Explanation Based Learning

Presentation · April 2020

CITATIONS
0

READS
64

2 authors, including:

Shaik Naseera
Jawaharlal Nehru Technological University, Anantapur
42 PUBLICATIONS **67** CITATIONS

SEE PROFILE

Explanation Based Learning

Prof. Shaik Naseera

JNTUACEK

Introduction

- Generally, humans appear to learn quite a lot from a single **example**.

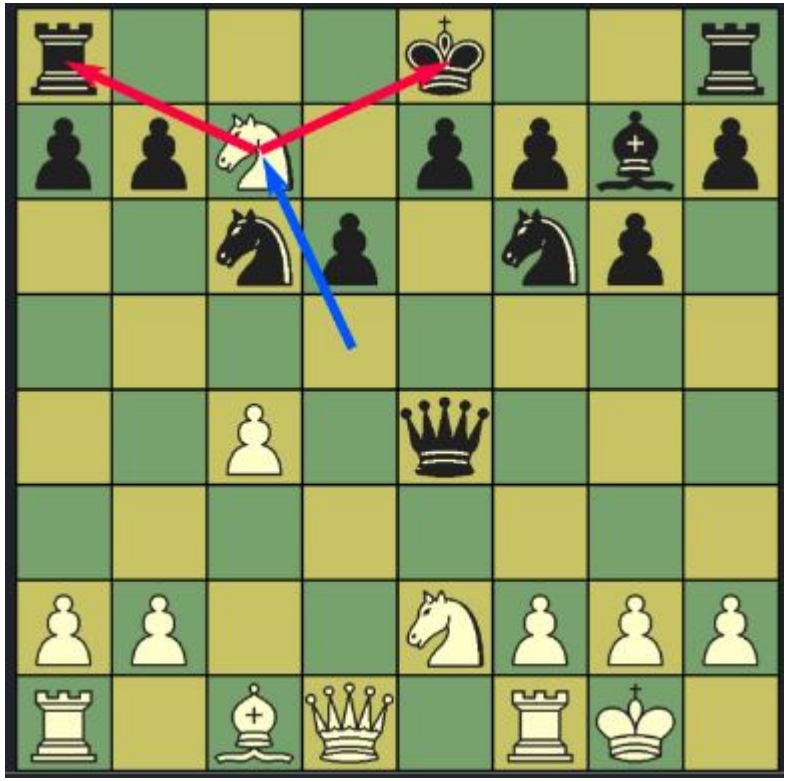
Ex:- if we touch the heat, it burns our hand.

- **Explanation-based learning (EBL) extracts general rules from single example by explaining** the example and generalizing the explanation.

Ex:- The knight’s attack on both the king and queen of the chess board .

Example: chess game

The knight's attack on the king and queen on the chess board is as shown below. This position is called fork. Because knight attacks both the king and queen . i.e., double simultaneous attack.



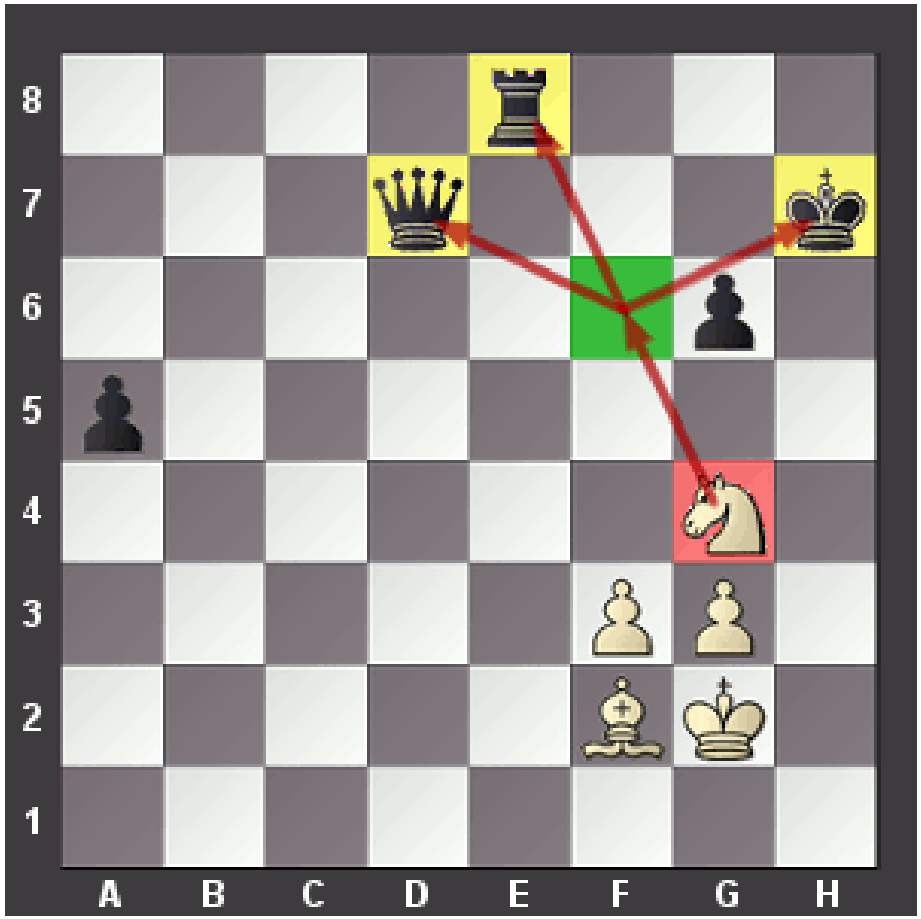
In fork, the chess layer should move the king thereby leaving the queen open to capture.

- From this single experience the player is able to learn quite about the fork trap.
- The idea is that if any piece x attacks both the opponents king and another piece y, then the piece y will be lost.
- We don’t need to see dozens of positive and negative examples of fork positions in order to draw these conclusions.
- From this one experience, we learn to avoid this trap in the future and perhaps to use it to our own advantage.

What makes such single-example learning possible?

- The answer is knowledge
- The chess player has plenty of domain specific knowledge including the rules of chess and previously acquired strategies.
- That knowledge is used to identify the critical aspects of the training example.
- In case of fork, we know that the double simultaneous attack is important while the precise position and type of attacking piece is not.

Another example

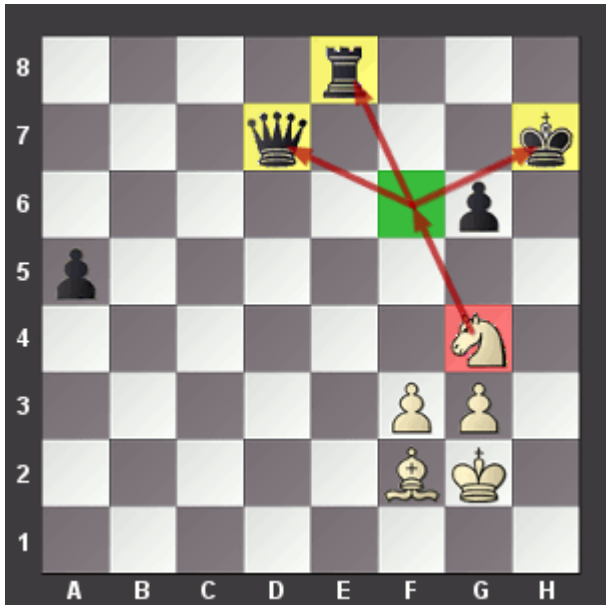


Strategy of EBL

- Unlike other methods, EBL is not data intensive.
- EBL is analytical and knowledge–intensive approach.
- EBL system learn to attempt from a single example x by explaining why x is an example for the target concept.
- The explanation is the generalized and the system's performance is improved through the availability of this knowledge.

- An EBL accepts 4 kinds of input:
  - **A training example**-- what the learning program *"sees"* in the world.
  - **A goal concept**-- a high level description of what the program is supposed to learn.
  - **A operational criterion**-- a description of which concepts are usable.
  - **A domain theory**-- a set of rules that describe relationships between objects and actions in a domain.
- From this EBL computes a generalization of the training example that is sufficient not only to describe the goal concept but also satisfies the operational criterion.

- In chess game, the goal concept might be “bad position for black”
- And the operationalzed concept would be a generalized description of situations similar to the training example, given in terms of pieces and their relative positions.
- The last input to EBL is domain theory, in our case, the rules of chess.

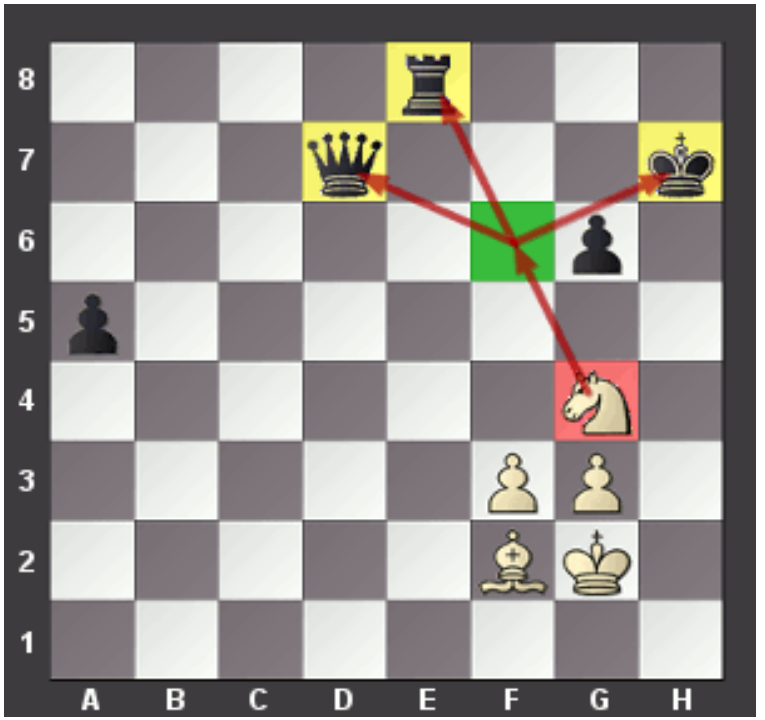


EBL generalization (EBG)

- This has two steps: Explain and generalize
- **Explanation**-- the domain theory is used to prune away all unimportant aspects of the training example with respect to the goal concept.
- **Generalization**-- the explanation is generalized as far possible while still describing the goal concept.

Example: Chess game

- First EBL step chooses to ignore white's pawn, king and rook and construct an explanation of white's knight, black's king and black's queen each in their specific positions.
- Next, explanation is generalized, i.e., moving the pieces to different part of the board is still bad for the black.



Arithmetic simplification

- General rules in LISP, By using this knowledge base more general rules are extracted
- simplify (Mult (Const 0) x) = Const 0
- simplify (Mult x (Const 0)) = Const 0
- simplify (Plus (Const 0) x) = simplify x // (constant 0 + x)
- simplify (Plus x (Const 0)) = simplify x //(x + constant 0)
- simplify (Mult (Const 1) x) = simplify x
- simplify (Mult x (Const 1)) = simplify x
- simplify (Minus x (Const 0)) = simplify x
- simplify (Plus (Const x) (Const y)) = Const (x + y)
- simplify (Minus (Const x) (Const y)) = Const (x - y)
- simplify (Mult (Const x) (Const y)) = Const (x * y)
- simplify x = x

Procedure for proof tree

- First thing is to convert the expression from infix notation to an S-Expression (symbolic expression).
- Traverse the "tree" recursively and apply a set of rules at each node.

e.g. if this node contains an operation whose operands are both constants, perform the operation now and replace the node with the result.

- Once the basic functionality was in place, it was a matter of adding new new simplification rules to the system.

Another example

- Suppose our problem is to simplify 1 x (0 + X).
- The knowledge base includes the following rules:

//if u is written as v and v is simplified to w then u is simplified to w

1. **Rewrite(u, v) Λ Simplify(v, w) ⇒ Simplify(u, w) .**

//if u is a primitive then u is simplified to u

2. **Primitive (u) ⇒ Simplify (u, u) .**

//if u is an arithmetic unknown then u is a primitive

3. **ArithmetiUnknown (u) ⇒ Primitive (u).**

//if u is a number then u is a primitive

4. **Number (u) ⇒ Primitive (u) .**

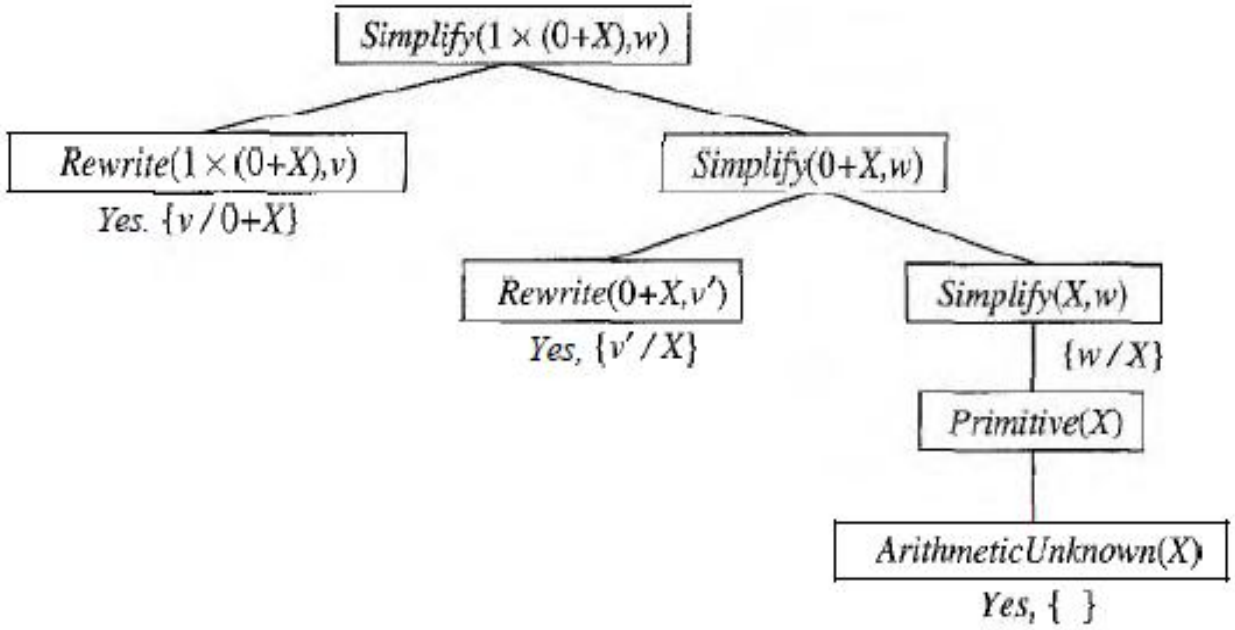
// 1xu is written as u

5. **Rewrite(1 x u, u) .**

//0+u is written as u

6. **Rewrite(0 + u, u) .**

Proof Tree for simplify 1 x *(0 + X)*



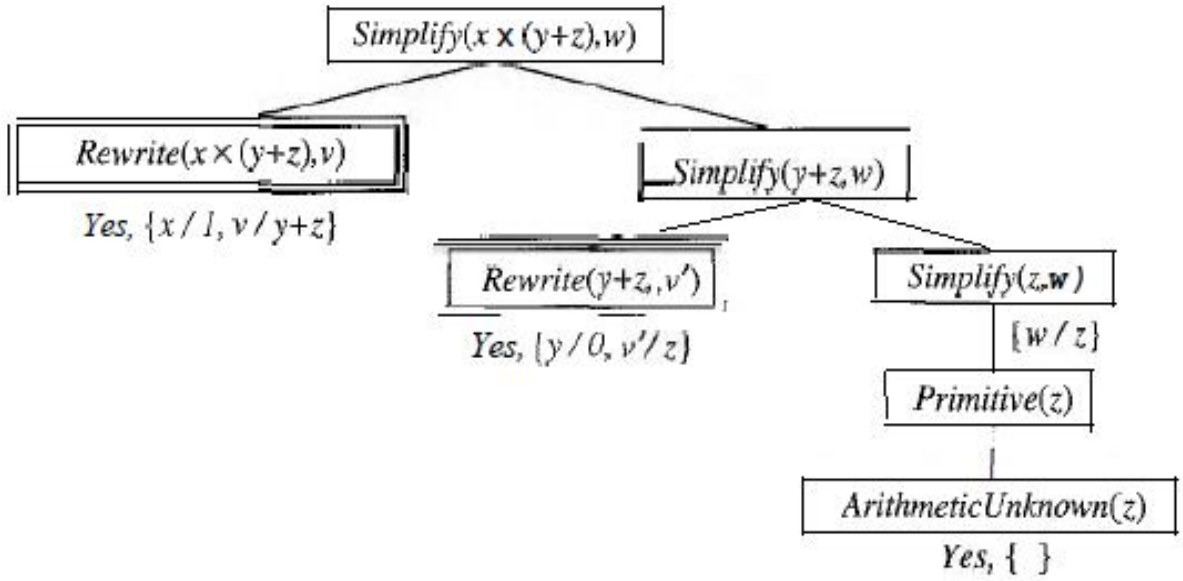
- The proof for the original problem instance using the rules from 1 to 6.
- The leaf nodes form the solution for the main problem.

  Rewrite(1x(0+X),v)∧ Rewrite((0+X),v ')∧ ArithmeticUnknown(X)⇒Simplify(1x(0+X),w)
- Notice that the first two conditions on the left-hand side are true *regardless of the value of X.* We can therefore drop them from the rule, yielding

  ArithmeticUnknown(x) ⇒ Simplify(1 x (0 + x), x)
- *Proof, that the answer is X.*

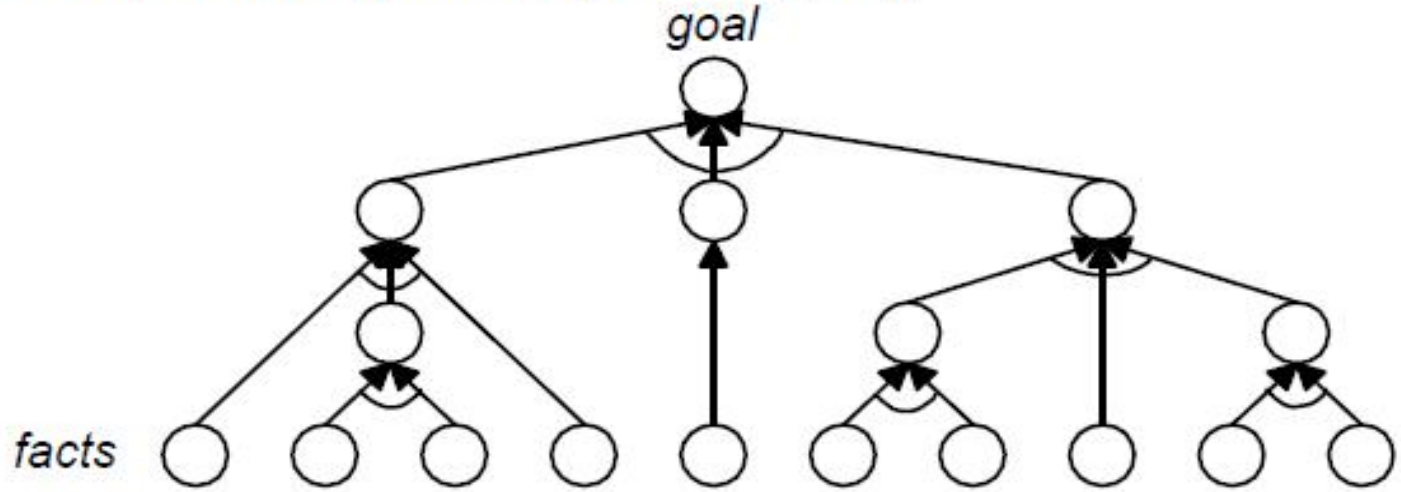
Generalized tree

- The below tree shows the proof for a problem instance with all constants replaced by variables, from which we can derive a variety of other rules.

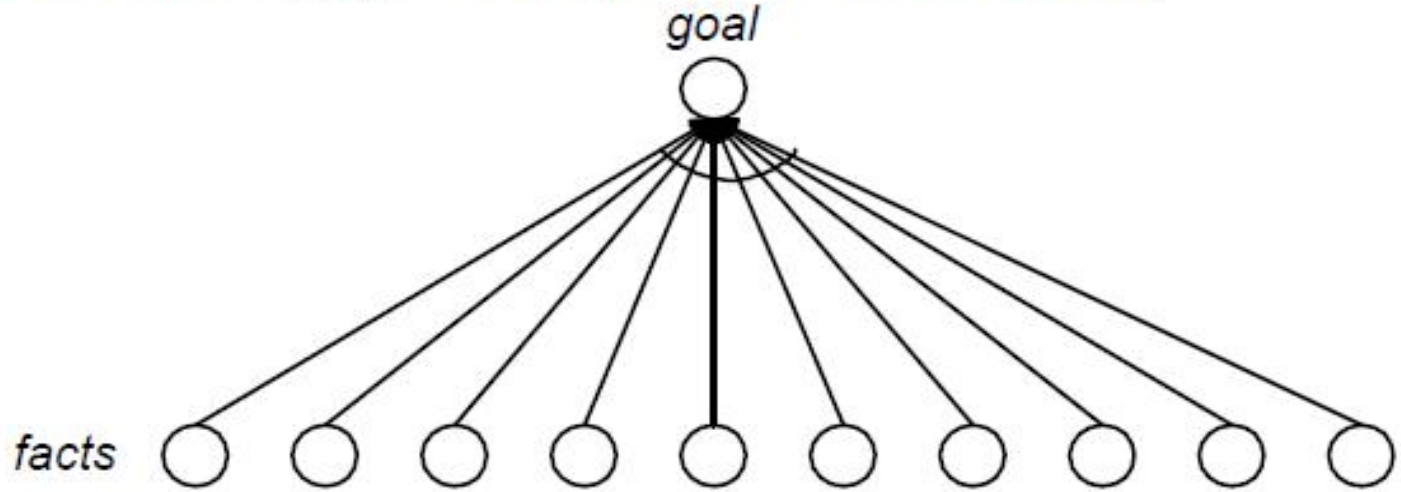


Standard Approach to EBL

An Explanation (detailed proof of goal)



After Learning (go directly from facts to solution):



EBL method

The EBL method actually constructs two proof trees simultaneously.

- The first proof for the original problem instance using the rules in knowledge base
- The second proof tree uses a *variabilized* goal in which the constants from the original goal are replaced by variables.
- As the original proof proceeds, the variabilized proof proceeds in step, using *exactly the same rule applications.*
- This could cause some of the variables to become instantiated. For example, in order to use the rule *Rewrite(1 x u, u), the variable x in the subgoal Rewrite(x x (y + z), v) must* be bound to 1.
- Similarly, y must be bound to *0 in the subgoal Rewrite(y + z, v') in order to* use the rule *Rewrite(0 + u, u).*
- *Once we have the generalized proof tree, we take the leaves* (with the necessary bindings) and form a general rule for the goal predicate:
- *Rewrite(1 x (0 + z) , 0 + z) A Rewrite(0 + z, z) A Arithmetic Unknown(z) ⇒ Simplify(1 x (0 + z), z) .*
- Notice that the first two conditions on the left-hand side are true *regardless of the value of z.*
- We can therefore drop them from the rule, yielding

  ArithmeticUnknown(z) ⇒ Simplify(1 x (0 + z), z)

Proof, that the answer is z.

Recap

- To recap, the basic EBL process works as follows:

1. Given an example, construct a proof that the goal predicate applies to the example using the available background knowledge.
2. In parallel, construct a generalized proof tree for the variabilized goal using the same inference steps as in the original proof.
3. Construct a new rule whose left-hand side consists of the leaves of the proof tree and whose right-hand side is the variabilized goal (after applying the necessary bindings from the generalized proof).
4. Drop any conditions that are true regardless of the values of the variables in the goal.

Improving the Efficiency

$$Primitive(z) \Rightarrow Simplify(1 \times (0 + z), z).$$

This rule is as valid as, but ***more general*** than, the rule using $Arithmetic Unknown$, because it covers cases where z is a number. We can extract a still more general rule by pruning after the step $Simplify(y + z, w)$, yielding the rule

$$Simplify(y + z, w) \Rightarrow Simplify(1 \times (y + x), w).$$

In general, a rule can be extracted from ***any partial subtree*** of the generalized proof tree. Now we have a problem: which of these rules do we choose?

The choice of which rule to generate comes down to the question of efficiency. There are three factors involved in the analysis of efficiency gains from EBL:

1. Adding large numbers of rules can slow down the reasoning process, because the inference mechanism must still check those rules even in cases where they do riot yield a solution. In other words, it increases the **branching factor** in the search space.
2. To compensate for the slowdown in reasoning, the derived rules must offer significant increases in speed for the cases that they do cover. These increases come about mainly because the derived rules avoid dead ends that would otherwise be taken, but also because they shorten the proof itself.
3. Derived rules should be as general as possible, so that they apply to the largest possible set of cases.

Prototypical EBL Architecture