Higher-Order and Modal Logic as a Framework for Explanation-Based Generalization

Scott Dietzen Frank Pfenning

CMU-CS-89-160

October 16, 1989

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract. Certain tasks, such as formal program development and theorem proving, are inherently *higher-order* because they fundamentally rely upon the manipulation of higher-order objects such as functions and predicates. Computing tools intended to assist these higher-order tasks are at present inadequate in both the amount of 'knowledge' they contain (*i.e.*, the level of support they provide) and in their ability to 'learn' (*i.e.*, their capacity to enhance that support over time). The application of a relevant machine learning technique — explanation-based generalization (EBG) — has been limited to first-order problems. We extend EBG to generalize higher-order values, thereby facilitating its application to higher-order domains.

Logic programming provides a uniform framework in which all aspects of explanation-based generalization and learning may be defined and carried out. First-order Horn logics (*e.g.*, Prolog) are not, however, well suited to higher-order applications. Instead, we employ λProlog, a higher-order logic programming language, as a framework for realizing higher-order EBG. This requires extending λProlog with the necessity operator $\Box$ of modal logic, which leads to the language $\lambda^{\Box}$Prolog. The necessity operator elegantly captures the distinction between domain theory and training instance upon which EBG depends. We develop a meta-interpreter realizing EBG for $\lambda^{\Box}$Prolog and provide examples of higher-order EBG.

EBG has been described as 'speed-up' generalization in that its results, while improving performance, do not address previously unsolvable problems. By extending the framework with user interaction, EBG becomes a means by which user-provided search-control knowledge may be made manifest. This facilitates application to domains in which unguided problem solving is often intractable. Of particular interest to us, and thus investigated here, are theorem proving and formal program development.

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

1 Introduction

Certain tasks, such as program development and theorem proving, are inherently *higher-order* because they fundamentally rely upon the manipulation of higher-order objects such as functions and predicates. To enhance the support computing tools can provide for such complex domains, it will be necessary to increase considerably the 'knowledge' represented in these tools. Successfully coding all this knowledge *a priori* is impossible due to the scope, complexity, and evolutionary nature of these domains. Rather, tools must support *assimilation* of problem solving experience. However, simply memoizing particular solutions will be insufficient; instead experience must be *abstracted* or *generalized*. *Learning*, the ability to generalize and assimilate from experience, will therefore have a significant impact on the success of future tools.

Much of machine learning research may be divided between *inductive*, or *similarity-based* learning, and *analytical*, or *explanation-based* learning. To date, work on the latter relies primarily on *explanation-based generalization* (EBG) as its central mechanism [35,5,34,13]. Through the analysis of a formal problem solution (*i.e.*, a proof or *explanation*), EBG determines the preconditions sufficient to apply the same solution strategy in general. EBG yields a derived rule that more efficiently solves the original as well as related problems. Under EBG a single example may be generalized since the proof constrains the space of possible results. Similarity-based learning and generalization (SBG), on the other hand, rely upon multiple training examples (often both positive and negative) to arrive at an articulation of the sharing among those (positive) instances [1,8]. While the proof-based generalizations of EBG are necessarily valid, similarity-based generalizations are guaranteed only to the extent that they cover the given examples. Hybrids of these approaches are a current topic of research; see Hirsh [21], for example.

Generalization and learning performance are intimately tied to the underlying language for representation, or *representation domain*. If knowledge is encoded in an inappropriate representation domain, then it is less likely that the desired generalizations can be expressed in a natural and concise manner, and also less likely that they can even be found. In particular, the cumbersome encoding of higher-order domains within first-order languages inhibits reasoning and generalization. But to date, the application of EBG has been limited to first-order representation domains. To facilitate EBG's application to higher-order domains, we extend the technique to *higher-order explanation-based generalization* – that is, EBG in which functions and predicates as well as first-order constants may be abstracted, or replaced with variables.

Recently, the logic programming paradigm has been used as a foundation for EBG [27,40,22,2]. One argument put forward in favor of the logic programming framework is that it admits a uniform representation for all aspects of EBG: domain theory, training instance, goal, derived rule, operationality criteria, *etc.* (These concepts are defined in Section 2.) This helps in explicating the underlying principles in a uniform way and clarifies semantic issues. In this paper we explore two ways of enriching the representation domain of Horn logic (*e.g.*, Prolog): integrated support for higher-order objects including variables ranging over such objects, and support for modal concepts. Both of these have a significant impact on EBG.

EBG has often been characterized as 'speed-up' generalization in that its derived rules improve performance by reducing or eliminating search, but cannot enable the solution of previously unsolvable problems. This description becomes misleading if user interaction is combined with EBG

[1] An extended abstract appears in the *Sixth International Workshop on Machine Learning* [9].

in the treatment of *intractable* domains, such as those that arise in theorem proving and program development. Under such a scenario, the user guides problem solving and EBG becomes a means by which user-provided search-control knowledge may be made manifest. Although the resulting generalizations are in the deductive closure of the rule-base (*i.e.*, the set of existing rules), the intractability of the domain can preclude discovery without user guidance.

We begin, in Section 2, by introducing first-order EBG within the logic programming framework. Our formulation of EBG differs from the traditional in that operationality criteria have been replaced with an explicit separation between domain theory and training instance. Section 3 explores the enrichment of the representation domain with higher-order values, leading to the higher-order logic programming language λProlog and higher-order EBG. In Section 4 EBG's reliance upon the partitioning of its rule-base into *domain theory* (*i.e.*, general rules) and *training instance* (*i.e.*, particular facts) is addressed by extending λProlog with the □ operator of modal logic. This yields a rich language for EBG, $\lambda^{\Box}$Prolog. Sections 5 and 6 illustrate higher-order EBG through its application to theorem proving and program development tasks. We then explore, in Section 7, the ramifications of our examples, and in particular user interaction, upon assimilation and learning. $\lambda^{\Box}$Prolog is further formalized in Section 8. We develop, in Sections 9 and 10, an implementation of $\lambda^{\Box}$Prolog and EBG through a series of meta-interpreters written in λProlog. All the examples contained herein were actually produced with the prototype. Section 11 then motivates an enhanced $\lambda^{\Box}$Prolog interpreter currently under construction. Finally, we offer some suggestions for future work in Section 12.

Acknowledgments. Our meta-interpreter depends on the eLP, the implementation of λProlog developed by Conal Elliott and Frank Pfenning in the framework of the Ergo project at Carnegie Mellon University [12]. We thank Conal Elliott, Masami Hagiya, Haym Hirsh, Dale Miller, Tom Mitchell, and William Scherlis for their thoughtful comments on our presentation.

2 First-order EBG in the Logic Programming Framework

We begin by briefly illustrating explanation-based generalization with a first-order example from DeJong & Mooney [5, pages 158–166]. (We apologize to any readers offended by the morbidity of this example, but it has become standard in the literature.) EBG problems consist of a *domain theory*, or set of general rules:

```
kill A B  :-  hate A B, possess A C, weapon C.
hate W W  :-  depressed W.
possess U V  :-  buy U V.
weapon Z  :-  gun Z.
```

and a *training instance*, or set of particular facts:

```
depressed john.
buy john obj1.
gun obj1.
```

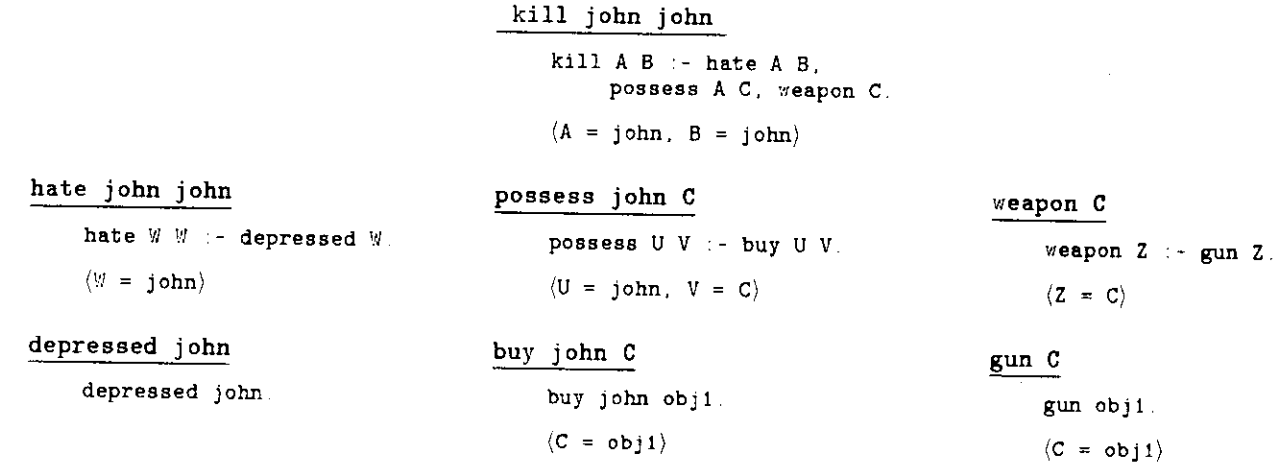

Figure 1: First-order proof.

The above example, as well as those to follow, is formulated in λProlog. As in Prolog, variables are distinguished by capitalization and ',' denotes conjunction. The variables of clauses, such as `Z` in `weapon Z :- gun Z`, are implicitly universally quantified.

The EBG algorithm is additionally provided with a goal or *query*, such as

```
kill john john.
```

EBG then requires a proof, or *explanation*, that solves the given query. Within the logic programming paradigm, such a proof may be expressed as a trace of λProlog search. A proof of the above query is illustrated in Figure 1. Goals of the proof are underlined, while the program clause which reduces a particular goal appears underneath. In the course of applying a clause, its variables may be *unified*, or instantiated, with constants or variables of the goal. These associated *unification constraints* appear enclosed by '⟨⟩'.

EBG produces an encapsulation of the proof strategy by generalizing this explanation. In Figure 2 a generalized proof is constructed that corresponds to the specific one except that training instance clauses are *omitted*. At the root of the new proof is a generalized query, which is derived from the original query by replacing all of the first-order constants with logical variables: the goal `(kill john john)` becomes the *fully* general goal `(kill X Y)`. Each domain theory rule applied in the first proof is correspondingly applied in the second. This potentially restricts the outcome by propagating unification constraints through the proof tree (for example, `(kill X Y)` becoming `(kill X X)`). Leaves of the generalized proof (*e.g.*, `(gun V)`) correspond to subgoals of the original proof that were derived from the rules of the training instance. These leaves are accumulated in a conjunction of conditions sufficient to establish the generalized query:

```
kill X X  :-  depressed X, buy X C, gun C.
```

We will frequently refer to the resulting derived rule, or proof encapsulation, as an *explanation-based generalization*, or simply a *generalization*.

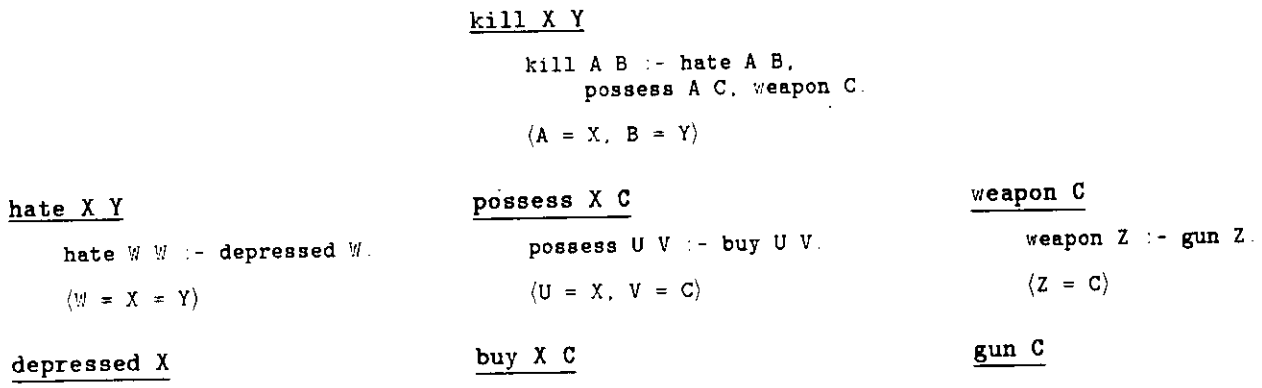

Figure 2: First-order generalized proof.

Operationality. Suppose we revise the example by replacing the last rule of the domain theory with

```
weapon Z  :-  gun Z; knife Z; grenade Z; ...
```

where ';' represents disjunction. Although this version yields the same generalization as above, the new clause suggests we might prefer a more generally applicable result in terms of weapon rather than gun:

```
kill X X  :-  depressed X, buy X C, weapon C.
```

Our formulation of EBG already affords a means for achieving the above result: since only domain theory rules are included in the generalized proof, we may rewrite the example with domain theory

```
kill A B  :-  hate A B, possess A C, weapon C.
hate W W  :-  depressed W.
possess U V  :-  buy U V.
```

and training instance

```
weapon Z  :-  gun Z; knife Z; grenade Z; ...
depressed john.
buy john obj1.
gun obj1.
```

This yields the desired derived rule and the associated generalized proof shown in Figure 3.

Traditionally, the means of restricting the depth of the generalized proof tree is termed *operationality*: by establishing that a particular goal (*e.g.*, weapon Z) meets an *operationality criteria*, the subtree deriving it is pruned from the generalized proof. Under such a formulation of EBG, the distinction between training instance and domain theory is meaningless, at least with respect to generalization. Our approach differs in that operationality is expressed as a predicate over clauses (whether or not they are domain theory), rather than a predicate over goals. That is, we have essentially replaced a 'goal-based' notion of operationality with a 'clause-based' one. Section 4 continues this discussion.

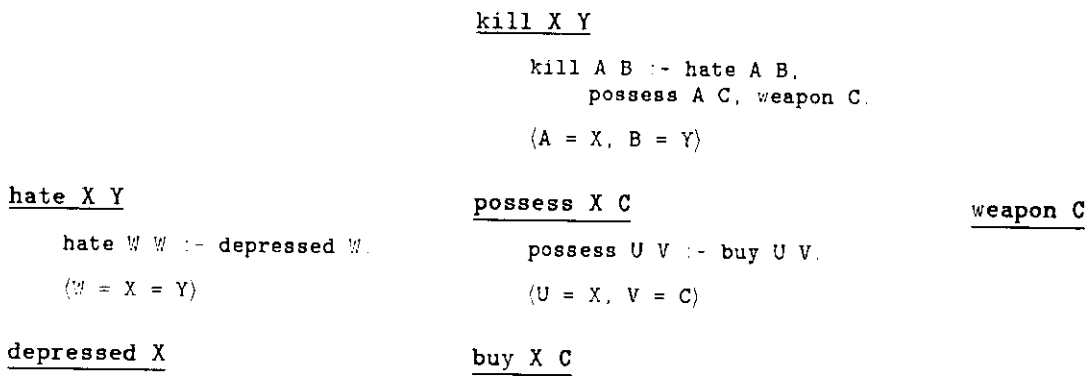

Figure 3: Alternate first-order generalized proof.

---

3 Higher-order Representation Domains

A domain is *higher-order* if it contains higher-order values such as functions or predicates. A language or application over such a domain is itself said to be higher-order. For example, a higher-order programming language allows functions to be bound to variables, passed as parameters, and returned from function calls. Similarly, a higher-order logic provides for quantification over functions and predicates. *Higher-order EBG* is, then, explanation-based generalization in which the candidates for variable replacement include higher-order objects.

A *representation domain* is a language for expressing the values of an application. When higher-order values are expressed in first-order representation domains, reasoning and programming with these *ad hoc* encodings is difficult: We often need 'new variables', need to check conditions such as 'where ... does not occur in ...', or must implement substitution in a way that 'renames bound variables if necessary.' We advocate, instead, *higher-order* representation domains that provide for natural expression and manipulation of higher-order objects. For example, the function $f(x) = g(x, 2)$ might be represented as $f = \lambda x.\, g(x, 2)$, or the quantified expression $\forall x\, \exists y.\, x < y$ as $\forall\,(\lambda x.\, \exists\,(\lambda y.\, x < y))$. The use of the single name-binding operator λ allows the following operations to be implemented once within the representation domain rather than within the rules of each of its clients [39,20]:

- α-conversion — the renaming of bound variables ($e.g.$, $\lambda x.x = \lambda y.y$)
- β-conversion — capture-avoiding substitution ($e.g.$, $(\lambda x.\lambda y.fxy)y = \lambda y'.fyy'$)
- η-conversion — a weak extensionality principle ($e.g.$, $\lambda x.fx = f$)

Example 1. Consider the following higher-order rules for symbolic integration: The first treats exponentiation (missing is the restriction that $a \neq -1$), the second extracts a constant factor, and the third splits a sum.

```
intgr  X\((3 * (expn X 2)) + cos X)   R
    intgr  X\((F X) + (G X))   X\((Fi X) + (Gi X))
        :-  intgr  F  Fi,  intgr  G  Gi.
    (F = X\(3  (expn X 2)),  G = cos,  R = X\((Fi X) + (Gi X)))

intgr X\(3 * (expn X 2))  Fi
    intgr  X\(A  (H X))   X\(A  (Hi X))  :-  intgr  H  Hi.
    (A = 3,  H = X\(expn X 2),  Fi = X\(A  (Hi X)))

intgr cos Gi
    intgr  cos  sin.
    (Gi = sin)

intgr  X\(expn X 2)   Hi
    intgr  X\(expn X A1)   X\((expn X (A1 + 1)) div (A1 + 1))
    (A1 = 2,  Hi = X\((expn X (A1 + 1)) div (A1 + 1)))
```

Figure 4: Higher-order proof.

$$\forall a.\quad \int \lambda x.\ x^a \;=\; \lambda x.\ x^{(a+1)}/(a+1)$$

$$\forall a\ \forall f\ \forall f'.\quad \int f \;=\; f' \quad\Rightarrow\quad \int \lambda x.\ a * f(x) \;=\; \lambda x.\ a * f'(x)$$

$$\forall f\ \forall f'\ \forall g\ \forall g'.\quad \int f \;=\; f' \ \wedge\ \int g \;=\; g' \quad\Rightarrow\quad \int \lambda x.\ f(x) + g(x) \;=\; \lambda x.\ f'(x) + g'(x)$$

(The traditional binding notation dx has been replaced with explicit function definition via λ.) Within a higher-order representation domain, that is, one that supports name binding (via λ), the restriction that x not be free in the expression a is captured without complicating side conditions.

To encode the previous integration rules within our representation domain, λProlog, we use a predicate '`intgr`' to relate a function and its indefinite integral:

```
pi A\ (
   intgr   X\(expn X A)    X\((expn X (A + 1)) div (A + 1))).
pi A\ (pi F\ (pi G\ (
   intgr   X\(A * (F X))   X\(A * (G X))     :-  intgr F  G))).
pi F\ (pi Fi\ (pi G\ (pi Gi\ (
   intgr   X\((F X) + (G X))   X\((Fi X) + (Gi X))
     :-  intgr F  Fi,  intgr G  Gi)))).
```

'\' binds the variable that immediately precedes it, thus acting as infix λ-abstraction. The symbol `pi` stands for universal quantification: $\forall y\ .\ B$ is represented as `pi Y\`B, expressing that `Y` is bound in B. Although the `pi`'s would have been inferred by λProlog, they are explicitly included to simplify future discussion.

With the additional training instance fact

```
intgr  cos  sin.
```

the query

GG

```
intgr  X\((F X) + (G X))   X\((Fi X) + (Gi X))
     :-  intgr F  Fi,  intgr G  Gi.
(GG = intgr  X\((F X) + (G X))   X\((Fi X) + (Gi X)))
```

intgr F Fi

```
intgr  X\(A * (H X))   X\(A * (Hi X))  :-  intgr  H  Hi.
(F = X\(A * (H X)),  Fi = X\(A * (Hi X)))
```

intgr G Gi

intgr H Hi

```
intgr  X\(expn X A1)   X\((expn X (A1 + 1)) div (A1 + 1))
(H = X\(expn X A1),  Hi = X\((expn X (A1 + 1)) div (A1 + 1)))
```

Figure 5: Higher-order generalized proof.

```
intgr  X\(3 * (expn X 2) + cos X)   F
```

yields the solution

```
F   =   X\ (3 * (expn X (2 + 1) div (2 + 1)) + sin X)
```

and the generalization

```
intgr G Gi  =>
   intgr    X\(A * expn X B + G X)    X\(A * (expn X (B + 1) div (B + 1)) + Gi X)
```

We will often use implication '=>' in place of ':-'; `A => B` is a notational variant of `B :- A`.

The proof and generalized proof associated with this example are given in Figures 4 and 5. The generalization space of higher-order EBG is significantly larger than that of first-order since higher-order constants are additionally subject to variable replacement: consider that in the first-order case of Figure 2, the goal `(kill X Y)` is fully general, while for higher-order, a single variable `GG` ranging over propositions is fully general.

Also unlike the proofs of Section 2, the integration proofs make use of higher-order unification, which allows variables to be instantiated with functions as well as first-order constants. For example, `(F a)` and `(g a a)` may be unified by instantiating `F` to `X\(g X X)`, `X\(g a X)`, `X\(g X a)`, or `X\(g a a)`. Since none of these alternatives is an instance of another, this example illustrates the nondeterministic nature of higher-order unification. Restrictions on free and bound variables are enforced by higher-order unification: consider that `X\(expn X A)` will not unify with `X\(expn X X)`, since `A` may not contain occurrences of `X`. Donat & Wallen [11] also use higher-order unification, but their representation of integrals does not use functions in the same way, and some of the problems that arise when trying to apply generalizations are avoided in our representation.

Many other domains naturally involve name binding constructs, and are thus best represented in a higher-order language: logics (when viewed as an object language to be manipulated), programming languages, and natural language [39,32,29]. This same lack of adequate representation also

arises when one wants to reason 'at the meta-level' — that is, about control strategies for logic programming, theorem proving, or the EBG algorithm itself. One would like facts (propositions) or properties (predicates) to be objects themselves. Prolog and other first-order representation domains allow this to some extent, but in a way that is only operationally, but not logically motivated. This complicates reasoning about these concepts and in practice prohibits the application of methodologies such as EBG.

Nadathur & Miller introduce λProlog, a logic programming language supporting higher-order functions and predicates [36]. Although λProlog is a typed language, we will omit type declarations from our examples for simplicity. λProlog utilizes Huet's complete algorithm for higher-order unification [24]. Although higher-order unification is only semi-decidable and can be highly nondeterministic, Huet's algorithm is very effective in practice.

4 Modal Logic and EBG

EBG relies upon the separation of domain theory and training instance since only rules of the former are incorporated into the generalized proofs. To differentiate the two, we prefix domain theory clauses with □. The □ (or L) operator is borrowed from modal logic — *i.e.*, logics in which propositions have multiple levels or *modes* of truth, such as 'may be' and 'must be' (see, for example, Hughes & Cresswell [26]). Intuitively, □ precedes *necessarily* true sentences, or equivalently, those true in *all possible states* or at *all times*. Non-prefixed sentences are only *contingently* true, true in the *current state* or at the *current time*.

We illustrate the use of □ on the first-order example cited previously. Within λProlog □ is replaced with box. The set of rules and facts, or *program*, may now be expressed as

```
box (pi A\ (pi B\ (pi C\  (kill A B  :-  hate A B, possess A C, weapon C)))).
box (pi W\                (hate W W  :-  depressed W)).
box (pi U\ (pi V\         (possess U V  :-  buy U V))).
box (pi Z\                (weapon Z  :-  gun Z)).
depressed john.
buy john obj1.
gun obj1.
```

Due to the inclusion of box, we may no longer rely upon λProlog's implicit universal quantification. This is because our EBG algorithm differentiates between the clauses `box (pi X\ (G X))` and `pi X\ (box (G X))`. Section 8 continues this discussion.

From the query `(kill john john)`, EBG produces the generalization

```
kill X X  :-  depressed X, buy X C, gun C.
```

This derived rule holds for all `X` and `C` and, moreover, is necessarily true in that it follows from the domain theory. Thus, it may be better expressed as

```
box (pi X\ (pi C\  (kill X X  :-  depressed X, buy X C, gun C))).
```

Although, the latter representation of the generalization is preferred, it is the former which is generated by the current implementation. The issue is resolved in Section 11.

EBG's distinction between domain theory and training instance corresponds to the modal logic distinction between necessary and contingent truth in that *boxed* clauses (domain theory) represent knowledge which is already *general* in that it is *necessarily* true, while unboxed clauses (training instance) represent *contingent* or *particular* knowledge. Clauses of the training instance, as they are excluded from generalized proofs, can safely be removed without invalidating the derived generalizations. Such revision can be explained semantically as changing states.

Suppose that within the suicide example of Section 2, we replace the last program clause with `(box (gun obj1))`. This has the effect of anchoring the generalization to `obj1`, with the result of the identical query being

```
box (pi X\ (depressed X, buy X obj1  =>  kill X X)).
```

The generalized proof associated with this result is similar to that of Figure 2, except that its rightmost branch is solved. By converting training instance clauses to domain theory, we have made the resulting generalization more specific. This is, however, dangerous in that the generalization then depends upon the validity of `(box (gun obj1))`. In another configuration where `obj1` is not a gun, the derived rule becomes false!

In realizing our version of operationality, we have already illustrated the conversion of clauses in the opposite direction, that is, from domain theory to training instance (Section 2, Figure 3). Such a transformation is benign in that the resulting derived rules can not be invalidated by revising the training instance.

Traditional notions of operationality do offer the advantage of allowing more localized pruning of the generalized proof: by declaring only a single subgoal to be operational, the entire branch underneath is excluded. Accomplishing the same pruning within our paradigm requires that each of the program clauses applied within the branch be removed from the domain theory. If a particular rule is used pervasively in a proof, alternate occurrences might have to be artificially discriminated for the purposes of EBG. Operationality criteria suffer a corresponding problem in the treatment of recurring subgoals.

Furthermore, clause-based operationality is more powerful in that clauses associated with interior nodes of the generalized proof tree may be abstracted. For example, consider a new formulation of the suicide problem in which the first clause is made training instance:

```
pi A\ (pi B\ (pi C\  (kill A B  :-  hate A B, possess A C, weapon C))).
```

This yields the generalized higher-order proof of Figure 6 captured in the following generalization:

```
boxpi (GG  :-  depressed W, buy U V, gun Z,
               (hate W W, possess U V, weapon  Z => GG)).
```

The above introduces a notational convention: rather than explicitly universally quantifying each of the variables of a domain theory clause as in `(box (pi GG\ (pi W\ ...)))`, we use the abbreviation `boxpi`. This should not yet be considered an extension to $\lambda^{\Box}$Prolog, but merely a device to make presentation more concise.

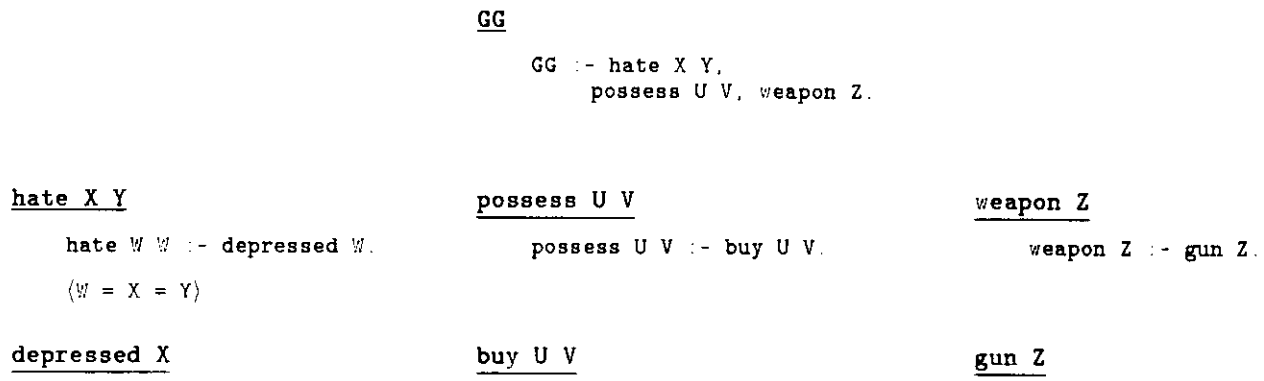

Figure 6: Alternative higher-order generalized proof.

Higher-order EBG allows goals, as well as functions and first-order objects, to be abstracted from the proof. The derived rule that results is more generally applicable in that GG may be given alternative instantiations. The expressive power of λProlog exceeds that of present EBG languages by affording the higher-order quantification of GG, as well as the inclusion of an implication within the preconditions of a derived rule.

Del Cerro takes another approach incorporating modal logic into the logic programming framework which is independent of EBG [6,7]. For treatments of automated theorem proving in modal logics outside of logic programming, see [45,44].

$\lambda^{\Box}$Prolog. As has already been suggested, the motivation for the extended language, $\lambda^{\Box}$Prolog, is that higher-order EBG may be realized within its underlying architecture. $\lambda^{\Box}$Prolog is currently implemented with an interpreter written in λProlog. Before discussing the implementation, we present further examples of higher-order EBG.

5 Theorem Proving

The logic programming paradigm is itself based on a theorem proving methodology, and its support for search and unification suggest it as an implementation language for theorem provers. First-order logic programming does not, however, provide the higher-order representation domain which facilitates the manipulation of logical formulas. λProlog is more expressive in its support for higher-order objects, and in its inclusion of implication and quantification. This promotes its use as a meta-logic for theorem proving. Felty & Miller present this case in more detail [16,15].

Example 2. The previous integration example relied upon λProlog search to solve queries. Additional levels of control (to constrain search) need not, however, interfere with the underlying EBG process! To confirm this, we implemented a tactic-style theorem prover over the same domain. At the bottom level, the rules of integration are each represented as tactics by giving them a name:

```
boxpi  (cf_l     (intgr  (X\ (A * (B X)))   (X\ (A * (Bi X))))
                 (intgr  B                  Bi)).
boxpi  (pw       (intgr   (X\ (expn X A))   (X\ ((expn X (A + 1)) div (A + 1))))
                 true).
```

Tactics perform goal reduction: `cf_l` (for 'constant factor left') reduces the integration task by removing a common factor, while `pw` (for 'power') solves the integration of an exponent.

Tacticals control the application of tactics; in other words, tacticals are *meta*-tactics. The tacticals which we implemented where borrowed from Felty and Miller [16, page 73], and include `then` (composition), `repeat` (iteration), `try` (optional), and `orelse` (alternative). We additionally defined an interactive predicate providing for the incremental application of tactics or tactical combinations of tactics. For example, applying the tactical `(then cf_l pw)` to the query

```
intgr    X\(3 * (expn X 2))    F.
```

yields

```
F  =  X\(3 * (expn X (2 + 1) div (2 + 1)))
```

and the generalization

```
boxpi (intgr    X\(B * expn X C)    X\(B * (expn X (C + 1)) div (C + 1))).
```

For a further discussion of tactic-style theorem proving, see, for example, LCF [17,37] or Nuprl [4].

6 Program Development

One paradigm for formal program development is that of program transformation [3,25,42,14,43]. Under such an approach, an abstract specification of an algorithm is refined, or *specialized*, through a sequence of formal elaboration steps, or *transformations*, into a program with acceptable performance. The resulting sequence of transformations along with the initial specification serve as a *derivation*, or justification, of the optimized program. That this process may be realized within a higher-order logic programming language is supported by Hannan & Miller [19]. And once the paradigm has been formalized within $\lambda^{\square}$Prolog, our EBG algorithm becomes applicable.

Example 3. To substantiate the claim that higher-order EBG is applicable to a program development methodology, we will generalize over a simple transformational system which we have applied to induce tail recursion in certain situations. This example is also treated tentatively in Dietzen & Scherlis [10], among others. From a tail recursive version, an iterative form could easily be derived. It is not our intention that the example be grasped in detail; rather we present it for the interesting generalization that results. Hence we defer a full discussion of the derivation to Appendix A.

We begin with a functional specification of the factorial program:

```
fix Fact\ (lam N\
  (ife (equals N 0)
       1
       (times (appl Fact (minus N 1)) N)))
```

The above is λProlog abstract syntax for a simple functional language. `lam` and `appl` represent explicit λ abstraction and application, respectively; the incorporation of explicit notation provides control over computation (β-reduction), rather than leaving it to λProlog. `fix` is the fixpoint or recursion operator: it is evaluated by substituting `fix`'s body for each occurrence of the bound identifier in that body.

The derivation proceeds by applying transformations, or rewrite rules, to this specification. For example, the following rule replaces an occurrence of `(G X)` with `(Op (G X) A)`, where `A` is a right identity of `Op` (*e.g.*, `Op` = `plus` & `A` = `0`).

```
boxpi (add_oper_rid1  Op   C
                      (C (X\ (G X)))
                      (C (X\ (Op (G X) A)))        :- right_identity Op A).
```

The second argument `C` is a context that determines the particular subexpression to be replaced. The third and forth arguments match the input and output object programs, respectively. For example, the following invocation of the transformation

```
add_oper_rid1  plus  G\(lam X\ (times (G X) (H X)))
                     (lam X\ (times (succ X) (pred X)))
                     Fn
```

instantiates

```
Fn  =  lam X\ (times (plus (succ X) 0) (pred X))
```

The full derivation consists of a sequence of ten such transformation rules and the contexts of their application. This constitutes a *meta-program* — a program to manipulate an object program (*e.g.*, factorial). Such a meta-program could be constructed interactively by alternatively selecting rules and contexts: rules could be chosen from a menu, while contexts would require a more elaborate mechanism, such as *pointing* with a mouse. (We hope to implement an interface to $\lambda^{\square}$Prolog facilitating such interaction.) Although this is ideally how such a meta-program would arise, ours was instead hand-coded. (A full listing appears in Appendix A.) The result of applying the meta-program to `Fact` is the tail recursive expression

```
appl (fix Fact1\ (lam M\ (lam N\
          (ife (equals N 0)
               M
               (appl (appl Fact1 (times N M)) (minus N 1))))))
     1
```

But more interesting is the generalization:

```
boxpi
    (derv Op

          (fix F\ (lam Y\
             (ife (H1 Y)
                  A
                  (Op (appl F (H2 Y)) (H3 Y)))))

          (appl (fix F1\ (lam X\ (lam Y\
                   (ife (H1 Y)
                        X
                        (appl (appl F1 (Op X (H3 Y)) (H2 Y)))))))
                B)

        .:-   right_identity Op B,
              left_identity Op A,
              associative Op).
```

derv is the name of the meta-program. The second argument to derv is the initial specification, while the third argument is the tail recursive output.

The generalization produced by our prototype is not as elegantly expressed: it consists instead of a series of constraint equations. We took the liberty of collapsing them into their 'most obvious' solution above for presentation. The problem of more elegantly displaying these constraints requires further consideration. In either form, however, the generalization may be applied to analogous programs such as list reversal

```
fix Rev\ (lam L\
  (ife (null L)
       nil
       (append (appl Rev (cdr L)) (cons (car L) nil))))
```

yielding the tail-recursive version

```
appl (fix Rev1\ (lam K\ (lam L\
        (ife (null L)
             K
             (appl (appl Rev1 (append (cons (car L) nil) K))
                   (cdr L))))))
      nil
```

The above result requires only the addition of a final simplification to make the reduction from (append (cons (car L) nil) K) to (cons (car L) K). Hence, the generalized fact derivation is sufficient for rev.

7 Conclusions from Higher-order Examples.

First vs. Higher-order. It may be the case that the above examples could be reproduced using a first-order encoding and first-order EBG. However, the additional complexity required to

implement side conditions on variables, substitution, *etc.* seem prohibitive. For such inherently higher-order problems, the designer of a rule-base will demand the expressiveness afforded by higher-order language. Higher-order EBG, then, provides a means by which explanation-based learning can be realized for problem domains formulated within higher-order language.

Learning and Assimilation. We have concentrated on how a rich representation language supports EBG, and have largely ignored questions concerning how these generalizations may be assimilated and applied automatically. Under the traditional approach, the underlying architecture produces and assimilates generalizations in the course of solving each query (at least when learning is 'switched on'). This assimilation may be selective or may involve the forgetting of those derived rules only infrequently referenced. For a discussion of these issues see Prieditis & Mostow [40, pages 496–497], Minton [33], and Donat & Wallen [11].

The calculus integration example of Section 5 reinforces our belief that EBG should be a *feature* of the language rather than a 'black box' within the architecture. Consider that adding rules produced by EBG directly to the rule-base is *not* generally desirable. In this particular example, the client has made a commitment to control the application of integration rules via tactics and tacticals. The incorporation of a derived rules into the program is undesirable and ineffective since it would still remain absent from the tactics. Instead, it is the client that *may* desire to assimilate the generalization as a new, derived tactic. The point is that the client, rather than the architecture, is in a position to control assimilation. This approach stands in sharp contrast to systems such as SOAR in which learning is confined to the underlying architecture [28,41].

Interaction and EBG. Explanation-based generalization is often labeled 'speed-up' learning in that EBG extends the domain theory by constructing new rules in the deductive closure of that domain theory; that is, nothing new may be proven, but the solution of problems covered by the derived rules is (hopefully) quicker. This characterization of EBG is not entirely accurate. Consider the addition of a user to the system, as in the integration and program development examples of Sections 5 & 6. (Actually, in the latter case the user is only hypothetical.) This user selects rules (and, in the latter case, rule contexts) thereby guiding or eliminating the search for a solution. His role is essential for domains in which the search problem is intractable, which is more clearly the case for the tail recursion derivation. The resulting problem solution and generalization, while in the deductive closure of the rule set, are *not* accessible without user guidance. Here EBG becomes a vehicle to *transfer* knowledge from the user to the learner. The combination of learner and user, when viewed as a whole, still only accomplish speed-up learning. But, after a joint derivation of `fact`, the learner could handle `rev` without user assistance. That is, from the individual perspectives of the learner and user, more than speed-up learning has taken place.

8 $\lambda^{\Box}$Prolog and EBG

Logic. $\lambda^{\Box}$Prolog does not distinguish sequences of the modal prefix; that is, $\Box\Box A = \Box A$. For those familiar with the subject, in this respect $\lambda^{\Box}$Prolog is akin to the modal logic *S5* [26]. However, $\lambda^{\Box}$Prolog is at most a subset of *S5* since it lacks negation ($\neg$) and the second modal operator of *possibility* $\Diamond$ or **M**, which may be defined as $\neg\Box\neg$. The difference between possible and contingent truth is conceptually similar to that between contingency and necessity: $\Diamond A$ is to A as A is to $\Box A$.

$\lambda^{\Box}$Prolog could equally have been formulated with unprefixed clauses representing domain theory and clauses prefixed with $\Diamond$ standing for training instance.

The syntax of $\lambda^{\Box}$Prolog is summarized by the following inductively defined classes:

```
 G  ::=  true | A | G1 , G2 | G1 ; G2 | D => G | pi X\ G | sigma X\ G | box BG
BG  ::=  true | A | BG1 , BG2 | pi X\ BG | box BG
 D  ::=  true | A | D1 , D2 | G => D | pi X\ D | box D
```

where G is a goal, BG is a boxed goal, D is a program clause, A ranges over atoms (*i.e.*, predicates with arguments), and X ranges over variables. The construct sigma stands for existential quantification.

Although our examples have only used $\Box$ at the top-level, it is not restricted to outermost occurrences. The use of $\Box$ does not, however, extend to arbitrary λProlog contexts. In particular, $\lambda^{\Box}$Prolog disallows goals of the form box (D => G), box (sigma X\ G), and box (G1 ; G2). This is because it is unclear how to give an operational definition to these constructs. It is also unclear what additional expressiveness would be provided.

The formulation of $\lambda^{\Box}$Prolog we give here is sound and (non-deterministically) complete for an extension of first-order modal Horn clauses to permit higher-order hereditary Harrop formulas [31] (*i.e.*, λProlog) in unboxed contexts. This can be proven by a combination of the methods of del Cerro [6,7] and Miller, *et al.* [30]. Without giving a precise definition here, we conjecture a similar result for a more general formulation in which $\Box$ may be applied to arbitrary λProlog constructs.

The Barcan Formula. The Barcan formula is as follows:

$$\forall x.\, \Box\, P(x) \;\Rightarrow\; \Box\, \forall x.\, P(x)$$

In $\lambda^{\Box}$Prolog this is equivalent to

```
pi P\ (pi X\ (box (P X))  =>  box (pi X\ (P X)))
```

While the converse of Barcan — that is,

$$\Box\, \forall x.\, P(x) \;\Rightarrow\; \forall x.\, \Box\, P(x)$$

is true in all modal logics, the validity of Barcan varies. It seemed most natural to include Barcan within $\lambda^{\Box}$Prolog.

However, the generalization algorithm differentiates between the left and right side of Barcan; that is, the relative order of box and pi, while not affecting provability, *can* affect generalization. In particular, variables whose universal quantifiers are outside of box are not generalized. For example, should a proof using the following clause

```
pi Z\ (box (weapon Z  :-  gun Z)).
```

instantiate Z, Z will be correspondingly instantiated within the generalization. That is, by nesting box within pi, the generalization may be restricted. Inserting the above clause into the suicide example of Section 2 yields

```
wsolve true        :- !.
wsolve (G1 , G2)   :- !, wsolve G1, wsolve G2.
wsolve (G1 ; G2)   :- !, (wsolve G1; wsolve G2).
wsolve (D => G)    :- !, hyp D => wsolve G.
wsolve (pi G)      :- !, pi X\ (wsolve (G X)).
wsolve (sigma G)   :- !, wsolve (G T).
wsolve (box G)     :- !, ssolve G.
wsolve Ga          :- !, hyp D, wmatch D Ga SG, wsolve SG.

ssolve true        :- !.
ssolve (G1 , G2)   :- !, ssolve G1, ssolve G2.
ssolve (pi G)      :- !, pi X\ (ssolve (G X)).
ssolve (box G)     :- !, ssolve G.
ssolve Ga          :- !, hyp D, smatch D Ga SG, wsolve SG.
```

Figure 7: Meta-interpreter without EBG: goal analysis.

```
boxpi (kill Y Y :- depressed Y, buy Y obj1, gun obj1).
```

This difference in treatment provides for greater expressiveness without sacrificing power. It is also possible to revise the implementation so that the EBG behaves identically for either side of Barcan, but we do not illustrate that version herein.

Implementation. It is important to distinguish the programming language $\lambda^{\Box}$Prolog from the underlying architecture which potentially produces generalizations of $\lambda^{\Box}$Prolog computation. To simplify the development of EBG within $\lambda^{\Box}$Prolog, we first present, in Section 9, a basic interpreter for $\lambda^{\Box}$Prolog without the generalizing component. That interpreter is written in λProlog. Due to the closeness of the correspondence between object language ($\lambda^{\Box}$Prolog) and meta-language (λProlog), we frequently refer to the interpreter as a meta-interpreter. This meta-interpreter is extended to perform EBG within a second prototype in Section 10. The expanded meta-interpreter exemplifies the generalization algorithm underlying $\lambda^{\Box}$Prolog, and has reproduced first-order examples from the literature (Section 2), as well as new higher-order examples (Sections 3,5 & 6). The full meta-interpreter may be found in Appendix C.

9 Implementing $\lambda^{\Box}$Prolog

In our discussion, we shall use `G` and `D` for arbitrary goal formulas and program clauses, respectively. The full program, or rule-base, is comprised by the $\lambda^{\Box}$Prolog clauses of the domain theory and training instance. Prior to invoking the meta-interpreter, each `D` has been asserted as a hypothesis via `hyp D`. This allows the meta-interpreter to enumerate the rules with λProlog's backtracking search, although obviously the performance of such an approach suffers in comparison with the hashing schemes employed by more low-level Prolog implementations.

The `solve` predicates of Figure 7 are responsible for proving a given goal `G`. Goal solution is divided between two sets of clauses: `wsolve` for 'weak solve' and `ssolve` for 'strong solve.' This distinction

```
wmatch (D1 , D2)   Ga  SG          :- !, (wmatch D1 Ga SG; wmatch D2 Ga SG).
wmatch (G => D)    Ga  (G, SG)     :- !, wmatch D Ga SG.
wmatch (pi D)      Ga  SG          :- !, wmatch (D Y) Ga SG.
wmatch (box D)     Ga  SG          :- !, wmatch D Ga SG.
wmatch Da          Ga  true        :- !, Da = Ga.

smatch (D1 , D2)   Ga  SG          :- !, (smatch D1 Ga SG; smatch D2 Ga SG).
smatch (G => D)    Ga  (G, SG)     :- !, smatch D Ga SG.
smatch (pi D)      Ga  SG          :- !, smatch (D Y) Ga SG.
smatch (box D)     Ga  (box SG)    :- !, wmatch D Ga SG.
```

Figure 8: Meta-interpreter without EBG: clause analysis.

arises from the more stringent proof required by the necessary truth of boxed goals: from p we cannot derive box p, but p does follow from box p.

Within `solve`, the solution of $\lambda^{\Box}$Prolog goals is largely realized by the corresponding λProlog constructs. For example, a $\lambda^{\Box}$Prolog conjunction (`G1 , G2`) is derived by establishing the λProlog conjunction of `G1` and `G2`, while a universally quantified $\lambda^{\Box}$Prolog goal is universally derived under λProlog. Similarly, an implicational goal (`D => G`), or equivalently (`G :- D`), is proven by first assuming `D`, and then attempting to derive `G`. Such sharing between object language ($\lambda^{\Box}$Prolog) and meta-language (λProlog) makes for elegant interpretation. The rules of `ssolve` do not address the range of λProlog constructs because additional restrictions are placed upon boxed goals (see Section 8).

The final clauses of `wsolve` and `ssolve` select a potentially pertinent clause `D` from the program, which the `match` predicates subsequently attempt to apply in the proof of `G`. At the point of invoking `match`, `G` has been reduced to an atomic goal, *i.e.*, a predicate followed by some number of arguments. As we shall see, `match` may produce a subgoal (represented by `SG`) that must then be solved to complete the proof.

The two `match` predicates of Figure 8 analyze the program clause `D` to find a sufficient condition for the pending atomic goal `Ga`. For a conjunction `(D1 , D2) => Ga`, the logic programming paradigm requires that either `D1` or `D2` individually derives `Ga`. If `D` is an implication (`G' => D'`), we conjoin `G'` with the subgoals that arise from establishing that `D'` implies `Ga`. For example, the goal `(weapon obj1)` matches the program `(weapon Z :- gun Z)` generating the subgoal `(gun obj1)`. A universally quantified clause `(pi D)` is reduced by replacing the bound variable with a new logical variable (in logic programming terminology) that may later be instantiated in the course of the proof. When `smatch` encounters a `box` in the program, the nested clause need only be weakly matched with the current goal. This is because proving a goal 'strongly' simply means that any utilized clauses must themselves be necessarily true.

In the final clause of `wmatch`, the unification of an atomic `Da` and atomic `Ga` is attempted. This is analogous to the unification of a goal and clause head within Prolog interpretations (*e.g.*, `weapon obj1` and `weapon Z`). If successful, this has the effect of 'returning' the accumulated conjunction of subgoals to the last clause in `wsolve` or `ssolve`, which will then solve `SG` recursively. The predicate `smatch` is, however, missing the analogue to the last clause of `wmatch`. This

```
wsolve true        true          true          :- !.
wsolve (G1 , G2)   (GG1 , GG2)   (DD1 , DD2)   :- !, wsolve G1 GG1 DD1, wsolve G2 GG2 DD2.
wsolve (G1 ; G2)   (GG1 ; GG2)   DD            :- !, (wsolve G1 GG1 DD; wsolve G2 GG2 DD).
wsolve (D => G)    (DD1 => GG)   DD2           :- !, ghyp D DD1 => wsolve G GG DD2.
wsolve (pi G)      (pi GG)       (DD X)        :- !, pi X\ (wsolve (G X) (GG X) (DD X)).
wsolve (sigma G)   (sigma GG)    (DD T)        :- !, wsolve (G T) (GG T) (DD T).
wsolve (box G)     (box GG)      DD            :- !, ssolve G GG DD.
wsolve Ga          GGa           DD1           :-    ghyp D DD2,
                                                   wmatch D Ga DD2 GGa MG,
                                                   meta_wsolve MG DD1.
wsolve Ga          GGa           (DD1 , DD2)   :- !, hyp D,
                                                   wmatch D Ga DD1 GGa MG,
                                                   meta_wsolve MG DD2.

ssolve true        true          true          :- !.
ssolve (G1 , G2)   (GG1 , GG2)   (DD1 , DD2)   :- !, ssolve G1 GG1 DD1, ssolve G2 GG2 DD2.
ssolve (pi G)      (pi GG)       (DD X)        :- !, pi X\ (ssolve (G X) (GG X) (DD X)).
ssolve (box G)     (box GG)      DD            :- !, ssolve G GG DD.
ssolve Ga          GGa           DD1           :-    ghyp D DD2,
                                                   smatch D Ga DD2 GGa MG,
                                                   meta_wsolve MG DD1.
ssolve Ga          GGa           (DD1 , DD2)   :- !, hyp D,
                                                   smatch D Ga DD1 GGa MG,
                                                   meta_wsolve MG DD2.
```

Figure 9: Generalizing meta-interpreter: goal analysis.

---

is because an unboxed atomic clause cannot be used to prove a boxed atomic goal; that is p is not sufficient to derive box p. As λProlog does not admit disjunction and existential quantification in programs [36], neither does $\lambda^{\Box}$Prolog.

10 Implementing EBG

The $\lambda^{\Box}$Prolog meta-interpreter of Section 9 may be extended to perform EBG. As in the first-order approach of Kedar-Cabelli and McCarty [27], our generalizing meta-interpreter develops two parallel proofs simultaneously: a specific proof of G and a generalized proof of GG. These proofs are not explicitly constructed; rather they are implicit in the λProlog search. In the course of proving G and GG, the implementation accumulates the conjunction of generalized clauses DD sufficient to establish GG – that is, the leaves of the generalized proof. Thus the generalizing solve predicates accept three arguments – the goal G (instantiated), the generalized goal GG (uninstantiated), and the conjunction of generalized clauses DD (uninstantiated). The resulting explanation-based generalization is then boxpi (DD => GG).

In the extended wsolve and ssolve of Figure 9, the decomposition of G guides the corresponding instantiation of the generalized goal GG. It is only at the atomic level — the predicates nested within the $\lambda^{\Box}$Prolog operations — where G and GG diverge. Solving an implicational goal requires a new predicate ghyp (for 'generalized hyp') because it is no longer sufficient to simply recall D for

later proof: the associated generalization DD must be retained as well. ghyp then appears parallel to hyp at the end of the solve predicates to recall D and DD for matching. (The DD2 argument of ghyp need not be included in the generalized clause to be returned at this level as it has already been incorporated higher in the recursion.)

The MG's (for *meta-goals*) in the final clauses of solve assume a role analogous to that of the subgoal SG in the previous meta-interpreter — that is, MG retains subproof tasks for later derivation. The transition from the subgoals of the first interpreter to the current meta-subgoals is due to the need to retain both G and GG. This is accomplished by the predicate gsolve, or 'generalized solve.' The straight-forward clauses meta_wsolve and meta_ssolve for meta-goal solution may be found in Section C.

When solve selects a clause D from the program to prove an atomic Ga, the match predicates of Figure 10 yield a generalized atom GGa and the generalized clause DD sufficient to derive GGa. At the atomic level where Da is unified with Ga (analogous to the last clause of the original wmatch), DDa is instead unified with GGa. That neither the pair Ga and GGa nor the pair Da and DDa are unified at the atomic level is essential for generalization: DD and GG need only be instantiated to the point that GG necessarily follows from DD. How then do any of the constants of D (first or higher-order) ever end up in DD? The answer is that unless some of the D's employed in the proof are boxed, none ever will. The following degenerate generalization results from running the ubiquitous suicide example with all clauses training instance:

```
boxpi (GG  :-  (GG  :- GG1, GG2, GG3),
               (GG1 :- GG4),
               (GG2 :- GG5),
               (GG3 :- GG6),
               GG4,
               GG5,
               GG6)
```

Unrestricted higher-order EBG, in which the entire program is training instance, is so overgeneral as to be uninteresting: each goal is simply abstracted with a variable. Of course, the derived rule is still valid, but cannot be used in a very directed way. (For an approach that tries to exploit similar, very general higher-order rules see Donat & Wallen [11].)

Generalization is suppressed when matching boxed D's by explicitly unifying D and DD in the invocation of smatch (for 'boxed match'). Unlike wmatch and smatch, within bmatch DD is instantiated, initially to the same value as D. However, distinct logical variables X and Y are substituted within universally quantified programs because D will subsequently be unified with Ga and DD with GGa. (The type declaration ':A' insures that the substituted variables have the same type. This is important in restricting higher-order unification.) Thus through universal quantification, D and DD may again diverge.

As both boxed and unboxed clauses are used in the proofs we have developed, the reader might rightfully expect both to appear in DD, the resulting sufficient conditions of the generalization. However, boxed clauses are necessarily true, and hence need not be re-checked in the application of a derived rule. Instead, it is the conjunction of utilized unboxed clauses which constitute the simplest expression of the sufficient conditions for GG. Removing boxed clauses from DD requires a simple reduction predicate, whose definition may again be found in Section C. It is more general to remove boxed clauses from the completed generalization than to avoid their initial incorporation, since only

```
wmatch (D1 , D2)  Ga  DD           GGa  MG :- !, (wmatch D1 Ga DD GGa MG;
                                                  wmatch D2 Ga DD GGa MG).
wmatch (G => D)   Ga  (GG => DD)   GGa  (gsolve G GG, MG)
                                           :- !, wmatch D Ga DD GGa MG.
wmatch (pi D)     Ga  DD           GGa  MG :- !, wmatch (D X) Ga DD GGa MG.
wmatch (box D)    Ga  (box D)      GGa  MG :- !, bmatch D Ga D GGa MG.
wmatch Ga         Ga  GGa          GGa  true.


smatch (D1 , D2)  Ga  DD           GGa  MG :- !, (smatch D1 Ga DD GGa MG;
                                                  smatch D2 Ga DD GGa MG).
smatch (G => D)   Ga  (GG => DD)   GGa  (gsolve G GG, MG)
                                           :- !, smatch D Ga DD GGa MG.
smatch (pi D)     Ga  DD           GGa  MG :- !, smatch (D X) Ga DD GGa MG.
smatch (box D)    Ga  (box D)      GGa  (box MG)
                                           :- !, bmatch D Ga D GGa MG.


bmatch (D1 , D2)  Ga  (DD1 , DD2)  GGa  MG :- !, (bmatch D1 Ga DD1 GGa MG;
                                                  bmatch D2 Ga DD2 GGa MG).
bmatch (G => D)   Ga  (GG => DD)   GGa  (gsolve G GG, MG)
                                           :- !, bmatch D Ga DD GGa MG.
bmatch (pi D)     Ga  (pi DD)      GGa  MG :- !, bmatch (D  X:A) Ga
                                                        (DD Y:A) GGa MG.
bmatch (box D)    Ga  (box D)      GGa  MG :- !, bmatch D Ga D GGa MG.
bmatch Ga         Ga  GGa          GGa  true.
```

Figure 10: Generalizing meta-interpreter: clause analysis.

top-level boxed clauses could reasonably be recognized in `solve`. These simplification predicates will unavoidably destroy degenerate generalizations such as the example above, something which will be addressed in future, more efficient implementations (see Section 11).

11 Weaknesses of the Implementation

Direct Interpretation. The current $\lambda^{\square}$Prolog implementation in λProlog has been extremely valuable for experimenting with different variations of $\lambda^{\square}$Prolog and the EBG algorithm. It is, however, extremely slow due to the additional level of interpretation, which also precludes the application of λProlog optimizations (such as hashing rules based upon predicate names). Furthermore, the meta-interpreter is not powerful enough to handle λProlog primitives (*e.g.*, cut or arithmetic), or to realize the `boxpi` notational convention.[2] Moreover, the prototype generalizing meta-interpreter contains a deficiency in its application of higher-order unification that can only be addressed at the level of the λProlog implementation. Higher-order unification underlies λProlog and its EBG extension. The prototype relies on parallel unifications between `D` and `G` and between `DD` and `GG`. However, recall that higher-order unification is nondeterministic. It is thus not sufficient to enforce that two unifications occur; rather, the unifications themselves must correspond — that is, represent an analogous nondeterministic choice.[3] It is not possible to address this problem within the meta-interpreter since λProlog does not permit control over the underlying unification. The generalizations that otherwise result, although not what the user expects, are still valid. None of the examples herein employ the nondeterministic unification required to exhibit this behavior.

We plan to address the above weaknesses by extending an existing λProlog interpreter, eLP, to realize $\lambda^{\square}$Prolog and EBG. eLP is based in COMMON LISP, and was developed at CMU by Conal Elliott and Frank Pfenning [12]. It is available free of charge (send mail to elp-request@cs.cmu.edu on the Internet for more information) and includes all the examples in this report.

Interfacing $\lambda^{\square}$Prolog and EBG. The prototype produces a generalization in association with each solved query. To realize learning within the system, we need simply assimilate (*i.e.*, add to the program) each of these resulting generalizations. The problem with such an approach is that it confines learning to the architecture, and therefore precludes client control (Section 7).

We suggest, instead, that the programming language $\lambda^{\square}$Prolog be extended with primitives for controlling learning. By providing the programmer with an explicit means to address generalization and assimilation, we defer the difficult problem of determining when to generalize and assimilate; that is, generalization and assimilation are provided as features of the language rather than as aspects of the architecture. Clients have the advantage of bringing domain knowledge and user interaction to bear in determining what is to be learned. Such explicit control requires that $\lambda^{\square}$Prolog include primitives to (1) produce a generalization (the extra cost involved suggests that application be selective), to (2) access the resulting generalization, and to (3) dynamically extend the program with a derived rule (perhaps modified by the client).

---

[2] In fact `boxpi` represents more than just a convenience. The problem is that the generalizations produced by our prototype contain variables that must be universally quantified before application. Consider the derived rule `p X. p X => (p a , p b)` is not true since in the course of proving `(p a)`, `X` is instantiated to `a`. Of course, `pi X\(p X) => (p a , p b)` is true because the universal quantification allows `X` to be multiply instantiated. λProlog provides no mechanism by which existing free variables (such as `X` above) can be captured with an inserted quantifier.

[3] We are grateful for Masami Hagiya for this observation.

12 Future Work

Further experimentation. Our method of generalization is independent of the depth-first search strategy of the underlying logic programming language: as we have shown, it also applies to heuristic search paradigms or user-guided deduction. More general approaches to search facilitate application of our algorithm to more difficult problem solving domains. We plan to apply our techniques to larger domains, particularly program derivation [10,19] and theorem proving [16], as well as hybrids of the two [38]. Much of our original motivation for the work reported here comes from these areas. Such experimentation should be substantially facilitated by the new implementation suggested in Section 11.

Dynamic operationality. Hirsh introduces *dynamic* operationality criteria, which allow the operationality of goals to be defined and redefined within the computational framework [22]. In Sections 2 & 4, we illustrated our alternative realization of operationality based on the separation of domain theory and training instance. This partitioning of the rule-base need not be static; instead, we plan to investigate a dynamic box predicate over program clauses analogous to dynamic operationality over goals. The expressive power of dynamic operationality is afforded by this dynamic box, which may shift domain theory rules to training instance. (The reverse direction is dangerous; see Section 4.)

Additional modal operators. Other modal operators (such as 'knows') can be added to the language in a sound and complete way (following [6]) while still preserving its basic character as a logic programming language. Would such a more general language admit EBG?

Parameterized modal operators such as knows also introduce the additional complexity of generalizing over modal functions: when the 'knower' is also part of the object language (for example, `knows a (hates b a)`), we would like to generalize over occurrences in the modal function as well as those in the object language.

A Tail Recursion via Program Transformation

As it may be of interest to the less casual reader, this section includes the details of the transformational derivation summarized in Section 6. The following development assumes that the functional object language is side-effect free. This restriction allows individual transformations to preserve correctness in the weak sense (ignoring termination), although we have not attempted to prove either weak or strong (termination preserving) correctness. Figure 11 lists the necessary higher-order transformations, while Figure 12 contains a specific meta-program for applying these rules. For example, the transformation `dist_ife_2` distributes a binary function `F` over an *if*-statement. An associated step in the meta-program applies `dist_ife_2` by specifying `F` and a particular context `C` in which an *if* is currently nested within `F`. In general, the higher-order context variable `C` indicates the position in the program where the rule is to be applied. Without such contexts, the application of a single transformation can be highly nondeterministic. The bound variables (*e.g.*, `X` and `Y` in `dist_ife_2`) that appear directly within some contexts are necessary for higher-order matching: because certain parameters are not free within the program to be matched, higher-order unification requires that they be bound within the transformation as well. The numerical suffixes given to rules (*e.g.*, 2 of `dist_ife_2`) indicate the number of bound variables within the context. Although rules are only defined with the quantity of context parameters necessary for this derivation, one may envision a family of rules for each transformation.[4]

Hand-coding a meta-program is a tedious and error prone process. We wish to emphasize that we do not advocate the unaided construction of meta-programs as an attractive means for developing programs. However, we do claim these meta-programs could be the output of an interactive tool for the selection of transformations and their associated contexts. Assuming the plausibility of such interaction, $\lambda^{\square}$Prolog provides an elegant means to generalize user-guided transformational development of programs or proofs.

The application of the rules and meta-program (*i.e.*, the domain theory) additionally requires the following training instance:

```
associative times.
left_identity  times 1.
right_identity times 1.
```

We now enumerate the individual steps in the derivation of the tail recursive factorial as dictated by the transformations and meta-program (Figures 11 & 12, respectively).

0. The initial definition.

```
fix Fact\ (lam N\ (ife (equals N 0)
                       1
                       (times (appl Fact (minus N 1)) N)))
```

[4]That multiple versions of rules are required is a drawback of λProlog, and could potentially hinder generalization in that conceptually equivalent rules require multiple expression. If λProlog provided a product type, this problem could be addressed using a product of logical variables represented as a single argument as shown in [39].

1. η-expand term; that is, insert a λ and application. ('...' elides the body of `fact`.)

```
lam N\ (appl (fix Fact\ ...)
             N)
```

2. Insert a multiplication by 1.

```
lam N\ (times (appl (fix Fact\ ...)
                    N)
              1)
```

3. Abstract over the argument 1; that is, make 1 a parameter. This introduces a second argument which is to become the accumulator in the eventual tail recursive version.

```
appl (lam M\ (lam N\ (times (appl (fix Fact\ ...)
                                  N)
                            M)))
     1
```

4. Name the resulting two argument function `Fnew` — Since `fix` specifies the expansion of recursive functions, one may think of it a mechanism for function definition in general. Such a method avoids the global binding and recall of function names and bodies. This initial definition of `Fnew` will be used later in the derivation.

```
appl (fix Fnew\ (lam M\ (lam N\ (times (appl (fix Fact\ ...)
                                             N)
                                       M))))
     1
```

5. *Unfold* the recursive definition of `Fact`; that is, expand the fixpoint operator once.

```
appl (fix Fnew\ (lam M\ (lam N\
       (times (appl
                (lam N1\
                   (ife (equals N1 0)
                        1
                        (times (appl (fix Fact\ ...)
                                     (minus N1 1))
                               N1)))
                N)
              M))))
     1
```

6. β-reduction — `(appl (lam N1\ (G N1)) N)` $\longrightarrow_{beta}$ `(G N)`.

```
appl (fix Fnew\ (lam M\ (lam N\
       (times (ife (equals N 0)
                   1
                   (times (appl (fix Fact\ ...)
                                (minus N 1))
                          N))
              M))))
     1
```

7. Distribute `times` over the *if-then-else*.

```
appl (fix Fnew\ (lam M\ (lam N\
        (ife (equals N 0)
             (times 1 M)
             (times (times (appl (fix Fact\ ...)
                                 (minus N 1))
                           N))
                    M))))
     1
```

8. Simplify the *then*-clause using the fact that 1 is the left identity of `times`.

```
appl (fix Fnew\ (lam M\ (lam N\
        (ife (equals N 0)
             M
             (times (times (appl (fix Fact\ ...)
                                 (minus N 1))
                           N))
                    M))))
     1
```

9. Reassociate the multiplicative expression of the *else*-clause.

```
appl (fix Fnew\ (lam M\ (lam N\
        (ife (equals N 0)
             M
             (times (appl (fix Fact\ ...)
                          (minus N 1))
                    (times N M))))))
     1
```

10. Observe that

```
(times (appl (fix Fact\ ...)
             (minus N 1))
       (times N M))
```

within step 9 is an instance of the original definition of **Fnew** given in step 4:

```
(fix Fnew\ (lam M\ (lam N\ (times (appl (fix Fact\ ...)
                                        N)
                                  M))))
```

The only difference is the values of the arguments M and N. This means that we may *fold* the expression into an Fnew invocation.

```
appl (fix Fnew\ (lam M\ (lam N\
        (ife (equals N 0)
             M
             (appl (appl Fnew (times N M)) (minus N 1))))))
     1
```

This completes the derivation.

```
boxpi (insert_lam     C
        (C (fix F\ (lam M\ (G F M))))
        (C (lam M1\ (appl (fix F\ (lam M\ (G F M))) M1)))).

boxpi (add_oper_rid1  Op   C
        (C (X\ (G X)))
        (C (X\ (Op (G X) A)))                    :- right_identity Op A).

boxpi (abstract_arg   Op   C1   C2
        (C1 (C2 A))
        (C1 (appl (lam M\ (C2 M)) A))).

boxpi (name_fn        C
        (C G)
        (C (fix Fnew\ G))).

boxpi (unfold         C
        (C (fix F\ (G F)))
        (C (G (fix F\ (G F))))).

boxpi (reduce_1       C
        (C (X\ (appl (lam M\ (G M)) X)))
        (C (X\ (G X)))).

boxpi (dist_ife_2     Op   C
        (C  (X\Y\ (Op (ife (Bool X Y) (E1 X Y) (E2 X Y)) (H X Y))))
        (C  (X\Y\ (ife (Bool X Y) (Op (E1 X Y) (H X Y))
                                  (Op (E2 X Y) (H X Y)))))).

boxpi (left_id_2      Op   C
        (C  (X\Y\ (Op A (H X Y))))
        (C  (X\Y\ (H X Y)))                      :- left_identity Op A).

boxpi (assoc_2        Op   C
        (C  (X\Y\ (Op (Op (H1 X Y) (H2 X Y)) (H3 X Y))))
        (C  (X\Y\ (Op (H1 X Y) (Op (H2 X Y) (H3 X Y)))))
                                                 :- associative Op A).

boxpi (fold_two_3     C1   C2   C3
        (C2 (fix F\ (lam M\ (lam N\ (C3 G M N)))))
        (C1 (F\X\Y\ (C3 G (H1 X Y) (H2 X Y))))
        (C1 (F\X\Y\ (appl (appl F (H2 X Y)) (H1 X Y))))).
```

Figure 11: Transformation Rules

```
boxpi (derv  Op F0 F10  :-

  insert_lam          G\G
                      F0 F1,

  add_oper_rid1       Op
                      G\(lam M\ (G M))
                      F1 F2,

  abstract_arg        Op
                      G\G
                      G\(lam M\ (Op (W0 M) G))
                      F2 F3,

  name_fn             G\(appl G W)
                      F3 F4,

  unfold              G\(appl (fix F1\ (lam M\ (lam N\
                            (Op (appl G N) M)))) W)
                      F4 F5,

  reduce_1            G\(appl (fix F1\ (lam M\ (lam N\ (Op (G N) M)))) W)
                      F5 F6,

  dist_ife_2          Op
                      G\(appl (fix F1\ (lam M\ (lam N\ (G M N)))) W)
                      F6 F7,

  left_id_2           Op
                      G\(appl (fix F1\ (lam M\ (lam N\
                          (ife (W1 M N) (G M N) (W3 M N))))) W)

                      F7 F8,

  assoc_2             Op
                      G\(appl (fix F1\ (lam M\ (lam N\
                          (ife (W1 M N) (W2 M N) (G M N))))) W)
                      F8 F9,

  fold_two_3          G\(appl (fix F1\ (lam M\ (lam N\
                          (ife (W1 M N) (W2 M N) (G F1 M N))))) W)
                      G\(appl G W)
                      G\H1\H2\(Op (appl G H1) H2)
                      F4 F9 F10)
```

Figure 12: Meta-program

```
wsolve Ga        GGa        DD        :-    ghyp D DD1,
                                            wmatch D Ga DD1 GGa MG,
                                            meta_wsolve MG DD2,
                                            ((oper Ga, DD = GGa)
                                            ; DD = DD2).
wsolve Ga        GGa        DD        :- !, hyp D,
                                            wmatch D Ga DD1 GGa MG,
                                            meta_wsolve MG DD2,
                                            ((oper Ga, DD = GGa)
                                            ; DD = (DD1 , DD2)).
```

Figure 13: New clauses

B Implementing Operationality

Although we have replaced the traditional notion of operationality, for those who prefer the former approach, we illustrate its realization within our prototype. $\lambda^{\Box}$Prolog supports both dynamic and goal-based operationality criteria within the same uniform framework of higher-order logic. (This is similarly possible in Prolog through its meta-programming facilities [22,23].) Incorporating operationality into our implementation requires providing the meta-interpreter with access to an operationality predicate. The revision, which is illustrated in Figure 13, involves the last two clauses of the `wsolve` predicate; an analogous change is necessary in `ssolve`. The computation proceeds in the same manner, but subgoals encountered in the course of the proof of operational goals are simply *not* incorporated in `DD`. It is then the client's responsibility to specify the computation necessary to determine `oper` of particular goals. Should no clauses be provided for `oper`, the above implementation behaves in the same manner as that given previously.

C Generalizing interpreter for $\lambda^{\Box}$Prolog

For the sake of completeness, we list the unabridged λProlog implementation discussed in Sections 9 through B.

```
module metaebg.


wsolve true        true          true          :- !.
wsolve (G1 , G2)   (GG1 , GG2)   (DD1 , DD2)   :- !, wsolve G1 GG1 DD1,
                                                    wsolve G2 GG2 DD2.
wsolve (G1 ; G2)   (GG1 ; GG2)   DD            :- !, (wsolve G1 GG1 DD;
                                                     wsolve G2 GG2 DD).
wsolve (D => G)    (DD => GG)    DD1           :- !, ghyp D DD => wsolve G GG DD1.
wsolve (pi G)      (pi GG)       (DD X)        :- !, pi X\
                                                     (wsolve (G X) (GG X) (DD X)).
wsolve (sigma G)   (sigma GG)    (DD T)        :- !, wsolve (G T) (GG T) (DD T).
wsolve (box G)     (box GG)      DD            :- !, ssolve G GG DD.
wsolve Ga          GGa           DD1           :-    ghyp D DD,
                                                    wmatch D Ga DD GGa MG,
                                                    meta_wsolve MG DD1.
wsolve Ga          GGa           (DD , DD1)    :- !, hyp D,
                                                    wmatch D Ga DD GGa MG,
                                                    meta_wsolve MG DD1.


ssolve true        true          true          :- !.
ssolve (G1 , G2)   (GG1 , GG2)   (DD1 , DD2)   :- !, ssolve G1 GG1 DD1,
                                                    ssolve G2 GG2 DD2.
ssolve (pi G)      (pi GG)       (DD X)        :- !, pi X\
                                                     (ssolve (G X) (GG X) (DD X)).
ssolve (box G)     (box GG)      DD            :- !, ssolve G GG DD.
ssolve Ga          GGa           DD1           :-    ghyp D DD,
                                                    smatch D Ga DD GGa MG,
                                                    meta_wsolve MG DD1.
ssolve Ga          GGa           (DD , DD1)    :- !, hyp D,
                                                    smatch D Ga DD GGa MG,
                                                    meta_wsolve MG DD1.

ssolve (G1 ; G2)   (GG1 ; GG2)   DD
       :- !, error (writesans "Illegal disjunction in boxed goal").
ssolve (D => G)    (DD1 => GG)   DD2
       :- !, error (writesans "Illegal implication in boxed goal").
ssolve (sigma G)   (sigma GG)    (DD T)
       :- !, error (writesans "Illegal existential in boxed goal").
```

```
wmatch (D1 , D2)  Ga  DD          GGa  MG :- !, (wmatch D1 Ga DD GGa MG;
                                                wmatch D2 Ga DD GGa MG).
wmatch (G => D)   Ga  (GG => DD)  GGa  (gsolve G GG, MG)
                                          :- !, wmatch D Ga DD GGa MG.
wmatch (pi D)     Ga  DD          GGa  MG :- !, wmatch (D X) Ga DD GGa MG.
wmatch (box D)    Ga  (box D)     GGa  MG :- !, bmatch D Ga D GGa MG.
wmatch Ga         Ga  GGa         GGa  true.
```

```
smatch (D1 , D2)  Ga  DD          GGa  MG :- !, (smatch D1 Ga DD GGa MG;
                                                smatch D2 Ga DD GGa MG).
smatch (G => D)   Ga  (GG => DD)  GGa  (gsolve G GG, MG)
                                          :- !, smatch D Ga DD GGa MG.
smatch (pi D)     Ga  DD          GGa  MG :- !, smatch (D X) Ga DD GGa MG.
smatch (box D)    Ga  (box D)     GGa  (box MG)
                                          :- !, bmatch D Ga D GGa MG.
```

```
bmatch (D1 , D2)  Ga  (DD1 , DD2)  GGa  MG :- !, (bmatch D1 Ga DD1 GGa MG;
                                                  bmatch D2 Ga DD2 GGa MG).
bmatch (G => D)   Ga  (GG => DD)   GGa  (gsolve G GG, MG)
                                           :- !, bmatch D Ga DD GGa MG.
bmatch (pi D)     Ga  (pi DD)      GGa  MG :- !, bmatch (D  X:A) Ga
                                                        (DD Y:A) GGa MG.
bmatch (box D)    Ga  (box D)      GGa  MG :- !, bmatch D Ga D GGa MG.
bmatch Ga         Ga  GGa          GGa  true.
```

```
wmatch (D1 ; D2)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal disjunction in program").
wmatch (sigma D)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal existential in program").
```

```
smatch (D1 ; D2)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal disjunction in program").
smatch (sigma D)  Ga  DD       GGa  MG
    :- !, error (writesans "Illegal existential in program").
```

```
bmatch (D1 ; D2)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal disjunction in program").
bmatch (sigma D)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal existential in program").
```

```
   % Meta-interpreter invocation.

dosolve  G GG DD :- !, wsolve G GG DD1, breduce DD1 DD2, reduce DD2 DD.



    % Solution of accumulated Meta-goals.

meta_wsolve true            true         :- !.
meta_wsolve (MG1 , MG2)     (DD1 , DD2)  :- !, meta_wsolve MG1 DD1,
                                               meta_wsolve MG2 DD2.
meta_wsolve (box MG)        DD           :- !, meta_ssolve MG DD.
meta_wsolve (gsolve G GG)   DD           :- !, wsolve G GG DD.



meta_ssolve true            true         :- !.
meta_ssolve (MG1 , MG2)     (DD1 , DD2)  :- !, meta_ssolve MG1 DD1,
                                               meta_ssolve MG2 DD2.
meta_ssolve (box MG)        DD           :- !, meta_ssolve MG DD.
meta_ssolve (gsolve G GG)   DD           :- !, ssolve G GG DD.
```

```
    % Replaces "(box H)" with "true" in DD --- the set of sufficient
    % conditions.

breduce  true        true          :- !.
    % Above should be first to avoid infinite recursion on logical variables.
    % This allows uninstaniated variables to be 'reduced' out of the picture.
    % (Good for all but degenerate higher-order generalizations.)
breduce  (H1 , H2)   (H1i , H2i)   :- !, breduce H1 H1i, breduce H2 H2i.
breduce  (H1 ; H2)   (H1i ; H2i)   :- !, breduce H1 H1i, breduce H2 H2i.
breduce  (H1 => H2)  (H1i => H2i)  :- !, breduce H1 H1i, breduce H2 H2i.
breduce  (pi H)      (pi Hi)       :- !, pi X\ (breduce (H X) (Hi X)).
breduce  (sigma H)   (sigma Hi)    :- !, pi X\ (breduce (H X) (Hi X)).
breduce  (box H)     true          :- !.
breduce  Ha          Ha            :- !.



    % Simplifies sufficient conditions by removing superfluous true's.

reduce   true          true          :- !.
    % Above should be first to avoid infinite recursion on logical variables.
    % This allows uninstaniated variables to be 'reduced' out of the picture.
    % (Good for all but degenerate higher-order generalizations.)

reduce   (H1 , H2)   H             :- !, reduce H1 H1i, reduce H2 H2i,
                                         reduce1 (H1i , H2i) H.
reduce   (H1 ; H2)   H             :- !, reduce H1 H1i, reduce H2 H2i,
                                         reduce1 (H1i ; H2i) H.
reduce   (H1 => H2)  H             :- !, reduce H1 H1i, reduce H2 H2i,
                                         reduce1 (H1i => H2i) H.
reduce   (pi H1)     H             :- !, pi X\ (reduce (H1 X) (H1i X)),
                                         reduce1 (pi H1i) H.
reduce   (sigma H1)  H             :- !, pi X\ (reduce (H1 X) (H1i X)),
                                         reduce1 (sigma H1i) H.
reduce   (box H1)    H             :- !, reduce H1 H1i,
                                         reduce1 (box H1i) H.
reduce   Ha          Ha            :- !.


reduce1   true              true    :- !.
reduce1   (true , H2)       H2      :- !.
reduce1   (H1 , true)       H1      :- !.
reduce1   (true ; H2)       true    :- !.
reduce1   (H1 ; true)       true    :- !.
reduce1   (true => H2)      H2      :- !.
reduce1   (H1 => true)      true    :- !.
reduce1   (pi X\ true)      true    :- !.
reduce1   (sigma X\ true)   true    :- !.
reduce1   (box true)        true    :- !.
reduce1   H                 H       :- !.
```

References

[1] Dave Angluin and Carl H. Smith. Inductive inference: theory and methods. *Computing Surveys*, 15(3):237–269, 1983.

[2] Neeraj Bhatnagar. A correctness proof of explanation-based generalization as resolution theorem proving. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*, pages 220–225, 1988.

[3] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[4] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[5] Gerald DeJong and Raymond Mooney. Explanation-based generalization: an alternate view. *Machine Learning*, 1(2):145–176, 1986.

[6] Luis Fariñas del Cerro. Molog: a system that extends PROLOG with modal logic. *New Generation Computing*, 4:35–50, 1986.

[7] Luis Fariñas del Cerro and Martti Penttonen. A note on the complexity of the satisfiability of modal horn clauses. *Journal of Logic Programming*, 4(1):1–10, 1987.

[8] T. M. Dietterich et al. Learning and inductive inference. In Paul R. Cohen and Edward A. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, pages 325–511, William Kaufmann, 1982.

[9] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. In Alberto Maria Segre, editor, *Sixth International Workshop on Machine Learning*, pages 447–449, Morgan-Kaufmann Publishers, San Mateo, California, June 1989.

[10] Scott Dietzen and William L. Scherlis. Analogy in program development. In J. C. Boudreaux, B. W. Hamill, and R. Jernigan, editors, *The Role of Language in Problem Solving 2*, pages 95–117, North-Holland, 1987. Also available as Ergo Report 86 013, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[11] Michael R. Donat and Lincoln A. Wallen. Learning and applying generalised solutions using higher order resolution. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 41–60, Springer-Verlag LNCS 310, Berlin, May 1988.

[12] Conal Elliott and Frank Pfenning. eLP: a Common Lisp implementation of λProlog in the Ergo Support System. Available via anonymous ftp from a.ergo.cs.cmu.edu, October 1989. Send mail to elp-request@cs.cmu.edu on the Internet for further information.

[13] Thomas Ellman. Explanation-based learning: a survey of programs and perspectives. *ACM Computing Surveys*, 21(2):163–221, June 1989.

[14] Martin S. Feather. A survey and classification of some program transformation approaches and techniques. In *IFIP TC2 Working Conference on Program Specification and Transformation*, North-Holland, 1986.

[15] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989.

[16] Amy Felty and Dale A. Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 61–80, Springer-Verlag LNCS 310, Berlin, May 1988.

[17] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF.* Springer-Verlag LNCS 78, 1979.

[18] Masami Hagiya. Generalization from partial parameterization in higher-order type theory. *Theoretical Computer Science*, 63:113–139, 1989.

[19] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 942–959, MIT Press, Cambridge, Massachusetts, August 1988.

[20] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, IEEE, June 1987.

[21] Haym Hirsh. Combining empirical and analytical learning with version spaces. In *Sixth International Workshop on Machine Learning*, Morgan-Kaufmann, June 1989. To appear.

[22] Haym Hirsh. Explanation-based generalization in a logic-programming environment. In *Proceedings of IJCAI*, pages 221–227, 1987.

[23] Haym Hirsh. Reasoning about operationality for explanation-based learning. In *Proceedings of the Fifth International Machine Learning Conference*, June 1988.

[24] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[25] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[26] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic*. Methuen and Co., Ltd., London, 1968.

[27] Smadar T. Kedar-Cabelli and L. Thorne McCarty. Explanation-based generalization as resolution theorem proving. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 383–389, 1987.

[28] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Soar: an architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[29] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational llinguistics. In *Proceedings of the 24th Annal Meeting of the Association for Computational Linguistics*, pages 247–255, 1986.

[30] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Journal of Pure and Applied Logic*, 1988. To appear. Available as Ergo Report 88-055, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[31] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Second Annual Symposium on Logic in Computer Science*, pages 98–105, IEEE, June 1987.

[32] Dale A. Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *Symposium on Logic Programming, San Francisco*, IEEE, September 1987.

[33] Steven Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of AAAI*, pages 564–569, 1988.

[34] Steven Minton, Jaime Carbonell, Craig Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. *Explanation-Based Learning: A Problem-Solving Perspective*. Technical Report CMU-CS-89-103, Carnegie Mellon University, Pittsburgh, PA, January 1989.

[35] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: a unifying view. *Machine Learning*, 1(1):47–80, 1986.

[36] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, MIT Press, Cambridge, Massachusetts, August 1988.

[37] Lawrence Paulson. *Tactics and Tacticals in Cambridge LCF.* Technical Report 39, University of Cambridge, Computer Laboratory, July 1983.

[38] Frank Pfenning. Program development through proof transformation. In Wilfried Sieg, editor, *Logic and Computation*, AMS, Providence, Rhode Island, 1988. To appear. Available as Ergo Report 88–047, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[39] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208, ACM Press, June 1988. Available as Ergo Report 88–036, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[40] Armand E. Prieditis and Jack Mostow. Prolearn: toward a Prolog interpreter that learns. In *Proceedings of AAAI*, Spring 1987.

[41] Paul S. Rosenbloom and John E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of AAAI*, pages 561–567, 1986.

[42] William L. Scherlis. Program improvement by internal specialization. In *Eighth Symposium on Principles of Programming Languages*, pages 41–49, ACM, ACM, January 1981.

[43] Douglas R. Smith and Thomas T. Pressburger. *Knowledge-Based Software Development Tools.* Technical Report KES.U.87.6, Kestrel Institute, June 1987.

[44] P.B. Thistlewaite, M.A. McRobbie, and R.K. Meyer. *Automated Theorem-Proving in Non-Classical Logics.* Pitman, London, 1988.

[45] Lincoln A. Wallen. *Automated Proof Search in Non-Classical Logics: Efficient Matrix Proof Methods for Modal and Intuitionistic Logics.* PhD thesis, University of Edinburgh, 1987.