Lecture Notes on
Monadic Logic Programming

15-816: Linear Logic
Frank Pfenning

Lecture 20

We have discussed both forward and backward chaining at length; in this lecture we address the question how they can be combined effectively. The central idea is the use of a *monad* to separate forward chaining from backward chaining. Monads originated in category theory and have been used successfully in functional programming. In logic programming, they were first proposed in their present incarnation in the LolliMon language [LPPW05].

1 Monads

We introduce a new type $\{A\}$ (sometimes written as $\bigcirc A$ in logic or $T\ A$ in category theory). It is *weaker* than truth in the sense that $\vdash\ A \multimap \{A\}$, but not the other way around. It may look like the dual to $!A$, but as we will see in a later lecture, this is not quite the case.

In order to explain the meaning of $\{A\}$ judgmentally we need a new judgment, $A\ \mathit{lax}$, which is somehow a counterpart of $A\ \mathit{pers}$. But it will only appear explicitly on the right-hand side of a sequent. This is the opposite of $A\ \mathit{pers}$, which only appears explicitly on the left-hand side.

The judgment $A\ \mathit{lax}$ is weaker than truth, so it can be derived from truth:[1]

$$\frac{\Delta \vdash A\ \mathit{true}}{\Delta \vdash A\ \mathit{lax}}\ \mathsf{lax}$$

[1] In this lecture we will uniformly write $A\ \mathit{true}$ for ephemeral truth, be it on the left-hand or the right-hand side of a sequent.

The connection in the other direction is given by a new cut rule.

$$\frac{\Delta \vdash A\ lax \quad \Delta', A\ true \vdash C\ lax}{\Delta, \Delta' \vdash C\ lax}\ \mathsf{cut}\{\,\}$$

It expresses that we can view $A\ lax$ as representing an ephemeral truth, but only while we try to prove the lax truth of C.

Originally, in lax logic [FM97] $\{A\}$ was conceived to stand for the truth of A under some constraints, where the constraints remain completely abstract. It is true only if the constraints are satisfied. We can see this in the two rules above. The rule lax allows the empty (or always true) constraint. The cut$\{\,\}$ rule expresses that if A is true under some constraint which may assume it is true, but only if we are trying to prove C under some constraint. In essence, the cut$\{\,\}$ rule relies on the conjunction of the constraints in the two derivations.

The two rules match in the sense that if the first premise of the cut$\{\,\}$ was derived using the lax rule, then the cut$\{\,\}$ is immediately reduced to a cut. This illustrates, however, that all cut and left rules of the sequent calculus we have introduced so far must allow the succedent of the sequent to be either $A\ true$ (which we have generally written as just A) or $A\ lax$. This is an analogous step to systematically adding persistent resources Γ to each rule in the calculus already.

Once the judgmental principles are settled, it is almost trivial to internalize $A\ lax$ as a proposition.

$$\frac{\Delta \vdash A\ lax}{\Delta \vdash \{A\}\ true}\ \{\,\}R \qquad \frac{\Delta, A\ true \vdash C\ lax}{\Delta, \{A\}\ true \vdash C\ lax}\ \{\,\}L$$

2 Cut and Identity

Let's check the cut reduction:

$$\frac{\dfrac{\Delta \vdash A\ lax}{\Delta \vdash \{A\}\ true}\ \{\,\}R \quad \dfrac{\Delta', A\ true \vdash C\ lax}{\Delta', \{A\}\ true \vdash C\ lax}\ \{\,\}L}{\Delta, \Delta' \vdash C\ lax}\ \mathsf{cut}$$

$$\longrightarrow_R$$

$$\frac{\Delta \vdash A\ lax \quad \Delta', A\ true \vdash C\ lax}{\Delta, \Delta' \vdash C\ lax}\ \mathsf{cut}\{\,\}$$

In the broader proof of the admissibility of cut, when considering a cut$\{\ \}$

$$\frac{\Delta \vdash A\ lax \quad \Delta', A\ true \vdash C\ lax}{\Delta, \Delta' \vdash C\ lax}\ \mathsf{cut}\{\ \}$$

we push up the cut into the inference in the proof of the first premise, keeping the second premise fixed. The only way this strategy does not apply is if the first premise is the lax rule, with the premise of $A\ true$. But then we can just reduce it to an ordinary cut of the two premises.

This is reminiscent of the way cut! always pushed up the cut into the second premise, until a copy rule was encountered. The lax judgment affords a simplification since there is no copying.

In the proof of the identity expansion the steps are more or less forced:

$$\frac{}{\{A\}\ true \vdash \{A\}\ true}\ \mathsf{id}_{\{A\}} \quad \longrightarrow_E \quad \frac{\dfrac{\dfrac{\dfrac{}{A\ true \vdash A\ true}\ \mathsf{id}_A}{A\ true \vdash A\ lax}\ \mathsf{lax}}{\{A\}\ true \vdash A\ lax}\ \{\ \}L}{\{A\}\ true \vdash \{A\}\ true}\ \{\ \}R$$

Because of the judgmental transition, we need three rule applications: two propositional, and one judgmental. We'll have to remember this when considering focusing.

3 Examples: Unit and Bind

In functional programming, monads [Mog91] are presented via two functions, unit and bind:

$$\begin{array}{lcl} \mathsf{unit} & : & A \to \{A\} \\ \mathsf{bind} & : & \{A\} \to (A \to \{B\}) \to \{B\} \end{array}$$

These characterize lax logic axiomatically, although we do not pursue this here. But we can prove linear versions of them in the sequent calculus. First unit (omitting the standard $true$ judgments as usual, but keeping lax

explicit):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\mathsf{id}}{A \vdash A\ lax}\,\mathsf{lax}}{A \vdash \{A\}}\,\{\}R}{\vdash A \multimap \{A\}}\,\multimap R$$

bind is only slight more complicated:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\mathsf{id} \quad \dfrac{\dfrac{\dfrac{}{B \vdash B}\,\mathsf{id}}{B \vdash B\ lax}\,\mathsf{lax}}{\{B\} \vdash B\ lax}\,\{\}L}{A, A \multimap \{B\} \vdash B\ lax}\,\multimap L}{\{A\}, A \multimap \{B\} \vdash B\ lax}\,\{\}L}{\{A\}, A \multimap \{B\} \vdash \{B\}}\,\{\}R}{\vdash \{A\} \multimap (A \multimap \{B\}) \multimap \{B\}}\,\multimap R^2$$

This demonstrates that monads are compatible with linearity.

4 Focusing

We recall that the identity expansion shown above starts with the $\{\}R$ rule. This is a hint that the lax modality should be considered *negative*, and its right rule should be invertible. For focused sequents $\Delta \to A$ when then have the following right rule:

$$\dfrac{\Delta \to A\ lax}{\Delta \to \{A\}}\,\{\}R$$

Correspondingly, the left rule should be applied in focus. The question is whether we can retain focus on A.

$$\dfrac{\Delta, [A] \to C\ lax}{\Delta, [\{A\}] \to C\ lax}\,\{\}L?$$

Because there is an implicit judgmental transition (from $\{A\}\ true$ to $A\ lax$ and then to $A\ true$) we suspect we may need to lose focus. To construct a

counterexample it is always useful to reconsider the identity, so let's try to construct a focused proof of $\{P^-\} \to \{P^-\}$.

$$\dfrac{\dfrac{\dfrac{[P^-] \to P^-\ lax}{[\{P^-\}] \to P^-\ lax}\ \{\}L?}{\{P^-\} \to P^-\ lax}\ \mathsf{focus}L}{\{P^-\} \to \{P^-\}}\ \{\}R$$

Notice that at this point we fail, because $[P^-] \to \gamma$ only succeeds if $\gamma = P^-\ true$.

So, as suspected, we need to lose focus and the correct rule is

$$\dfrac{\Delta, A \to C\ lax}{\Delta, [\{A\}] \to C\ lax}\ \{\}L$$

Finally, we consider the $A\ lax$ judgment. On the left, it does not occur because the judgment itself should be considered positive. It occurs on the right, but represents the end of an inversion phase, waiting for focusing to occur. This means we have the additional rule

$$\dfrac{\Delta \to [A]}{\Delta \to A\ lax}\ \mathsf{foc}R\{\}^*$$

As before, the focusing rules are restricted. If we are just modeling chaining (allowing inversion anywhere), the rule is restricted to the case where there is no focus in Δ. In *focusing*, Δ must also be stable, that is, consist entirely of negative propositions and positive atoms.

5 Polarization

As a step towards the logic programming implementation, we now *polarize* the propositions, including the lax modality. This will tell us where inversion and focusing phases begin and end, and it will restrict us to a fragment that has a sensible operational interpretation.

We have considered these before—the crucial issue is how to polarize the lax modality. We already saw that the proposition $\{A\}$ is negative, but the judgment hiding underneath, $A\ lax$, is positive. We say that $\{\}$ is *negative on the outside* and *positive on the inside*. Like the exponential modality, it

is a polarity-changing connective.

$$\begin{array}{llll} \text{Negative Props.} & A^- & ::= & P^- \mid A_1^+ \multimap A_2^- \mid A_1^- \mathbin{\&} A_2^- \mid \top \mid \forall x{:}\tau.\, A^- \mid \{A^+\} \\ \text{Positive Props.} & A^+ & ::= & P^+ \mid A_1^+ \otimes A_2^+ \mid \mathbf{1} \mid A_1^+ \oplus A_2^+ \mid \mathbf{0} \mid A^- \mid !A^- \mid \exists x{:}\tau.\, A^+ \end{array}$$

The inclusion of negative propositions in the positive ones is often written a $\downarrow A^-$, where "$\downarrow$" is like a modality. However, it does not change the logical meaning, just the focusing behavior of the propositions.

6 Forward and Backward Chaining

We have two forms of stable sequents, one represents forward chaining, the other backward chaining.

$$\begin{array}{ll} \Delta \to P^-\ \mathit{true} & \text{Backward Chaining } (\Delta \text{ stable}) \\ \Delta \to C^+\ \mathit{lax} & \text{Forward Chaining } (\Delta \text{ stable}) \end{array}$$

Notice in particular that we do not have sequents $\Delta \to C^+\ \mathit{true}$, because the only way that positives are included in negatives is through the lax modality.

A prototypical clause that participates in *backward chaining* has the form

$$A^+ \multimap P^-$$

Focusing on such a clause will only succeed if the succedent will be $P^-\ \mathit{true}$. If it is $C^+\ \mathit{lax}$, focusing will fail because there is no rule

$$\frac{\text{(fails)}}{\Delta, [P^-] \to C^+\ \mathit{lax}}$$

Conversely, a prototypical clause that participates in *forward chaining* has the form

$$A^+ \multimap \{B^+\}$$

Focusing on such a clause will only succeed if the succedent is $C^+\ \mathit{lax}$. If it is $P^-\ \mathit{true}$ it will fail, because there is no rule

$$\frac{\text{(fails)}}{\Delta, [\{B^+\}] \to P^-\ \mathit{true}}$$

If the conclusion is indeed $C^+\ lax$, then focusing on the clause will never fail at the end, since we have the general transition

$$\frac{\Delta, B^+ \to C^+\ lax}{\Delta, [\{B^+\}] \to C^+\ lax}\ \{\}L$$

which initiates an inversion phase.

How can we move between forward and backward chaining? When we decide to focus on a succedent $C^+\ lax$

$$\frac{\Delta \to [C^+]}{\Delta \to C^+\ lax}\ \mathsf{foc}R\{\}$$

we break down $[C^+]$ until we reach a negative, and this may break down all the way to an atom, initiating backward chaining. So in this case, we may leave forward chaining altogether.

Similarly, if we are focused on

$$\Delta, [A^+ \multimap \{B^+\}] \to C\ lax$$

then a subgoal will be

$$\Delta' \to [A^+]$$

which may devolve into a negative atom, starting forward chaining. In this case, the backward chaining is an auxiliary subgoal that enables a forward-chaining step to proceed.

During backward chaining, we switch to forward chaining if the succedent is $\{A\}$ in the inversion phase of backward chaining.

$$\frac{\Delta \to A\ lax}{\Delta \to \{A\}}\ \{\}R$$

This may also happen when working on a subgoal during backward chaining.

7 Example: Testing Bipartiteness

We develop a small implementation of testing whether a graph is bipartite, that is, can be colored with two colors such that no two adjacent vertices have the same color.

The algorithm we want to implement is quite simple. In each phase we traverse a connected component of graph. If there is no uncolored node, we

are finished and can check if there are two adjacent nodes with the same color. If not, pick an arbitrary node and color it an arbitrary color. Then we iterate: pick a node that has not yet been colored and has an adjacent colored node. Given it the opposite color of its neighbor. When no such node exists any more, we move to the next phase, again arbitrarily picking an uncolored node.

We start with some type declarations in CLF.

```
vertex : type.

edge : vertex -> vertex -> type.

color : type.
a : color.
b : color.
```

We have two predicates: node x, of which we initially have an linear one for every vertex x in the graph, and a persistent edge $x\ y$ for every edge in the graph. We assume it is already saturated to be symmetric.

Then the forward-chaining rules that constitute the inner loop, propagating the color constraints would be:

```
clr : vertex -> color -> type.
node : vertex -> type.

cb : node X * !edge X Y * !clr Y a -o {!clr X b}.
ca : node X * !edge X Y * !clr Y b -o {!clr X a}.
```

These rules consume node x, so that each node is colored exactly one.

Next we need to implement the outer loop. We assume we are given a list of vertices,

```
nil : vlist.
cons : vertex -> vlist -> vlist.
```

and have to assume each one into the context. This happens in a backward-chaining program. The previously mentioned convention and backward-chaining clauses are written as $A \circ\!\!- B$ and forward-chaining ones as $A \multimap \{B\}$ comes into effect.

```
nbp : vlist -> type.
nbp/cons : nbp (cons X L) o- (node X -o nbp L).
```

We use the name nbp (short for *not bipartite*) because we will succeed if we find a contradiction.

Once the context has been initialized, we have to pick a node, assign it an arbitrary color (say a), and then use the two rules above until we have reached quiescence.

```
nbp/nil : nbp nil o- node X * (!clr X a -o {nbp nil}).
```

This rule switches from forward chaining to backward chaining, once the persistent assumption !clr x a has been made.

When we have reached quiescence, we recurse (subgoal nbp nil), picking another node x that has not yet been colored. At the end, once all nodes have been colored, we look for a contradiction: adjacent nodes with the same color.

```
nbp/fail : nbp nil o- !clr X C * !edge X Y * !clr Y C.
```

We can improve this program by making the node predicate *affine*. Recall that affine resources (marked with the affine modality "@") are used *at most once*. This allows us to shirt-circuit the algorithm and exit as soon as we have found two adjacent nodes with the same color. The complete program is below.

```
vertex : type.
edge : vertex -> vertex -> type.

color : type.
a : color.
b : color.

clr : vertex -> color -> type.

node : vertex -> type.

cb : @node X * !edge X Y * !clr Y a -o {!clr X b}.
ca : @node X * !edge X Y * !clr Y b -o {!clr X a}.

vlist : type.
nil : vlist.
cons : vertex -> vlist -> vlist.

nbp : vlist -> type.
```

```
nbp/fail : nbp nil o- !clr X C * !edge X Y * !clr Y C.
nbp/nil : nbp nil o- @node X * (!clr X a -o {nbp nil}).
nbp/cons : nbp (cons X L) o- (@node X -o nbp L).
```

References

[FM97] M. Fairtlough and M.V. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, August 1997.

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, July 2005. ACM Press.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.