we begin, we go over our particular design strategy for epistemic models:

- A model has two components:

  (*1*) A `Map` which takes a letter `pred` to a `list` of worlds `i`

  (*2*) A `Map` which takes a world-modality relation pair `(i,mod)` to a `list` of worlds.

We assume that all lists are duplicate free by construction. We turn now to declaring these models and instantiating them. Here is the declaration of this type of model, along with a simple instance of `Show`:

```
data (Ord i, Ord mod, Ord pred) => MM i mod pred =
     MM (Map i [pred]) (Map (i,mod) [i])

instance (Show i, Show mod, Show pred, Ord i, Ord mod, Ord pred) =>
  Show (MM i mod pred) where
    show (MM val rel) = "val: " ++ show val ++ "\n" ++ "rel: " ++ show rel
```

We next turn to giving semantics. We will make use of the index operator `(!)`.

```
infixl 5 !
(!) :: (Ord a, Ord b) => (Map a [b]) -> a -> [b]
m ! i = maybe [] id $ Map.lookup i m

instance (Ord i, Ord mod, Ord pred) =>
 Model ((MM i mod pred),i) (EL mod pred)
 where
   ((MM val _),w) |= (ELPred p) = p `elem` (val ! w)
   _ |= ELTop = True
   mw |= (ELNeg phi) = not (mw |= phi)
   mw |= (phi `ELConj` psi) = (mw |= phi) && (mw |= psi)
   (m@(MM _ rel),w) |= (ELK ag phi) = all (\v -> (m,v) |= phi) (rel ! (w,ag))
```

3.4 Epistemic PDL

3.4.1 Data Structure

We now turn to giving the data structure for epistemic PDL. It is essentially the same as before, except now we carry around a list of agents for modeling common belief. It has a similar simple instance of `Show` as before:

```
data (Ord i, Ord ags, Ord pred) => MMepdl i ags pred =
  MMepdl [ags] (Map i [pred]) (Map (i,ags) [i])

instance (Show i, Show agent, Show pred, Ord i, Ord agent, Ord pred) =>
  Show (MMepdl i agent pred) where
    show (MMepdl ags val rel) =
      "ags: " ++ show ags ++ ['\n'] ++
      "val: " ++ show val ++ ['\n'] ++
      "rel: " ++ show rel
```

3.4.2 Basic Semantics

We now turn to giving the basic semantics. It is essentially the same as in case of `EL`:

```
instance (Ord i, Ord agent, Ord pred) =>
 Model ((MMepdl i agent pred),i) (EPDL agent pred)
 where
   ((MMepdl _ val _),w) |= (EPDLPred p) = p `elem` (val ! w)
   _ |= EPDLTop = True
   mw |= (EPDLNeg phi) = not (mw |= phi)
   mw |= (phi `EPDLDisj` psi) = (mw |= phi) || (mw |= psi)
   (m,w) |= (EPDLK pi phi) = all (\v -> (m,v) |= phi)
```

```
$ runpi m w pi (const []) []
```

... the only significant difference come from the fact that we are now running the program π to determine the worlds to visit. We turn now to a substantial digression:

3.4.3 Executing Modal Programs π

It is widely known that intersection-free PDL is rather well behaved. While it is true that it fails to have compactness, it retains the finite model property, so it has a completeness theorem thanks to the Fischer-Ladner construction.

Yet it has a surprising virtue no-one has thought of: it is a good candidate for model checking in a lazy functional programming environment. This holds true despite the fact that $(\cdot)^*$ appears to present a difficult computational challenge. This is an unexpected, but rather happy development.

The trick is that in a lazy environment, potentially costly computations are frequently short circuited. For instance, the following takes less than a second to compute:

```
*> all even [1..]
False
```

...even though the left hand side contains a computation which never halts. To see why this halts so quickly, it is instructive to look at the source:

```
all :: (a -> Bool) -> [a] -> Bool
all f = and . map f

and :: [Bool] -> Bool
and = foldr (&&) True

(&&) :: Bool -> Bool -> Bool
(&&) True True = True
(&&) True x = x
(&&) False _ = False
```

So our strategy is to generate the list of worlds to check using `runpi`, which will be programmed in a recursive manner to maximize potential laziness. We give its type so we can explain our design concept:

```
runpi :: (Ord i, Ord agent, Ord pred) =>
            (MMepdl i agent pred)          -- The model
         -> i                              -- The index of a world
         -> (EPDLM agent pred)             -- The program to be run
         -> ([i] -> [i])                   -- A continuation
         -> [i]                            -- A set of visited worlds
```

```
        -> [i]                                  -- A set of worlds
```

Note that we have chosen a restricted form for our continuation; we will discipline ourselves so as to only pass lists of visited worlds to continuations. Furthermore, we will employ the continuation like a stack. Recall that the call our `Model` instantiation makes to `runpi` is:

```
runpi m w pi (const []) []
```

Thus the initial continuation, `(const [])` behaves sort of like an empty stack.

We now turn to defining this function recursively:

```
-- Empty program
runpi _ _ EPDLN cont vis = cont vis

-- Atomic program ag
runpi (MMepdl _ _ rel) w (EPDLA ag) cont vis =
  let newvis = [ x | x <- (rel ! (w,ag)), not (x `elem` vis)] in
    newvis ++ cont (newvis ++ vis)

-- ? phi
runpi m w (EPDLT phi) cont vis =
  let newvis = if (((m,w)|= phi) && (not (w `elem` vis))) then [w] else [] in
    newvis ++ cont (newvis ++ vis)

-- Everyone Believes
runpi m@(MMepdl ags _ _) w EPDLE cont vis =
  runpi m w (foldl comps nl ags) cont vis
   where
     comps pi1 ag = pi1 -|- (EPDLA ag)
```

With a little reflection, we see that this definition of everyone believes is the usual one: if our agents are a, b, and c then it just runs the program $R_a \cup R_b \cup R_c$. However, since it is being output by a `foldl`, we are technically running the program $R_c \cup R_b \cup R_a \cup \varnothing$ in this case. We turn now to giving our first challenging operation:

```
-- Union
runpi m w (pi1 `EPDLU` pi2) cont vis =
  runpi m w pi1 newcont vis
    where
      newcont = runpi m w pi2 cont
```

Suppose in our program we are evaluating is $\pi_1 \cup \pi_2$. Then `runpi` evaluates π_1 and leaves evaluation of π_2 as a continuation. In Haskell, we essentially compose the existent continuation as part of `runpi m w pi2 cont`, which is a partial function application awaiting a list of visited worlds.

In this respect, the continuation behaves like a stack, and so we can think of $\cup$ as behaving kind of like an interrupting process.

A more complicated use of the continuation is in program composition:

```
-- Composition
runpi m w (pi1 `EPDLC` pi2) cont vis =
  newcont vis
    where
      pi1out = runpi m w pi1 (const []) []
      newcont = foldr pi2cont cont pi1out
      pi2cont v nxcont = runpi m v pi2 nxcont
```

In program composition, we first run π_1; but since we don't really care about its output, we pass it an empty set of visited worlds and continuation. We then take its output and put elements on the continuation stack. This is done by folding `newcont`, which takes as an argument a world v from the output of π_1, and a continuation.

Finally, the last case of a program constructor is Kleene star. We first remark that an easy definition of Kleene star is as follows (assuming we are evaluating at world w):

$$\pi^* = \{w\} \cup \pi \cup (\pi;\pi) \cup (\pi;\pi;\pi) \cup \ldots$$

Now, because of Haskell's laziness, this is actually rather simple to express:

```
myplusls pi = (test top) : (map (pi -$-) (myplusls pi))
mykl pi = foldr (-|-) (test top) (myplusls pi)
```

Although technically, `mykl` would return something like:

$$?\top \cup (\pi;?\top) \cup (\pi;\pi;?\top) \cup (\pi;\pi;\pi;?\top) \cup \ldots \cup ?\top$$

Now, while this way is somewhat easy to program in Haskell, we argue that we cannot use this approach. For, as we asserted above, in the case of program composition we dispense with the running list of visited worlds. As it is, if we implemented Kleene star according to the above intuition of its semantics, `runpi` would never halt on a π^* command.

Instead, we use a different intuition. If we are trying to evaluate $\mathbb{M}, w \models [\pi_1 \cup \pi_2]\varphi$ computationally we are employed in a search. In our system, we try π_1 and π_2 sequentially in an effort to find some world v such that $\mathbb{M}, v \not\models \varphi$. If we find such a world, we return `False`. If no such world is found, we return `True`. In the meanwhile, we use continuations in effect to keep track of the worlds we have visited.

Under this view, π^* can be understood as an instruction to start a new depth first search with π to find worlds in the modal model `m`, which can be thought of as a treelike structures. Since it is a new search, the old visited worlds have to be discarded. This is because there's no telling if some π worlds have been accessed that their π successors have been visited and their $\pi;\pi$ successors and

so on. Thus, the π search has new bookkeeping to keep track of the worlds it has found. We have carefully annotated our algorithm below, as it is not trivial.

```
-- Kleene star
runpi m w (EPDLS pi) cont vis =
  -- Output newfound worlds, then continue
  newvis ++ cont (newvis ++ vis)
    where
      -- Newfound worlds come from the output of our piprocess search
      -- we filter them against our list of previously visited worlds
      newvis = filter (not.(`elem` vis)) (piprocess [w] [w])

      -- Piprocess takes two arguments: a load a list of visited worlds
      -- it outputs a list of worlds
      -- if the load is empty, it outputs the empty list
      piprocess [] _ = []

      -- If the load is not empty, it
      --  (1) Outputs the top
      --  (2) Evaluates pi on the top, with with the list of visited worlds.
      --  (3) Pushes the output on the load,
      --      and prepends it to the visited worlds
      --  (4) Recurses
      piprocess (x:xs) subvis = x : (piprocess (piout ++ xs) (piout ++ subvis))
        where
          piout = runpi m w pi (const []) subvis
```

3.5 Testing

We finally turn briefly to implementation for testing purposes. The following is just simple model with three agents, *Abelard, Brauwer* and *Curry*. They all know that either q or r is true, exclusively, and they know that p is true. We implement this model in basic epistemic logic and epistemic PDL.

```
univ1 :: MM String String Char
univ1 = MM vals rels
  where
  vals = Map.fromList $ zip ["w1","w2"] $ sequence [['p'],['q','r']]
  rels = Map.fromList [((x,ag),["w1","w2"]) | x <- ["w1","w2"], ag <- [a,b,c]]

univ2 :: MMepdl String String Char
univ2 = MMepdl ags vals rels
  where
  ags = [a,b,c]
  vals = Map.fromList $ zip ["w1","w2"] $ sequence [['p'],['q','r']]
```

```
  rels = Map.fromList [((x,ag),["w1","w2"]) | x <- ["w1","w2"], ag <- ags]
```

We can run these models to check that our machinery is working:

```
*> (univ,"w1")|= q
True
*> (univ,"w2")|= q
False
*> (univ2,"w1")|= EPDLK cb q
False
*> (univ2,"w1")|= EPDLK cb (neg (q <->r))
 True
```

4 Conclusion and Further Research

4.1 Dynamics

We now have two possible roads for studying dynamics:

(*1*) We can think of model updates as part of our language, and provide truth conditions in our instantiation of `Models`.

(*2*) We can think contrive a program that takes as an argument an action model and action world pair, a formula and outputs a formula, via the reduction axioms in [2].

4.2 Classes

Further research is needed, we feel, on the use of multi-parameter classes and their use in logic. Since modal logic models are first order models under the standard translation, as well as models of propositional logic, our design choice to have a functional dependency for the `Models` class is regrettable.

4.3 Preprocessing and Normal Forms

It should be observed that $[\pi^{**}]\varphi$ is logically equivalent to $[\pi^{*}]\varphi$, yet the latter takes much less time to compute in the given framework. Thus, it seems natural to have pre-processing on formulae before they are evaluated, which would either involve heuristics for simplification or some kind of normal form, or both.

4.4 Formal Proof Verification

A promising platform for doing development in computational logic is the Isabelle proof assistant. Isabelle supports export of Haskell code; furthermore, Isabelle's class system includes the specification of laws that a class must conform to, which are proved in the process of instantiation. Such a platform would be ideal for codevelopment of computational logic. Sadly, the class system in Isabelle current lacks support for multi-parameter classes. At any rate, since the project of DEMO is to do computational modal logic, it would seem natural to combine giving proofs alongside programming.

4.5 Conclusion

We have successfully shown how to implement intersection-free PDL in Haskell, and we hope this will prove beneficial to further development of computational approaches to modal logic.

References

[1] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001.

[2] J. van Benthem, J. van Eijck, and B. Kooi. Logics of communication and change. *Inf. Comput.*, 204(11):1620–1662, 2006. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/j.jc.2006.04.006.