Machine Learning, 9, 23–55 (1992)

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

SCOTT DIETZEN dietzen@cs.cmu.edu
FRANK PFENNING fp@cs.cmu.edu
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890

Editor: Jack Mostow

Abstract. Certain tasks, such as formal program development and theorem proving, fundamentally rely upon the manipulation of higher-order objects such as functions and predicates. Computing tools intended to assist in performing these 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 thus far been limited to first-order problem representations. We extend EBG to generalize higher-order values, thereby enabling its application to higher-order problem encodings.

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 our basic framework for realizing higher-order EBG. In order to capture the distinction between domain theory and training instance upon which EBG relies, we extend λProlog with the necessity operator □ of modal logic. We develop a meta-interpreter realizing EBG for the extended langauge, $\lambda^{\Box}$Prolog, and provide examples of higher-order EBG.

Keywords. Explanation-based generalization (EBG), higher-order logic, modal logic, logic programming, metal-level reasoning, Prolog, theorem proving, program transformation

1. Introduction

Certain tasks, such as program development and theorem proving, 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 those 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 memorizing (*i.e.*, caching) particular solutions is 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* learning. Inductive methods produce a description of a desired

An extended abstract of this work appears in the *Sixth International Workshop on Machine Learning* (see (Dietzen & Pfenning, 1989)).

concept by examining a set of instances of that concept (Angluin & Smith, 1983; Dietterich, et al., 1982). Typically, syntactic operations are employed to derive a generalization covering those instances (*e.g.*, a pattern that matches each of them).

Conversely, analytical methods generalize from a single example by employing a *theory*, or knowledge of the problem domain, to determine how to generalize. To date, work on analytical techniques relies primarily upon *explanation-based generalization* (EBG) as its central mechanism (Mitchell, Keller & Kedar-Cabelli, 1986; DeJong & Mooney, 1986; Minton, et al., 1989; Ellman, 1989). EBG abstracts a particular problem solution (*i.e.*, a proof or *explanation*), yielding an encapsulation of that solution—that is, a *derived rule* that more efficiently solves the original as well as related problems. While the proof-based generalizations of EBG are necessarily valid (with respect to the domain theory), similarity-based generalizations are guaranteed only to the extent that they cover the given examples.

Generalization and learning performance are intimately tied to the underlying language for representing the domain, or *representation language.* If knowledge is encoded in an inappropriate language, 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. In order to realize EBG over problem domains formulated within higher-order language, 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 (Kedar-Cabelli & McCarty, 1987; Prieditis & Mostow, 1987; Hirsh, 1989; Bhatnagar, 1988). 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, query, derived rule, operationality criteria, *etc.* (These concepts are defined in §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 language of Horn logic (*e.g.*, Prolog), each of which has a significant impact on EBG:

- Integrated support for higher-order objects including variables ranging over such objects. This support is realized through the higher-order logic programming language λProlog.
- The incorporation of the modal operator $\Box$ to formalize the distinction between domain theory (*i.e.*, 'general rules') and training instance (*i.e.*, 'particular facts') upon which EBG relies. Extending λProlog with $\Box$ results in a rich language for higher-order EBG—$\lambda^{\Box}$Prolog.

Overview. We begin, in §2, by introducing first-order EBG within the logic programming framework. As this section also introduces notation of λProlog and concepts unique to our formulation of EBG, it should be worthwhile even for readers familiar with the topic. Next we extend λProlog with $\Box$ in §3. Within §4 we discuss higher-order representation language, and then consider higher-order EBG within §5. Sections 6 & 7 illustrate higher-order EBG's application to theorem proving and program development tasks, respectively. Finally, we develop, in §8, an implementation of $\lambda^{\Box}$Prolog and EBG through a meta-interpreter written in λProlog. Each of the examples contained herein was produced with this prototype.

The brevity of this paper requires that we presume some minimal familiarity with Prolog and the simply-typed λ-calculus; respective introductions may be found within Sterling and Shapiro (1986) and Hindley and Seldin (1986).

2. First-order EBG

We begin by briefly illustrating explanation-based generalization with a first-order example from DeJong and Mooney (1986, pp. 158–166): (We apologize to any readers offended by the morbidity of this example, but is has become standard in the literature.) EBG divides the theory of the problem domain between a *domain theory*, which we also denote with $\mathfrak{D}$:

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 theory or* $\mathfrak{I}$:

depressed john.
buy john obj1.
gun obj1.

(Within our examples, constants are in boldface while variables are in italics. As in Prolog ' , ' denotes conjunction. The symbol ⇐ represents implication, and is equivalent to Prolog's ' :- '.) Both $\mathfrak{D}$ and $\mathfrak{I}$ are composed of λProlog *clauses*. For readers familiar with EBG, $\mathfrak{I}$ roughly corresponds to training instance; justification for the new terminology is given within §3.

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

?- **kill john john.**

EBG then requires a proof that solves the given query. Within our paradigm, such an explanation may be expressed as a trace of λProlog computation. A proof of the above query is illustrated within Figure 1. Goals of the proof are underlined, while the program clause that reduces a particular goal appears underneath. In the course of applying each clause, its variables may be unified with constants or variables of the goals, resulting in the given unification constraints (enclosed in '⟨⟩').

EBG generalizes this explanation to produce an encapsulation of the employed proof strategy. In Figure 2, a generalized proof is constructed that corresponds to the original, except that clauses of $\mathfrak{I}$ (or $\mathfrak{I}$-clauses) are omitted. This forms EBG's bias in the generalization space: the proof of the given query is generalized by abstracting steps involving clauses of the training theory. At the root of the new proof is a generalized query, which is derived from the original by replacing each of the first-order constants with a variable: the goal **kill john john** becomes the general goal **kill** *X Y*. Clauses of $\mathfrak{D}$ (or $\mathfrak{D}$-clauses) applied in the first proof are correspondingly applied in the second. This restricts the outcome by propagating unification constraints through the proof (*e.g.*, **kill** *X Y* becoming **kill** *X X*).

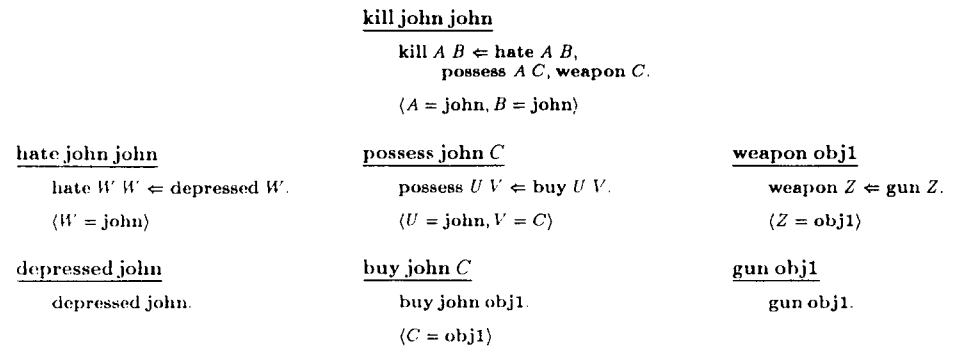

Figure 1. First-order proof.

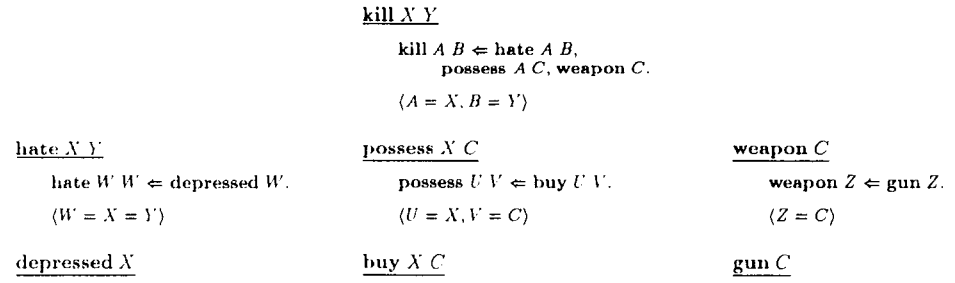

Figure 2. First-order generalized proof.

Leaves of the generalized proof (*e.g.*, **gun** *C*) correspond to subgoals of the original proof that were derived from $\mathfrak{I}$. 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 proof encapsulation as a *derived rule*, or as an *explanation-based generalization*, or simply as a *generalization*.

3. Modal logic

Our formulation of EBG relies upon the separation of $\mathfrak{D}$ and $\mathfrak{I}$, since only rules of the former are incorporated within generalized proofs. To differentiate the two, we prefix $\mathfrak{D}$-clauses with the □ operator, which is borrowed from modal logic—logics in which propositions have multiple levels or modes of truth, such as 'may be' and 'must be.'[1, 2]

We illustrate our use of □ on the first-order example of §2. $\mathfrak{D}$ and $\mathfrak{I}$, which constitute the logic program, may now be jointly expressed as

□ ∀*A*∀*B*∀*C*. **kill** *A B* ⇐ **hate** *A B*, **possess** *A C*, **weapon** *C*.
□ ∀*W*. **hate** *W W* ⇐ **depressed** *W*.

$\Box\ \forall U\ \forall V.$ **possess** $U\ V$ $\Leftarrow$ **buy** $U\ V.$
$\Box\ \forall Z.$ **weapon** Z $\Leftarrow$ **gun** $Z.$
depressed john.
buy john obj1.
gun obj1.

The above presentation does *not* rely upon λProlog's implicit universal quantification of a program's logical variables. This is because our EBG algorithm differentiates between the clauses $\Box\ \forall x.\ D$ and $\forall x.\ \Box\ D$. (The motivation for this distinction is beyond the scope of the paper; see Dietzen (1991).) However, since explicitly specifying quantifiers can become exceedingly tedious, we introduce the '!!' shorthand to represent this universal quantification implicitly. The first clause of $\mathfrak{D}$ may then be expressed as

!! **kill** $A\ B$ $\Leftarrow$ **hate** $A\ B$, **possess** $A\ C$, **weapon** $C.$

From the query **kill john john**, the resulting explanation-based generalization becomes

!! **kill** $X\ X$ $\Leftarrow$ **depressed** X, **buy** $X\ C$, **gun** $C.$

Traditionally, the modal operator $\Box$ (sometimes called '**L**') 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.' Our incorporation of $\Box$ is founded upon a correspondence between (1) EBG's separation of domain and training theory and (2) modal logic's separation of necessary and contingent truth: Because the validity of the generalizations derived through EBG depend solely upon $\mathfrak{D}$, more stringent truth requirements are placed upon $\mathfrak{D}$-clauses—namely that they be true in all possible configurations of the problem space being modeled. Clauses of $\mathfrak{I}$, as they are excluded from generalized proofs, can safely be revised or removed without invalidating the derived generalizations (*e.g.*, **depressed john** becoming false). Such revision could be explained semantically as 'changing states' or 'switching worlds.'

Suppose that within the suicide example, we remove the $\Box$ from the clause **weapon** Z $\Leftarrow$ **gun** Z. This results in the generalized proof of Figure 3 and the generalization

!! **kill** $X\ X$ $\Leftarrow$ **depressed** X, **buy** $X\ C$, **weapon** $C.$

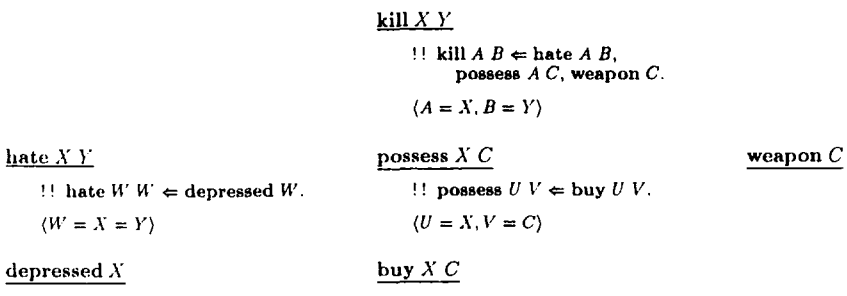

Figure 3. Less specific generalized proof.

The above rule is more general than the previous one, but its application requires more work. This illustrates the trade-off inherent in the partitioning of $\mathfrak{D}$ and $\mathfrak{I}$: $\mathfrak{D}$-clauses get 'compiled into' the rules derived through EBG, while $\mathfrak{I}$-clauses must be evaluated at the time of application.

Now, again within the original example, suppose instead that we replace the last clause with □ **gun obj1.** This has the effect of 'anchoring' the generalization to **obj1**, with the result of an identical query being the generalized proof of Figure 4 (whose rightmost branch is solved), and the generalization

!! **kill** X X ⇐ **depressed** X, **buy** X **obj1.**

By moving a clause from $\mathfrak{I}$ or $\mathfrak{D}$, we make the resulting generalization more specific. Such a shift is, however, dangerous in that the generalization then depends upon the validity of □ **gun obj1.** In another configuration where **obj1** is not a gun, the derived rule is false!

Training instance. Previous realizations of EBG have used the term 'training instance' rather than our 'training theory.' While the literature makes the same operational distinction of excluding clauses of the training instance from generalized proofs, the term 'training instance' additionally carries the connotation of embodying a single example situation from which the learner should generalize. We have taken the liberty of renaming $\mathfrak{I}$ to avoid that connotation.

Typically within logic programming implementations of EBG, atomic clauses are directly recognized as belonging to the training instance (Kedar-Cabelli & McCarty, 1987; Hirsh, 1987; Prieditis & Mostow, 1987)—*e.g.*, **gun obj1**. Although this notion of training instance offers some intuitive value, we find it artificially restrictive. There exist atomic clauses that we might want to include within $\mathfrak{D}$, such as !! **adjacent** X X.[3] The same is true even for constant atomic clauses: for example, to represent that **block1** is glued to the table we could assert □ **on block1 table**. Alternatively, we might want to include variables and logical connectives within $\mathfrak{I}$-clauses: for example, under the temporary condition that all blocks are stacked in two-high pairs, we might assert the rules **clear** X ⇐ **on** X Y and **on** Y **table** ⇐ **on** X Y. □ furthermore affords the potential to intermix knowledge of the domain and training theory through the nesting of □ below the top-level.

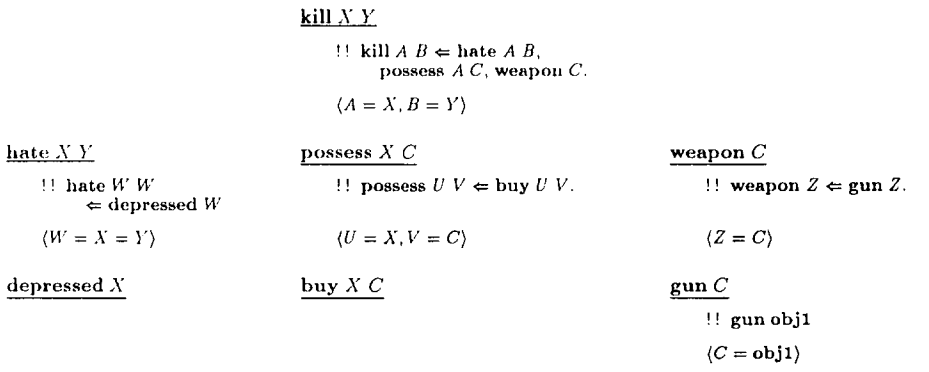

Figure 4. More specific generalized proof.

Our use of $\Box$, then, avoids what we believe to be undue limitations on the training instance; our training theory may instead contain arbitrary λProlog clauses. Besides providing greater expressiveness, a modal logic representation for the distinction between $\mathfrak{D}$ and $\mathfrak{I}$ can be given a clear semantics that is independent of a particular search procedure or generalization algorithm.

Modal logic and EBG. Admittedly, the analogy that contingency is to necessity as training theory is to domain theory is philosophically questionable. The basis for our incorporation of $\Box$ is rather that the operator elegantly *models* the difference between $\mathfrak{I}$ and $\mathfrak{D}$ in a formal (as opposed to an operational) manner—that is, through a formal language and the accompanying proof system. Our use of the terms 'contingency' and 'necessity' is meant to convey some semantic intuition about why $\Box$ models this distinction. One could easily turn this observation around and say that we have found yet another interpretation of $\Box$.[4]

Operationality. We illustrated in §3 how $\Box$ defined which proof steps are included in generalized proofs. Within the EBG paradigm, the traditional means of restricting the extent of generalized proofs is through *operationality criteria*: by establishing that a particular goal meets an operationality criterion the subtree deriving it is 'pruned' from the generalized proof. That is, an operationality criterion can be viewed as a predicate that determines whether a given goal should be a leaf of the generalized proof. The term 'operational' arises from the constraint that such subgoals be efficiently derivable at the time of rule application. To illustrate, if we augment the original formulation of the suicide example (§3) with a declaration that the goal **weapon** *Z* is operational, the EBG algorithm produces the derived rule

$$!!\ \textbf{kill}\ X\ X \Leftarrow \textbf{depressed}\ X,\ \textbf{buy}X\ C,\ \textbf{weapon}\ C.$$

as the derivation of **weapon** *Z* is excluded from the generalized proof.[5, 6]

Although $\Box$ and operationality criteria are both mechanisms that limit the extent of generalized proofs, the former is a property of clauses (*i.e.*, whether or not they contain $\Box$), while the latter is a property of goals (*i.e.*, whether or not they are operational). Operationality criteria present the same trade-off we have seen for $\Box$: the closer the operational subgoals are to the root of the generalized proof, the more generally applicable the derived rule is, but also the more work is required to apply it.

Before continuing our development of EBG, operationality, and $\Box$, we must further discuss higher-order language in general, and λProlog in particular.

4. λProlog—A higher-order logic programming language

λProlog (Nadathur & Miller, 1988) extends traditional logic programming languages primarily

- by providing the simply-typed λ-calculus as a data-type; that is, λProlog terms are simply-typed λ-terms.

- by incorporating the higher-order unification required for λ-terms.
- by including more expressive logic contructs—*e.g.*, embedded implication and quantification.
- by admitting higher-order predicates in a principled manner.

Within this section we briefly introduce λProlog. While this work relies upon and extends λProlog, the language is itself a research prototype.

Higher-order language. We follow common practice in overloading the term 'higher-order' by applying it to values and domains (semantic entities), as well as to languages (syntactic entities). A *domain* is said to be *higher-order* if it contains higher-order values—that is, values which take other values as arguments (*e.g.*, functions and predicates). For instance, the values manipulated by a higher-order programming language include functions. (By 'manipulated' we mean that functions are 'first-class' objects—*i.e.*, they can be bound to variables, passed as parameters, and returned from function calls.) Similarly, within a higher-order logic, the values that can be quantified include functions and predicates.

On the other hand, we consider a representation *language* to be *higher-order* if it contains a means for expressing argument binding: for example, the λ of λ-calculus or **lambda** in LISP. Such languages are particularly amenable to representing the values of higher-order domains, since the formation of higher-order objects can be expressed with λ.

λ-terms. Terms of the simply-typed λ-calculus take the form

$$M ::= c \mid x \mid M\ N \mid \lambda x : \tau.\ M$$

where M and N range over terms, c ranges over constants, x over variables, and τ over simple types. A given λ-*abstraction* $\lambda x{:}\tau.M$ is of function type $\tau \to \tau'$ provided M has type τ'. The juxtaposition $M\ N$ denotes a λ-term *application*, which is of type τ' provided M is of type $\tau \to \tau'$ and N is of type τ. λ-term application associates to the left: **a b c** is read as (**a b**) **c**. Thus the Prolog term **p(a, b)** is written as **p a b** in λProlog.

λ-terms become exceedingly redundant if all of the types required by the syntactic definition are explicitly included. A more succinct representation is afforded by eliding unnecessary type information. *Type reconstruction* is the process of rederiving those omitted types. In practice, all types are omitted from λProlog terms. The types of constants are instead specified by explicit declaration, and the types of variables, untyped constants, abstractions, and applications are then inferred from context. In the sequel, we will omit types with the understanding that they are to be subsequently derived through type reconstruction.

Basic operations on λ-terms. We use the notation $[N/x]M$ to denote the substitution of N for free occurrences of x in M. (Bound variables may have to be renamed to avoid capture; see the example below.) The term operations supported by λProlog include β- and η-reduction as well as α-conversion, which are defined as follows:

$(\lambda x.M)N \Rightarrow_\beta [N/x]M$			*e.g.*, $(\lambda x.\lambda y.\mathbf{f}xy)y \Rightarrow_\beta \lambda y'.\mathbf{f}yy'$
$\lambda x.Mx \Rightarrow_\eta M$	*provided* x not free in M		*e.g.*, $\lambda x.\mathbf{f}x \Rightarrow_\eta \mathbf{f}$
$\lambda x.M =_\alpha \lambda y.[y/x]M$	provided y not free in M		*e.g.*, $\lambda x.x =_\alpha \lambda y.y$

Closures over these operations yield corresponding notions of λ-term convertibility: M is said to be *β-convertible* to M' if there exists a sequence of β-reductions and β-expansions (the inverse), applied at the top-level or to subterms, transforming M to M'. In this calculus, $\beta\eta$-reductions are normalizing and Church-Rosser (Hindley & Seldin, 1986); that is, maximal sequences of such reductions terminate with a unique λ-term said to be in $\beta\eta$-normal form. This property is a consequence of the typing given to λ-terms, and is crucial for the unification algorithm, since the convertibility of two terms can be tested by comparing their normal forms for equivalence modulo the renaming of bound variables (α-conversion).

Higher-order unification. *Unification* is the process of producing a common instance from two or more terms by instantiating either term's free variables with other terms.[7] We use the λProlog notation $M = N$ to indicate that the λ-terms M and N are to be unified. When unifying terms, we are typically interested in the *most general unifier* (MGU); for example, the MGU of $\mathbf{p}x$ and $\mathbf{p}y$ is simply $\langle x = y \rangle$, rather than the overly specific $\langle x = \mathbf{a}, y = \mathbf{a} \rangle$.

Unification underlies the logic programming paradigm, but because λProlog terms are λ-terms, λProlog unification must be higher-order—*i.e.*, it must support the instantiation of variables to functions as well as to first-order constants. λ-terms, however, do not admit unique most general unifiers: consider that the unification of $F\,\mathbf{a} = \mathbf{caa}$ allows the variable F to be instantiated with any of $\lambda x.\mathbf{caa}$, $\lambda x.\mathbf{cxa}$, $\lambda x.\mathbf{ca}x$, or $\lambda x.\mathbf{c}xx$, none of which is an instance of another (they are all closed). Thus, higher-order unification is inherently nondeterministic. Even worse, Goldfarb shows that higher-order (and in particular, second-order) unification is undecidable (Goldfarb, 1981). However, a semi-decision procedure effective in practice is presented by Huet, (1975) and refined by Elliott (1990).

Using higher-order language. Within a higher-order language, binding operators are implemented via the primitive λ. For example, the function $f(x) \equiv 2 * x$ might be represented simply as $f \equiv \lambda x.2 * x$. Similarly, $\forall x \exists y.x < y$ might be expressed using the logical product Π and sum Σ as $\Pi\,\lambda x.\ \Sigma\,\lambda y.\ x < y$. (In fact, this is the representation used within λProlog; the former is simply a more readable abbreviation.) The implementation of other binding operators in terms of λ allows $\alpha\beta\eta$-conversion and λ-term unification to be implemented once within the representation language rather than within individual client programs (Pfenning & Elliott, 1988; Harper, Honsell & Plotkin, 1987). Relegating such tasks to the representation language makes for more succint, elegant programs.

Many domains naturally involve binding constructs, and are thus best represented within higher-order languages: logics, programming languages, and natural languages (Pfenning & Elliott, 1988; Miller & Nadathur, 1987; Miller & Nadathur, 1986; Pereira, 1991). This same need for higher-order representation also arises when one wants to reason 'at the meta-level'—that is, about aspects of λProlog. One would like facts (propositions) or properties (predicates) to be objects themselves. Prolog and other first-order representation languages allow this to some extent, but in a way that is only operationally, but not logically motivated. λProlog, on the other hand, facilitates higher-order programming—that is, the ability to create goals and programs, and pass them as arguments.

First- *vs.* higher-order. When higher-order values are represented within first-order languages, we often need 'new variables,' need to check conditions such as 'where x does

not occur in M', or must implement substitution in a way that 'renames bound variables if necessary.' Additionally, procedures that depend upon the binding operator—*e.g.*, $\alpha\beta\eta$-conversion and higher-order unification—must be explicitly programmed. All this makes for a prohibitively complex encoding.

The extent of the overhead incurred through these higher-order operations remains unclear. On the one hand, direct implementation of $\alpha\beta\eta$-conversion and higher-order unification is generally more efficient than user-programmed encodings, since an extra layer of language is avoided. On the other hand, the full power of higher-order unification is potentially too costly. Yet Huet's semi-decision procedure is effective in practice, simply because typical applications of λ-term unification are more restricted than the worst case.

A subset of λProlog named L_λ is currently being developed by Miller (1990). L_λ restricts higher-order unification to maintain the attractive properties of first-order unification, namely decidability and most general unifiers. The overhead of L_λ's restricted higher-order unification is not significantly different than that of first-order. A discussion of the relevance of L_λ to this work is beyond the scope of the paper.

λProlog clauses and goals. Simply-typed λ-terms and higher-order unification underlie λProlog; now we turn to the logical connectives of the language. λProlog terms are distinguished based upon whether they appear as a goal G or a program clause D. For Prolog the two classes may be inductively defined as

$$G ::= \mathbf{true} \mid A \mid G_1 \,,\, G_2 \mid G_1 \,;\, G_2$$
$$D ::= \mathbf{true} \mid A \mid A \leftarrow G$$

where G ranges over goals (also termed G-formulas or G-forms), D over program clauses (or D-forms), and A over atoms. (Atoms are terms of type $\mathbf{o}$—the reserved type of λProlog propositions—that do not have a logical operator at the top level. Variables within D are implicitly universal, while those within G are existential.)

For λProlog, we have instead

$$G ::= \mathbf{true} \mid A \mid G_1 \,,\, G_2 \mid G_1 \,;\, G_2 \mid D \Rightarrow G \mid \forall x\,[:\tau].\, G \mid \exists x\,[:\tau].\, G$$
$$D ::= \mathbf{true} \mid A \mid D_1 \,,\, D_2 \mid D \leftarrow G \mid \forall x\,[:\tau].\, D$$

where both $\Rightarrow$ and $\leftarrow$ represent (intuitionistic) implication. Thus $G \Rightarrow D$ and $D \leftarrow G$ stand for the same formula. Typically, a goal will be written as $D \Rightarrow G$, while a clause will be written as $D \leftarrow G$. Operationally, the G-form $D \leftarrow G$ is interpreted by assuming D for the solution of G.

The above classes define the core of λProlog—the higher-order hereditary Harrop formulas (Nadathur & Miller, 1988), which generalize Horn clauses while preserving the basic character of a logic programming language. Clauses are restricted in that they may not contain disjunction (;) or existential quantification ($\exists$), because of the difficulty in giving an operational interpretation to such D-forms.

An example λProlog program. We herein consider the programming of a higher-order predicate **select**, such that **select** $P\ K\ L$ insures that L is a sublist of K for which P holds. The type of **select** is

type **select** $(A \to \mathbf{o}) \to \mathbf{list}\ A \to \mathbf{list}\ A \to \mathbf{o}$.

select may be programmed as

select P $(x :: K)$ $(x :: L)$ $\Leftarrow$ $P\ x$, **select** $P\ K\ L$.
select P $(x :: K)$ L $\Leftarrow$ **select** $P\ K\ L$.
select P **nil** **nil**.

(:: adds an element to the head of a list.) The following query, for example, selects grandparents from the given input list:

?- **select** $(\lambda x.\exists y.$ (**parent** $y\ x$, $\exists z.$**parent** $z\ y$)) (**tom** :: **kate** :: **leo** :: **nil**) L.

Readers may argue that **select** could be formulated within Prolog simply by replacing $P\ x$ with **apply** $P\ x$, and further, that **grandparent** could itself be encoded as a top-level Prolog predicate:

grandparent x $\Leftarrow$ **parent** $y\ x$, **parent** $z\ y$.

In many situations, however, the 'inline' expression of higher-order arguments (such as the unnamed **grandparent**) is either necessary or desirable: for instance, reformulation at the top-level is not applicable to higher-order functions that are not predicates (*i.e.*, not of type **o**). Moreover, first-order languages do not permit many operations over predicates, such as composition: consider

select_or P Q K L $\Leftarrow$ **select** $(\lambda x.Px;\ Qx)\ K\ L$.

(where the operator ' ; ' represents inclusive 'or'.) Within Prolog, **select_or** cannot be programmed in terms of **select**, at least not without revising **select**'s original definition.

5. Higher-order EBG

In §4 we made the case for the additional expressiveness afforded by higher-order language, and in particular for λProlog. Expressive elegance is intimately tied to effective generalization: succinct and elegant rules make for succinct and elegant generalizations.

We would like to assert more, namely that first-order encodings are inadequate for the task of generalization within higher-order domains, because primitive syntactic manipulations inevitably intrude into the generalizations. To justify this claimed inadequacy, one might attempt to formalize a given higher-order example within a first-order language. However, at best such a strategy could only establish the inadequacy of one particular formulation. Arguing that first-order encodings are generally insufficient for higher-order domains and higher-order generalization is more problematic, because first-order languages certainly are expressively (and computationally) adequate.

Instead, we suggest the proper question is whether programmers should be unduly constrained in their choice of language. Higher-order representation languages are *higher-level* in that their additional expressiveness eases the task of programming over higher-order domains: witness the success of higher-order programming languages such as ML, LISP, Scheme, and more recently (and to a lesser degree) λProlog. As we can expect programmers to continue to make use of higher-order languages, the successful application of EBG to these domains necessitates that the paradigm be extended to higher-order generalization.

We illustrate higher-order EBG within the domain of symbolic integration. Consider the following higher-order rules: the first treats exponentiation, the second extracts a constant factor, and the third splits a sum. (The predicate **intgr** relates a function to its indefinite integral. We use a mathematical notation for arithmetic operators not included in λProlog—in particular, exponentiation and division—as it increases readability.)

```
!! intgr (λx.x^a)          (λx.x^(a+1)/(a + 1)).
!! intgr (λx.a * Fx)       (λx.a * Hx)          ⇐ intgr F H.
!! intgr (λx.Fx + Hx)      (λx.F'x + H'x)       ⇐ intgr F F', intgr H H'.
   intgr cos               sin.
```

The traditional binding notation of dx has been replaced with λ-terms. Missing from the first rule is the restriction that $a \neq -1$, because λProlog does not admit constraints other than those imposed by unification.[8] The integration rule for cosine is an example of a Ɜ-clause that is not 'contingent': while the rule is valid in the same sense as the others, it represents a proof step we wish to abstract under EBG.

The query

?- **intgr** $(\lambda x.3 * x^2 + \cos x)$ H.

yields the solution

$$H = \lambda x.3 * x^{2+1}/(2 + 1) + \sin x$$

and the generalization

!! **intgr** $(\lambda x.a * x^b + Fx)$ $(\lambda x.a * x^{b+1}/(b + 1) + F'x)$ ⇐ **intgr** F F'.

The proof and generalized proof associated with this example are given in Figures 5 and 6, respectively.[9]

The generalization space of higher-order EBG is significantly larger than that of first-order in that 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 G ranging over goals is fully general. Also unlike the proofs of §2 & 3, the integration proofs make use of higher-order unification, which implicitly enforces the restrictions placed upon free and bound variables: for example, within an application of the power rule, $\lambda x.x^a$ will not unify with $\lambda x.x^x$ since a may not contain free

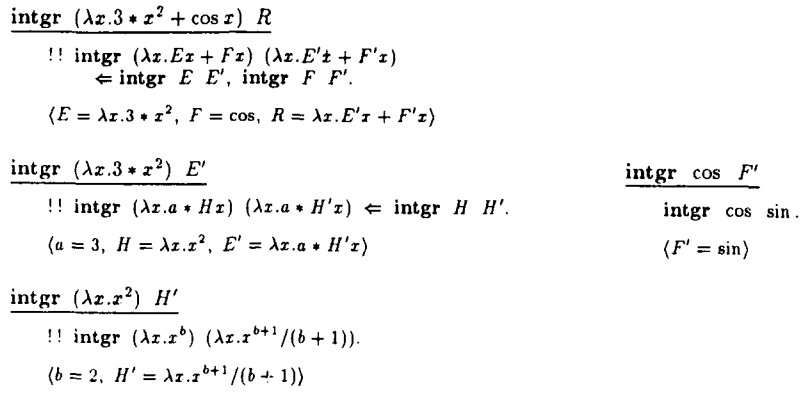

Figure 5. Higher-order proof.

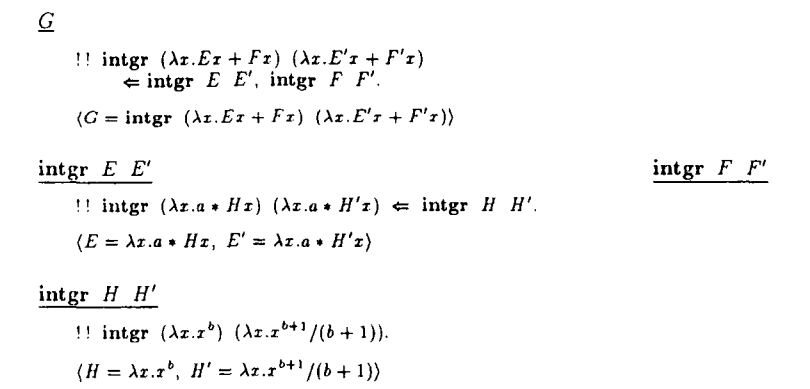

Figure 6. Higher-order generalized proof.

occurrences of x.[10] Moreover, function variables may then appear in the derived generalizations (*e.g.*, F). While this application is limited to abstracting λ-terms, $\lambda^{\Box}$Prolog supports generalization over predicates as well.

6. Search control via tactics

The previous integration example relied upon logic programming's implicit search to solve queries. Additional levels of search control need not, however, interfere with the underlying process of EBG! We demonstrate this by implementing a *tactic-based* approach to the symbolic integration problem. Search is controlled within a tactic-based theorem prover (or problem solver) by requiring the user to *a priori* or interactively specify a combination

of proof steps, or *tactics*, with which to attempt the derivation of a goal (Gordon, Milner & Wadsworth, 1979; Constable, et al. 1986). These tactics guide the construction of an actual proof (or problem solution).[11]

Tactics are simply named rules: for the integration domain, we have

```
!! tac power          (intgr (λx.x^a)     (λx.x^(a+1)/(a + 1)))  true.
!! tac constant_left  (intgr (λx.a * Fx)  (λx.a * F'x))          (intgr F F').
   tac cos_tac        (intgr cos          sin)                   true.
```

Tactics perform goal reduction: the input goal G_{in} (2nd argument) is reduced to a more easily solved subgoal G_{out} (3rd argument).

To represent compositions of tactics, we have problem independent *meta*-tactics, or *tacticals*, such as

```
!! tac idtac          G_in G_in.
!! tac (then T1 T2)   G_in G_out ⇐ tac T1 G_in G_med, tac T2 G_med G_out.
!! tac (orelse T1 T2) G_in G_out ⇐ tac T1 G_in G_out; tac T2 G_in G_out.
!! tac (repeat T)     G_in G_out ⇐ tac (orelse (then T (repeat T)) idtac) G_in G_out.
```

We augment the above with a special interactive tactical:

```
!! tac interactive G_in G_out ⇐ write_string ''Goal to be reduced: '', write G_in,
                                newline, write_string ''Enter tactic/tactical :'',
                                read λT. tac T G_in G_med, ((G_med = true, G_out = true)
                                                          ; tac interactive G_med G_out).
```

(λProlog's input predicate **read**, which is of type $(A \to \mathbf{o}) \to \mathbf{o}$, differs from Prolog's in that the entered term is bound to the variable T before execution of **read**'s body.)

Now to solve the query

```
?- tac interactive (intgr (λx.2 * (3 * cos x)) H)  G_out.
```

we could enter the series of tactics **constant_left**, **constant_left**, and **cos_tac** as prompted; or equally, the tactical (**then** (**repeat contant_left**) **cos_tac**), yielding

$$H = \lambda x.2 * (3 * \sin x)$$
$$G_{\text{out}} = \mathbf{true}$$

as well as the generalization

```
!! tac interactive (intgr (λx.a * (b * Fx)) (λx.a * (b * F'x)))  (intgr F F').
```

Level of generalization. Since tactics of the training theory are abstracted, the above generalization is applicable to problems not addressed by the original tactical: for example, $\lambda x.2 * (3 * \sin x)$. At the same time, this derived rule does not cover the range of

problems for which the tactical (**then** (**repeat constant_left**) **cos_tac**) is applicable: consider **intgr** ($\lambda x.3 * \cos x$) H. This is because the tactical- or meta-level is formulated completely within $\mathfrak{D}$, and hence generalization does not occur at that level; instead generalization is confined to the tactic- or rule-level. This is, of course, exactly what we were after when we set out to make the additional level of search control transparent to EBG. Alternative formulations could produce generalizations at the tactical-level, but those derived rules are more likely to be so general that they would be difficult to apply.

Level of assimilation. Within this paper, we have concentrated on how a rich representation language supports EBG, and have ignored questions concerning how these generalizations may be assimilated and applied automatically. Under the traditional approach, the underlying architecture of the problem solver 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.[12]

This approach to assimilation is, however, problematic for tactic-based paradigms. In the above example, although generalization occurs only at the level of tactics, the derived rule nevertheless contains a reference to the tactical **interactive**. If we are to maintain a strict separation of the rule-level and meta-level, it does not make sense to assimilate a generalization encompassing both levels. Rather, a slightly modified generalization could be assimilated at the rule-level as a derived tactic:

!! **tac** **constant_left_two** (**intgr** ($\lambda x.a * (b * Fx)$) ($\lambda x.a * (b * F'x)$)) (**intgr** $F F'$).

The point is that it is the user (or client program), rather than the architecture, which is in a position to control assimilation. If we were to instead directly assimilate the original generalization, we compromise the predicate **interactive** in that a subsequent invocation might no longer prompt the user; that is, we compromise the user's control over search.

This example reinforces our belief that for such applications EBG should be a *feature* of the language in which problem solvers are coded, rather than a 'black box' within the problem solving architecture. In other words, what is required is a language in which one can *program* the learning mechanism. By providing the programmer with an explicit means to control generalization and assimilation, we defer the difficult problem of determining when to generalize and assimilate. Client programs have the potential advantage of bringing domain knowledge and user interaction to bear in determining what is to be learned. This concept of programming generalization and learning within the same language in which problem solving and interaction occur is markedly different from what we label 'black-box' learning. Hence our approach stands in contrast to systems such as SOAR (Laird, Rosenbloom & Newell, 1987), Prodigy (Minton, et al., 1989), and LEAP (Mitchell, Mahadevan, & Steinberg, 1985) in which learning is largely relegated to the underlying architecture. (A thorough treatment of our approach to programmable generalization and assimilation is beyond the scope of this paper; see instead Dietzen (1991).

Operationality *vs.* □. While §3 illustrated that both □ and operationality criteria serve to define EBG's generalized proofs (and hence its results), the tactic example above demonstrates that the mechanisms are *not* equivalent: consider that a formulation of the integration domain that replaces □ with operationality criteria (defined via the predicate **oper**) requires specifying

oper (**tac** **interactive** (**intgr** cos sin) **true**).

The problem is that this definition again forces the mixing of the rule- and meta-level, thereby violating the modularity of our encoding.

Operationality criteria, on the other hand, do provide features beyond the capabilities of □. For instance, they offer a generally more concise means to define generalized proofs: by declaring that only a single subgoal is operational, the entire branch of the generalized proof underneath is excluded, or 'pruned', from the generalization. Achieving the same effect with □ alone would require removing each of the program clauses applied within that branch from $\mathfrak{D}$. Furthermore, if a particular rule is used pervasively in a proof, it might be necessary to include it within both $\mathfrak{D}$ and $\mathfrak{I}$ (and then use some form of additional control to discriminate between occurrences.) Operationality criteria do not present a corresponding problem, as it is unlikely that recurring subgoals could be considered both operational and non-operational.

□ does, however, offer the means to generalize in an entirely different manner: consider that even interior steps can be abstracted from generalized proofs via □.

We conclude that the mechanisms of operationality criteria and □ are complementary, and while □ is sufficient to formulate the examples presented within this paper, we do not suggest it as a replacement for operationality criteria. In fact, the combination of the two is particularly attractive: modal logic induces an underlying limit to the specialization of derived rules that potentially prohibits EBG from yielding 'incorrect' generalizations, while operationality criteria provide a means to 'fine tune' selection from the space of possible generalizations admitted by □.

7. Program transformation and apprentice learning

One paradigm for formal program development is that of *program transformation* (Burstall & Darlington, 1977; Huet & Lang, 1978; Feather, 1986; Partsch & Steinbrüggen, 1983). Under a transformational 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.[13,14]

We illustrate EBG over a transformational system which we have applied to induce tail recursion in certain situations (Dietzen & Scherlis, 1987). (From a tail recursive version, an iterative form could easily be derived.) As an understanding of the derivation's details is unnecessary for this discussion, we defer presentation of the full derivation to Appendix A.

We begin with a functional specification of the factorial program:

$$\textbf{fix}\ \lambda \mathit{fact}.\ \textbf{lam}\ \lambda n.\ \textbf{if}\ (\textbf{equals}\ n\ 0)\ 1\ (\textbf{appl}\ \mathit{fact}\ (n-1)) * n$$

The above is a λProlog abstract syntax for a simple functional language. The constructs **lam** and **appl** represent object-level λ-abstraction and application, respectively. The fixpoint or recursion operator **fix** is 'applied' by substituting its body for each occurrence of the bound identifier within its body (see Appendix A).

The derivation proceeds by applying transformations to this specification. For example, the following transformation replaces an occurrence of e with $op\ e\ \mathbf{z}$, where $\mathbf{z}$ is a right identity of op (for example, mapping $\mathbf{a}$ to $\mathbf{a} + 0$):

$$!!\ \textbf{add\_id\_right}\ \ op\ \ C\ \ (C\ e)\ \ (C\ (op\ e\ z))\ \ \Leftarrow\ \ \textbf{right\_identity}\ op\ z.$$

The third and fourth arguments match the input and output object programs, respectively. The second argument C specifies a *context—i.e.*, the particular subexpression of the input program to be transformed. These higher-order context variables serve to formally encode subterm or *occurrence* selection, which might, for example, result from "pointing with a mouse" (Pfenning & Elliott, 1988). (This constitutes yet another application of higher-order representation language: the formal expression of occurrences.) For example, within the following invocation of the transformation

$$?-\ \textbf{add\_id\_right}\ \ (\lambda x.\lambda y.x + y)\ \ (\lambda g.g * h)\ \ (\mathbf{a} * \mathbf{b})\ \ F_{\text{out}}.$$

the context variable C is $\lambda g.g * h$. From the definition of **add_id_right** above, C is applied to e and then matched against the input $\mathbf{a} * \mathbf{b}$; that is,

$$\begin{aligned} C\ e &= (\lambda g.g * h)\ e \\ &=_\beta e * h \\ &= \mathbf{a} * \mathbf{b} \end{aligned}$$

Thus, e is instantiated to $\mathbf{a}$ and h to $\mathbf{b}$. Now, given that

$$\textbf{right\_identity}\ (\lambda x.\lambda y.x + y)\ 0$$

the output F_{out} is instantiated as follows:

$$\begin{aligned} F_{\text{out}} &= C\ (op\ e\ z) \\ &= (\lambda g.g * \mathbf{b})\ ((\lambda x.\lambda y.x + y)\ \mathbf{a}\ 0) \\ &=_\beta (\lambda g.g * \mathbf{b})\ (\mathbf{a} + 0) \\ &=_\beta (\mathbf{a} + 0) * \mathbf{b} \end{aligned}$$

The full derivation **tail_rec**, which consists of a sequence of ten such transformation rules and the associated contexts, constitutes a *meta-program—i.e.*, a program that manipulates an object program such as *fact*. This meta-program would ideally be constructed interactively by alternatively selecting rules and contexts; as eLP currently lacks the interface necessary to interpret 'mouse input', **tail_rec** was instead hand-coded. The meta-program is applied to *fact* through the query

```
?- tail_rec (λx.λy.x * y)
            (fix λfact. lam λn. if (equals n 0)
                                   1
                                   (appl fact (n − 1)) *n)
            F_out.
```

which yields the tail recursive expression

```
F_out = appl (fix λfact_1. lam λm. lam λn. if (equals n 0)
                                              m
                                              (appl (appl fact_1 (n *m)) (n − 1)))
             1
```

But more interesting is the generalization:

```
!! tail_rec op
             (fix λf. lam λy. if (H_1 y)
                                 a
                                 (op (appl f (H_2 y)) (H_3 y)))
             (appl (fix λf'. lam λx. lam λy. if (H_1 y)
                                                x
                                                (appl (appl f' (op (H_3 y) x))
                                                      (H_2 y)))
                   b)
  ⇐ right_identity op b, left_identity op a, associative op.
```

The result produced by our prototype is not so 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:

```
?- tail_rec append
            (fix λrev. lam λl. if (null l)
                               nil
                               (append (appl rev (tl l)) (hd l) :: nil)))
            F_out.
```

which instantiates[15]

$$\begin{aligned} a &= \textbf{nil} \\ b &= \textbf{nil} \\ H_1 &= \textbf{null} \\ H_2 &= \textbf{tl} \\ H_3 &= \lambda l.\,\textbf{hd}\, l :: \textbf{nil} \end{aligned}$$

yielding the tail-recursive version

$$\begin{aligned} F_{\text{out}} = \textbf{appl}\,(\textbf{fix}\,\lambda rev_1.\ \textbf{lam}\,\lambda k.\ \textbf{lam}\,\lambda l.\ \textbf{if}\,(\textbf{null}\, l) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad k \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{appl}\,(\textbf{appl}\, rev_1\,(\textbf{append}\,((\textbf{hd}\, l) :: \textbf{nil})\, k)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{tl}\, l))) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{nil} \end{aligned}$$

The above result requires only the addition of a final simplification to make the reduction from (**append** ((**hd** l) :: **nil**) k) to ((**hd** l) :: k). Hence, the generalized *fact* derivation is sufficient for *rev* as well (except for final simplification).

The elegance of the above generalization is largely due to the expressiveness of higher-order language. In particular, essential restrictions on the input program are implicit in the higher-order notation: (1) that the function argument y may not appear in the 'then' part of the **if**-statement, (2) that the function f may not be recursively invoked in the 'conditional' or 'then' parts of the **if**, and (3) that the recursive call to f within the 'else' branch must be the argument to a particular function *op* having special properties. These restrictions are not explicit in any single transformation step, but rather are spread over the sequence of transformations embodied by the generalization. Realizing a similar result within a first-order system would be substantially complicated by, for example, the need for these explicit occurrence checks.

Apprentice learning. We believe that the search space for the above derivation is intractable; that is, without user guidance (*e.g.*, via an explicit meta-program), it would not be feasible for a system to 'discover' the sequence of transformations and the associated contexts with which to induce tail recursion. The problem space of program transformation is further complicated in that it is the user who decides when a derived program is acceptably 'efficient'—in this case, when it is tail-recursive. For transformation systems, we are not in the situation of theorem proving where there are only two answers—"yes, a goal is provable" or "no, it is not." Instead, the role of the user is two-fold: to guide the derivation and to make value judgments upon the resulting programs. Currently we are so far from automating the latter that transformation systems will continue to depend upon user assistance.

The fact that these value judgments are not part of the transformations means they are not manifest in the resulting generalizations. There is an important underlying assumption here: a sequence of transformations that leads to a 'good' program in one particular case (*e.g.*, *fact*) is presumed to do the same for other programs to which it is applicable (*e.g.*,

rev). However, as this 'goodness' exists outside of the transformation system, there is no guarantee that a derived rule indeed yields a 'good' program.[16]

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. In other words, under EBG nothing new may be proven, but the solution of problems covered by derived rules is (hopefully) quicker. With the incorporation of user interaction to address the problem of intractable search, this characterization of EBG becomes incorrect: the resulting generalizations, while in the deductive closure of the rule set, are generally *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 accomplishes speed-up learning. But, after a joint derivation of *fact*, the learner could handle *rev* without user assistance (presuming that the system could find the final simplification). That is, from the individual perspectives of the learner and user, more than speed-up learning has taken place (DeJong & Mooney, 1986; Dietterich, 1986, pp. 304–305; Mitchell, Mahadevan, & Steinberg, 1985; Mahadevan 1990).

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

Within this section we more formally describe $\lambda^{\Box}$Prolog and higher-order EBG through a pair of interpreters written in λProlog.

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

$$
\begin{aligned}
G &::= \mathbf{true} \mid A \mid G_1 \,,\, G_2 \mid G_1 \,;\, G_2 \mid D \Rightarrow G \mid \forall x\,[:\tau].\; G \mid \exists x\,[:\tau].\; G \mid \Box\, G_{\Box} \\
G_{\Box} &::= \mathbf{true} \mid A \mid G_{\Box 1} \,,\, G_{\Box 2} \mid \forall x\,[:\tau].\; G_{\Box} \mid \Box\, G_{\Box} \\
D &::= \mathbf{true} \mid A \mid D_1 \,,\, D_2 \mid D \Leftarrow G \mid \forall x\,[:\tau].\; D \mid \Box\, D \\
\mathcal{P} &::= \epsilon \mid D.\ \mathcal{P} \mid !!\ D.\ \mathcal{P}
\end{aligned}
$$

where the new meta-variable $G_{\Box}$ over ranges over 'boxed' goals, ϵ is the null terminal, and $\mathcal{P}$ ranges over logic programs (omitting type definitions, module declarations, *etc.*)

Although our examples have only employed $\Box$ at the top level, the operator is not restricted to outermost occurrences. The use of $\Box$ does not, however, extend to arbitrary λProlog constructs. In particular, $\lambda^{\Box}$Prolog disallows goals of the form $\Box(D \Rightarrow G)$, $\Box(\exists x.G)$, and $\Box(G_1 \,;\, G_2)$, because it is unclear how to give them an operational definition. It is also unclear what additional expressiveness would be provided.

$\lambda^{\Box}$Prolog does not distinguish sequences of the modal prefix; that is, $\Box\Box A$ is equivalent to $\Box A$. In this respect $\lambda^{\Box}$Prolog may be considered an *intuitionistic* version of the *classical* modal logic *S5* (Chellas, 1980). However, $\lambda^{\Box}$Prolog is a proper subset of *S5* as it lacks negation ($\neg$) and the second modal operator of *possibility* $\Diamond$, 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 theory.

Interpreting & generalizing $\lambda^{\Box}$Prolog. It is important to distinguish tl.e programming language $\lambda^{\Box}$Prolog from the algorithm that produces explanation-based generalizations of $\lambda^{\Box}$Prolog computation. To simplify discussion, we first present, in §8.1, a $\lambda^{\Box}$Prolog interpreter without the generalizing component. That interpreter is written in λProlog. Due the closeness of the correspondence between object-language ($\lambda^{\Box}$Prolog) and meta-language (λProlog), we often use the more descriptive term 'meta-interpreter.'

This meta-interpreter is extended to perform EBG within a second prototype in §8.2. The expanded meta-interpreter exemplifies the generalization algorithm admitted by $\lambda^{\Box}$Prolog, and has produced the examples contained within this paper. In §8.3 we augment this meta-interpreter to admit operationality criteria.

To run examples using the meta-interpreters to follow, the $\lambda^{\Box}$Prolog program $\mathcal{P}_{ob}$ to be interpreted must be available as data. This is accomplished by asserting **hyp** D for each clause D of $\mathcal{P}_{ob}$ prior to invoking the meta-interpreter. The prototype may then enumerate $\mathcal{P}_{ob}$ with λProlog's backtracking search (by successively solving the goal **hyp** D).[17] Variables of clauses asserted with **hyp** must be explicitly universally quantified. (The '!!' convention, while part of the eventual system, is not realizable within the prototype.) What follows is a portion of the ubiquitous suicide example in the form recognized by the meta-interpreter:

hyp ($\Box$ $\forall A\, \forall B\, \forall C$. **kill** A B $\Leftarrow$ **hate** A B, **possess** A C, **weapon** C).
hyp (**gun obj1**).

8.1. The meta-interpreter

The $\lambda^{\Box}$Prolog interpreter is divided between two sets of clauses: the **solve** predicates of Figure 7, which reduce a given $\lambda^{\Box}$Prolog goal G to some number of atomic subgoals (G_{a}'s), and the **match** predicates of Figure 8, which attempt to solve a pending atomic subgoal G_{a}.

```
wsolve  true          ⇐  !.
wsolve  (G1 , G2)     ⇐  !, wsolve G1, wsolve G2.
wsolve  (G1 ; G2)     ⇐  !, (wsolve G1; wsolve G2).
wsolve  (D ⇒ G)       ⇐  !, hyp D ⇒ wsolve G.
wsolve  (Π G)         ⇐  !, ∀x. wsolve (G x).
wsolve  (Σ G)         ⇐  !, ∃t. wsolve (G t).
wsolve  (□ G)         ⇐  !, ssolve G.
wsolve  Ga            ⇐  !, hyp D, wmatch D Ga Gs, wsolve Gs.

ssolve  true          ⇐  !.
ssolve  (G1 , G2)     ⇐  !, ssolve G1, ssolve G2.
ssolve  (Π G)         ⇐  !, ∀x. ssolve (G x).
ssolve  (□ G)         ⇐  !, ssolve G.
ssolve  Ga            ⇐  !, hyp D, smatch D Ga Gs, wsolve Gs.
```

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

```
wmatch  (D1 , D2)   Ga   Gs        ⇐ !, (wmatch D1 Ga Gs; wmatch D2 Ga Gs).
wmatch  (D ⇐ G)     Ga   (G , Gs)  ⇐ !, wmatch D Ga Gs.
wmatch  (Π D)       Ga   Gs        ⇐ !, wmatch (D Y) Ga Gs.
wmatch  (□ D)       Ga   Gs        ⇐ !, wmatch D Ga Gs.
wmatch  Da          Ga   true      ⇐ !, Da = Ga.

smatch  (D1 , D2)   Ga   Gs        ⇐ !, (smatch D1 Ga Gs; smatch D2 Ga Gs).
smatch  (D ⇐ G)     Ga   (G , Gs)  ⇐ !, smatch D Ga Gs.
smatch  (Π D)       Ga   Gs        ⇐ !, smatch (D Y) Ga Gs.
smatch  (□ D)       Ga   (□ Gs)    ⇐ !, wmatch D Ga Gs.
```

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

The goal reduction performed by **solve** is again split between two sets of clauses: **wsolve** for 'weak-solve' and **ssolve** for 'strong-solve.' This distinction arises from the more stringent proof required by the necessary truth of 'boxed' goals: for example, from the clause **p** we cannot derive the goal □ **p**, but the goal **p** does follow from the clause □ **p**. The top-level predicate is **wsolve**, because goals are contingent until a □ has been encountered. Each of the G_a's derived through **solve** will require either a 'strong' or 'weak' proof, which is realized through the corresponding **match** predicates—**wmatch** and **smatch**.

Within the **solve** predicates, solution of $\lambda^{\Box}$Prolog goals is largely realized by the corresponding λProlog constructs. For example, a $\lambda^{\Box}$Prolog conjunction (G_1, G_2) is derived by establishing the λProlog conjunction of G_1 and G_2, while a universally quantified $\lambda^{\Box}$Prolog goal is universally derived under λProlog. 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 connectives because of the additional restrictions placed upon boxed goals.)

In the final clauses of **wsolve** and **ssolve**, the pending goal has been reduced to an atomic G_a. This is insured by our use of the *cut* operator '!' of logic programming: if G_a instead contained a logical connective, '!' would have committed the interpretation to one of the preceding clauses.[18]

Through the predicate **hyp**, the final clauses of **wsolve** and **ssolve** select a potentially pertinent clause D from the program, which the **match** predicates then attempt to apply in the proof of G_a. The selection of D is 'naive' in that each clause of $\mathcal{P}_{ob}$ is simply tried in order until one is found that derives G_a. As we shall see, in the course of deriving G_a from D, **match** may produce subgoals (G_s's) that must be subsequently solved to complete the proof.

The **match** predicates analyze the selected program clause D to determine if it is applicable in the solution of G_a. For a conjunction (D_1, D_2), the logic programming paradigm dictates that either D_1 or D_2 individually derives G_a (although both D_1 and D_2 are available for the derivation of any resulting subgoals). A universally quantified clause Π D (or equivalently, Π $\lambda x.Dx$) is reduced by replacing the bound variable with a new logical variable Y,[19] which may become instantiated in the course of the proof: for example, the clause Π λz.**weapon** z ⇐ **gun** z becomes **weapon** Y ⇐ **gun** Y. If D is a rule D' ⇐ G', we conjoin G' with the subgoals that arise from establishing that D' implies G_a: for the clause **weapon** Y ⇐ **gun** Y, the interpreter first determines whether **weapon** Y establishes G_a, and then attempts to solve **gun** Y. When **smatch** encounters a □ in the

program, the nested clause need only be weakly matched with the current goal. This is because proving a goal 'strongly' simply requires that any utilized clauses must themselves be necessarily true. The resulting subgoal G_s is, however, boxed as it too must be strongly proved. On the other hand, **wmatch** ignores □'s within D, because we are therein only concerned with a weak proof.

In the final clause of **wmatch**, the unification of an atomic D_a and G_a is attempted: for example, unifying the goal **weapon obj1** with the clause **weapon** *Y*. This is analogous to the unification of a goal and clause head under a Prolog interpretation. If successful, this has the effect of 'returning' the accumulated conjunction of subgoals G_s (in this case, **gun obj1**) to the last clause of **solve**, which will then derive G_s recursively. The predicate **smatch** is, however, missing the analogue to the last clause of **wmatch**. This is because a contingent atomic clause cannot be used to prove a necessary atomic goal; that is, the clause **p** is not sufficient to derive □ **p**.

This concludes the discussion of the basic $\lambda^{\Box}$Prolog meta-interpreter. The next step is extending it to perform EBG.

8.2. The generalizing meta-interpreter

Kedar-Cabelli & McCarty produce first-order explanation-based generalizations within Prolog via an augmented meta-interpreter (Kedar-Cabelli & McCarty, 1987). As we shall take a similar approach, we briefly review Kedar-Cabelli & McCarty's implementation: Under its second formulation (pp. 387–388), their meta-interpreter, **prolog_ebg**, solves a particular query in parallel with the construction of the associated explanation-based generalization. The predicate **prolog_ebg** takes three arguments: the particular query *G*, the generalized query *GG*, and the conjunction of generalized conditions *DD* sufficient to derive *GG*.

Each 'rule' applied by **prolog_ebg** in the proof of *G* is similarly applied in the proof of *GG*. Leaves of the Prolog computation that arise in the course of deriving *GG* (*i.e.*, those goals established by 'facts') are accumulated in the conjunction of sufficient conditions *DD*. The resulting explanation-based generalization is then $GG \Leftarrow DD$, where for example

$$GG = \textbf{kill } X\ X$$
$$DD = \textbf{depressed } X, \textbf{ buy } X\ Y, \textbf{ gun } Y$$

No explicit representation of the proof need be constructed; it is inherent in the Prolog search.

As in the first-order approach of Kedar-Cabelli and McCarty (1987), our generalizing meta-interpreter develops two parallel proofs simultaneously: a proof of *G* and a generalized proof of *GG*. Again these proofs are not explicitly constructed; rather they are implicit in the λProlog search. In the course of deriving *G* and *GG*, the implementation accumulates the conjunction of generalized clauses *DD* sufficient to establish *GG*—that is, the leaves of the generalized proof. The resulting explanation-based generalization is then $!!\ GG \Leftarrow DD$.

```
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)       GG            DD            ⇐ !,  hyp D ⇒ wsolve G GG DD.
wsolve  (Π G)         (Π GG)        (DD X)        ⇐ !,  ∀X. wsolve (G X) (GG X) (DD X).
wsolve  (Σ G)         (Σ GG)        (DD T)        ⇐ !,  wsolve (G T) (GG T) (DD T).
wsolve  (□ G)         (□ GG)        DD            ⇐ !,  ssolve G GG DD.
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  (Π G)         (Π GG)        (DD X)        ⇐ !,  ∀X. ssolve (G X) (GG X) (DD X).
ssolve  (□ G)         (□ GG)        DD            ⇐ !,  ssolve G GG DD.
ssolve  Ga            GGa           (DD1 , DD2)   ⇐ !,  hyp D,
                                                        smatch D Ga DD1 GGa MG,
                                                        meta_wsolve MG DD2.
```

Figure 9. Generalizing meta-interpreter: Goal analysis.

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 where G and GG diverge. (An exception is made for the handling of implicational goals $D' \Rightarrow G'$, which is simplified by locally treating D' as a part of $\mathcal{P}_{ob}$.) The MG's (for 'meta-subgoal') in the final clauses of **solve** assume a role analogous to that played by subgoals in the previous meta-interpreter—that is, MG's retain subproof tasks for later derivation. The transition from the G_s's of the first interpreter to the current MG's comes out of the need to maintain both G and GG for subsequent solution. The straight-forward clauses **meta_wsolve** and **meta_ssolve** that derive MG's are given within Figure 11.

After **solve** selects a clause D with which to derive G_a, the extended **match** predicates of Figure 10 attempt to apply D is the solution of G_a. But in the course of deriving G_a, the new **match** also yields a generalized atomic goal GG_a and a generalized clause DD sufficient to derive GG_a. Within the final clause of **wmatch** where D_a is unified with G_a, DD is instead unified with GG_a. That neither the pair G_a and GG_a nor the pair D_a and DD are unified 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 GG or DD? The answer is that unless some of the D's employed in the proof are boxed, none ever will. In the matching of boxed clauses, D and DD are explicitly unified in the invocation of **bmatch** (for 'boxed-match'): within the suicide problem, for example, both D and DD are bound to **∀Z.weapon** Z ⇐ **gun** Z. (The additional predicate **bmatch** is required to handle subtle differences in the matching of instantiated DD's.) While D and DD are initially equivalent within **bmatch**, they may later diverge as distinct new logical variables X and Y are substituted for universally quantified programs. This is because D is to be unified with G_a, while DD is to be unified with GG_a: again for the **weapon** clause, D's logical variable becomes bound to **obj1**, while that of DD remains uninstantiated.

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

```
wmatch  (D1 , D2)   Ga  DD          GGa  MG          ⇐ !,  (wmatch D1 Ga DD GGa MG
                                                           ; wmatch D2 Ga DD GGa MG).
wmatch  (D ⇐ G)     Ga  (DD ⇐ GG)   GGa  (mg G GG,
                                          MG)       ⇐ !,  wmatch D Ga DD GGa MG.
wmatch  (Π D)       Ga  DD          GGa  MG          ⇐ !,  wmatch (D X) Ga DD GGa MG.
wmatch  (□ D)       Ga  (□ 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  (D ⇐ G)     Ga  (DD ⇐ GG)   GGa  (mg G GG,
                                          MG)       ⇐ !,  smatch D Ga DD GGa MG.
smatch  (Π D)       Ga  DD          GGa  MG          ⇐ !,  smatch (D X) Ga DD GGa MG.
smatch  (□ D)       Ga  (□ D)       GGa  (□ 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  (D ⇐ G)     Ga  (DD ⇐ GG)   GGa  (mg G GG,
                                          MG)       ⇐ !,  bmatch D Ga DD GGa MG.
bmatch  (Π D)       Ga  (Π DD)      GGa  MG          ⇐ !,  bmatch (D X) Ga (DD Y) GGa MG.
bmatch  (□ D)       Ga  (□ D)       GGa  MG          ⇐ !,  bmatch D Ga D GGa MG.
bmatch  Ga          Ga  GGa         GGa  true.
```

Figure 10. Generalizing meta-interpreter: Clause analysis.

```
meta_wsolve  true          true          ⇐ !.
meta_wsolve  (MG1 , MG2)   (DD1 , DD2)   ⇐ !,  meta_wsolve MG1 DD1,
                                               meta_wsolve MG2 DD2.
meta_wsolve  (□ MG)        DD            ⇐ !,  meta_ssolve MG DD.
meta_wsolve  (mg 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  (□ MG)        DD            ⇐ !,  meta_ssolve MG DD.
meta_wsolve  (mg G GG)     DD            ⇐ !,  ssolve G GG DD.
```

Figure 11. Generalizing meta-interpreter: Meta-Goal solution.

generalization. However, boxed clauses are 'necessarily' true, and hence need not be re-checked during the application of a derived rule. Instead, it is the conjunction of utilized unboxed clauses which constitutes the simplest expression of the sufficient conditions for *GG*. Removing boxed clauses from *DD* requires a simple reduction predicate **reduce**, which replaces arbitrary occurrences of □ *P* by **true** within *DD*. (As **reduce** is relatively trivial, we omit its definition; instead see Dietzen (1991).)

Explanation-based generalizations may then be derived as follows:

```
do_ebg  G  (GG ⇐ DD')  ⇐  wsolve G GG DD,
                          reduce DD DD'.
```

8.3. Operationality

Incorporating operationality criteria within the preceding prototype requires providing the meta-interpreter with access to an operationality predicate **oper**. The revision involves inserting the following clause at the head of the **solve** predicates. We illustrate the change for **wsolve**; an analogous change is necessary in **ssolve**:

$$\textbf{wsolve}\ \ G\ \ GG\ \ DD \Leftarrow \textbf{oper}\ G,\ !,\ DD = GG,\ \textbf{wsolve\_orig}\ G.$$

where **wsolve_orig** is the version of **wsolve** that does not perform EBG—*i.e.*, that given within Figure 7.[20] The computation proceeds in the same manner, but EBG is suspended during the solution of operational subgoals. Instead, *DD* is bound to the current generalized goal *GG*, which, because it is operational, becomes one of the sufficient conditions of the resulting generalization. The above clause is expected to be used for recursive invocations of **solve**—*i.e.*, during the solution of subgoals; if a top-level goal is made operational, the resulting explanation-based generalization is trivial.

It is the user'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 the original. Moreover, this formulation of operationality is *dynamic*, in that **oper** may be defined and redefined within the course of the computation (Hirsh, 1988).[21]

8.4. Direct implementation

The above $\lambda^{\Box}$Prolog implementation in λProlog has been extremely valuable for experimenting with different variations of $\lambda^{\Box}$Prolog and the EBG algorithm, and further for providing a formal specification of each. 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 !! convention,[22] or to implement primitives for controlling EBG as well as those for manipulating and assimilating rules derived through EBG. (Again, a discussion may be found in Dietzen (1991).) We have addressed the above deficiencies by extending our existing λProlog interpreter, eLP, to realize $\lambda^{\Box}$Prolog and EBG.

9. Conclusion

To date, the application of the machine learning technique of explanation-based generalization has largely been limited to first-order representation languages. Such encodings do not lend themselves to the natural and concise expression and manipulation of higher-order

objects such as functions and predicates. To facilitate generalization over these higher-order values, we have expanded the paradigm to higher-order explanation-based generalization, and further have provided a formal characterization of higher-order EBG through the higher-order logic programming language λProlog and concepts of modal logic. Potential applications include learning systems that manipulate programs, logical formulas, mathematical expressions, and natural language.

Our presentation herein has focused upon the formulation and illustration of higher-order EBG within the framework of $\lambda^{\Box}$Prolog. In order to derive and exploit these generalizations, there must be mechanisms for initiating EBG within $\lambda^{\Box}$Prolog (in our view, under the programmer's direction), and also for extending the existing logic program with newly derived clauses. The latter consideration is problematic in that the standard construct by which Prolog programs are extended, **assert**, is not semantically well-behaved. For this and other reasons, **assert** is not part of λProlog. Elsewhere, we propose new constructs, **rule** and **rule_ebg**, that provide for *programmable* generalization and assimilation (§6), and that offer a straightforward semantics reconcilable with λProlog (Dietzen, 1991).

Broadly speaking, then, our work should be viewed as a language design effort. This distinction is fundamentally important to the evaluation of our efforts. Unlike typical 'stand alone' learning systems, $\lambda^{\Box}$Prolog does not pose its own learning problems. Instead, by integrating learning mechanisms within the programming language, we defer one of the most difficult problems faced by a 'learner': determining over what computations to attempt learning, or in other words, determining when to learn. While $\lambda^{\Box}$Prolog is not itself a learning system, it is intended to serve as a high-level foundation for the implementation of such systems.

Many questions remain, but perhaps the predominant one is whether a relatively complete, higher-order learning system can be effectively realized within $\lambda^{\Box}$Prolog. While we have provided example scenarios—both of EBG's direct use to reduce λProlog search, and of its role within an apprentice learner to encapsulate the results of interactive problem solving—we have not yet produced such a system. This is largely due to limitations of our present implementation, both in performance and in functionality (*e.g.*, its lack of a mouse interface) (Dietzen, 1991). Also of particular interest to the authors is the further development of the 'language-based' approach to learning (of which $\lambda^{\Box}$Prolog is an exemplar) to encompass other EBG methodologies (*e.g.*, generalizing iterative and recursive theories, (Cohen, 1988; Shavlik, 1990)) and other paradigms of generalization (*e.g.*, similarity-based methods, (Hirsh, 1989)).

Appendix A. Tail recursion via program transformation

As it may be of interest to the less casual reader, this section illustrates the nature of program transformation by enumerating the individual steps of the tail-recursive factorial derivation. The actual higher-order transformations and meta-program for applying them may be found within Dietzen (1991).

0. We begin with the initial definition of *fact*.

$$\begin{array}{l}\textbf{fix}\ \lambda \mathit{fact}.\ \textbf{lam}\ \lambda n.\ \textbf{if}\ (\textbf{equals}\ n\ 0)\\ \qquad\qquad\qquad\qquad 1\\ \qquad\qquad\qquad\qquad (\textbf{appl}\ \mathit{fact}\ (n-1)) * n\end{array}$$

1. η-expand term in the object-language; that is, insert a **lam** and an **appl**. ('. . .' elides the body of *fact*.)

$$\begin{array}{l}\textbf{lam}\ \lambda n.\ \textbf{appl}\ (\textbf{fix}\ \lambda \mathit{fact}.\ \ldots)\\ \qquad\qquad\qquad\qquad n\end{array}$$

(The above n is distinct from that within *fact*'s body.)

2. Insert a multiplication by 1. This transformation relies upon **right_identity** $(\lambda x.\lambda y.x * y)$ 1.

$$\begin{array}{l}\textbf{lam}\ \lambda n.\ (\textbf{appl}\ (\textbf{fix}\ \lambda \mathit{fact}.\ \ldots)\\ \qquad\qquad\qquad\qquad n) * 1\end{array}$$

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

$$\begin{array}{l}\textbf{appl}\ (\textbf{lam}\ \lambda m.\ \textbf{lam}\ \lambda n.\\ \qquad (\textbf{appl}\ (\textbf{fix}\ \lambda \mathit{fact}.\ \ldots)\\ \qquad\qquad n) * m)\\ \quad 1\end{array}$$

4. Name the resulting two argument function $fact_1$: since **fix** specifies the expansion of recursive functions, one may think of it a mechanism for function definition. This initial definition of $fact_1$ will be used later in the derivation.

$$\begin{array}{l}\textbf{appl}\ (\textbf{fix}\ \lambda \mathit{fact}_1.\ \textbf{lam}\ \lambda m.\ \textbf{lam}\ \lambda n.\\ \qquad (\textbf{appl}\ (\textbf{fix}\ \lambda \mathit{fact}.\ \ldots)\\ \qquad\qquad n) * m)\\ \quad 1\end{array}$$

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

$$\begin{array}{l}\textbf{appl}\ (\textbf{fix}\ \lambda \mathit{fact}_1.\ \textbf{lam}\ \lambda m.\ \textbf{lam}\ \lambda n.\\ \qquad (\textbf{appl}\ (\textbf{lam}\ \lambda n'.\ \mathit{if}\ (\textbf{equals}\ n'\ 0)\\ \qquad\qquad\qquad\qquad 1\\ \qquad\qquad\qquad\qquad ((\textbf{appl}\ (\textbf{fix}\ \lambda \mathit{fact}.\ \ldots)\ n'-1) * n'))\\ \qquad\qquad n) * m)\\ \quad 1\end{array}$$

6. β-reduction in the object-language; that is, **appl** (**lam** $\lambda n'.\ Fn'$) $n \Rightarrow_\beta Fn$.

```
appl (fix λfact1. lam λm. lam λn.
        (if (equals n 0)
           1
           ((appl (fix λfact. . . .) n − 1) * n))
        * m)
     1
```

7. Distribute $*$ over **if**.

```
appl (fix λfact1. lam λm. lam λn.
        if (equals n 0)
          1 * m
          ((appl (fix λfact. . . .) n − 1) * n) * m)
     1
```

8. Simplify the *then*-clause using the fact that **left_identity** $(\lambda x.\lambda y.\ x * y)$ 1.

```
appl (fix λfact1. lam λm. lam λn.
        if (equals n 0)
          m
          ((appl (fix λfact. . . .) n − 1) * n) * m)
     1
```

9. Reassociate the multiplicative expression of the *else*-clause, since **associative** $\lambda x.\lambda y.\ x * y$.

```
appl (fix λfact1. lam λm. lam λn.
        if (equals n 0)
          m
          ((appl (fix λfact. . . .) n − 1) * (n * m))
     1
```

10. Observe that within step 9 the subexpression

```
(appl (fix λfact. . . .) n − 1) * (n * m))
```

is a higher-order instance of the original definition of $fact_1$ given in step 4:

```
fix λfact1. lam λm. lam λn.
   (appl (fix λfact. . . .)
         n) * m)
```

The only difference is the values of the arguments m and n. This means that we may *fold* the above expression into a $fact_1$ invocation.

$$\textbf{appl}\ (\textbf{fix}\ \lambda fact_1.\ \textbf{lam}\ \lambda m.\ \textbf{lam}\ \lambda n.\ \textbf{if equals}\ n\ 0\ m\ (\textbf{appl}\ (\textbf{appl}\ fact_1\ (n * m))\ (n - 1)))\ 1$$

This completes the derivation.

Acknowledgments

Our meta-interpreter depends on eLP, the implementation of λProlog developed by Conal Elliott and the second author in the framework of the Ergo project at Carnegie Mellon University (Elliott & Pfenning, 1989). eLP is written in COMMON LISP and relies upon syntax tools of the Ergo Support System (Lee, et al., 1988). A substantially more efficient implementation of λProlog within Standard ML is currently being completed by the second author.

We thank Penny Anderson, Conal Elliott, Masami Hagiya, Haym Hirsh, Dale Miller, Jack Mostow, Tom Mitchell, William Scherlis, and the anonymous referees for their thoughtful comments on our presentation.

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.

Notes

1. For an introduction to modal logic, see Chellas (1980).
2. Del Cerro offers another approach to incorporating modal logic within the logic programming framework that has nothing to do with EBG (del Cerro, 1986; del Cerro & Penttonen, 1987). For treatments of automated theorem proving in modal logics outside of logic programming, see Wallen (1987) and Thistlewaite, McRobbie and Meyer (1988).
3. To accomplish this, some EBG systems employ the trick of writing the clause as **adjacent** $X\ X \Leftarrow$ **true**.
4. There are already many such interpretations: $\Box$ can stand for 'formally provable', or for truth in all reachable worlds in a Kripke semantics, (Hughes & Cresswell, 1968), *etc.*
5. Keller reviews existing formulations of operationality (Keller, 1988).
6. Perhaps one reason that the partition between domain and training theory has not received more consideration within the literature is that the mechanism of operationality typically precludes ∃-clauses from entering into generalized proofs.
7. Knight presents an overview of unification research in Knight (1989).
8. For a discussion of logic programming with constraints, see Jaffar and Lassez (1987).
9. Donat & Wallen also address the step from first- to higher-order EBG over the domain of symbolic integration (Donat & Wallen, 1988). Higher-order generalization allows extremely general rules to be extracted from particular problem solutions. Our work focuses on how to control EBG to avoid over-general generalizations, and yet at the same time take advantage of the higher-order nature of the language. Donat & Wallen's work concentrates on how one could still usefully apply very general learned rules. To that end they introduce some control constructs into the higher-order unification process.

10. Donat and Wallen's approach (1988) instead utilizes a first-order representation of integrals (from which they produce higher-order generalizations). This first-order encoding requires additional constraints manifest in the **constant** primitive, which pervades their derived rules and which is avoided within our higher-order encoding.
11. Our tactic-based approach to integration may be considered a rudimentary theorem prover. In fact, λProlog has much to generally suggest it as a language for implementing theorem provers (Felty & Miller, 1988): λProlog combines logic programming's support for search and unification with the expressiveness afforded by higher-order language and additional logical connectives. And, of course, theorem proving tasks formulated within $\lambda^{\Box}$Prolog admit EBG.
12. For a discussion of these issues see Prieditis and Mostow (1987, pp. 496–497), Minton (1988), and Donat and Wallen (1988).
13. For the same reasons that suggest it as a language in which to write theorem provers, λProlog is an attractive implementation language for formal program development tools (Hannan & Miller, 1988; Miller & Nadathur, 1987).
14. Hill also considers the application of EBG to the domain of program development (Hill, 1987). However, Hill's research utilizes a first-order encoding, and focuses upon a particular application within formal programming: the generalization of abstract datatype representations. Our work is directed, instead, toward the realization of a common language, $\lambda^{\Box}$Prolog, in which a multiplicity of programming and theorem proving methodologies can be realized.
15. While the derivation never establishes that $a \equiv b$, this follows from the fact that $a \equiv (op\ a\ b) \equiv b$ using **right_identity** *op b* and **left_identity** *op a*.
16. We are grateful to Jack Mostow for this observation, (Mostow, 1989).
17. Within Prolog the recall of $\mathcal{P}_{ob}$ could be handled directly through the **clause** predicate, but λProlog does not currently provide this functionality.
18. A thorough discussion of cut may be found in Sterling and Shapiro (1986).
19. As in Prolog, a top-level λProlog clause (in this case, **smatch**) is implicitly universally quantified over its variables. In order to apply that clause, the interpreter creates an instance by replacing those universal variables (*e.g., Y*) with new logical variables, thereby facilitating the unification of the clause head with the pending goal.
20. To maintain the separation of meta- and object-levels, the definition of **oper** is more elegantly included within $\mathcal{P}_{ob}$, and hence the determination of operationality is expressed as **wsolve_orig** (**oper** *G*).
21. Since dynamic operationality criteria are subject to change, derived rules often incorporate some representation of the utilized criteria to insure the continued integrity of those rules. This formulation of operationality, however, does not admit the expression of these preconditions.
22. In fact !! 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. Suppose we derived the rule $\mathbf{p}\,x$. The problem is that $\mathbf{p}\,x \Rightarrow (\mathbf{p\ a}, \mathbf{p\ b})$ is not true since in the course of proving $\mathbf{p\ a}$, x is instantiated to **a**. Of course, it is the case that $(\forall x.\ \mathbf{p}\ x) \Rightarrow (\mathbf{p\ a}, \mathbf{p\ b})$, because the universal quantification allows x to be multiply instantiated. But λProlog provides no mechanism by which existing free variables (such as x) can be captured with an inserted quantifier.

References

Angluin, D. & Smith, C.H. (1983). Inductive inference: Theory and methods. *Computing Surveys, 15*, 237–269.

Bhatnagar, N. (1988). A correctness proof of explanation-based generalization as resolution theorem proving. *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning* (pp. 220–225).

Burstall, R.M. & Darlington, J. (1977). A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery, 24*, 44–67.

Chellas, B.F. (1980). *Modal logic: An introduction*. Cambridge University Press.

Cohen, W.W. (1988). Generalizing number and learning from multiple examples in explanation-based learning. *Proceedings of the Fifth International Machine Learning Conference.*

Constable, R.L., et al. (1986). *Implementing mathematics with the Nuprl Proof Development System.* Englewood Cliffs, NJ: Prentice-Hall.

DeJong, G. & Mooney, R. (1986). Explanation-based generalization: An alternate view: *Machine Learning, 1,* 145-176.

del Cerro, L.F. (1986). Molog: A system that extends Prolog with modal logic. *New Generation Computing, 4,* 35-50.

del Cerro, L.F. & Penttonen, M. (1987). A note on the complexity of the satisfiability of modal Horn Clauses. *Journal of Logic Programming, 4,* 1-10.

Dietterich, T.G. (1986). Learning at the knowledge level. *Machine Learning, 1,* 287-315.

Dietterich, T.G. et al., (1982). Learning and inductive inference. In P.R. Cohen & E.A. Feigenbaum, (Eds.) *The handbook of artificial intelligence,* volume 3, pp. 325-511. William Kaufmann.

Dietzen, S. (1992). *A language for higher-order explanation-based learning.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Available as Technical Report CMU-CS-92-110.

Dietzen, S. & Pfenning, F. (1989). Higher-order and modal logic as a framework for explanation-based generalization (Extended abstract). In A.M. Segre, (Ed.), *Sixth International Workshop on Machine Learning,* pp. 447-449. San Mateo, California: Morgan-Kaufmann Publishers.

Dietzen, S. & Scherlis, W. (1987). Analogy in program development. In J.C. Boudreaux, B.W. Hamill & R. Jernigan, (Ed.) *The role of language in problem solving 2,* pp. 95-117. North-Holland. Also available as Ergo Report 86-013, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Donat, M.R. & Wallen, L.A. (1988). Learning and applying generalized solutions using higher order resolution. In E. Lusk & R. Overbeek (Eds.) *Ninth International Conference on Automated Deduction, Argonne, Illinois,* pp. 41-60. Berlin: Springer-Verlag LNCS 310.

Elliott, C. & Pfenning, F. (1989). eLP: A Common Lisp implementation of λProlog in the Ergo Support System. Available via ftp over the Internet. Send mail to elp-request@cs.cmu.edu on the Internet for further information.

Elliott, C.M. (1990). *Extensions and applications of higher-order unification.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Available as Technical Report CMU-CS-90-134.

Ellman, T. (1989). Explanation-based learning: A survey of programs and perspectives. *ACM Computing Surveys, 21,* 163-221.

Feather, M.S. (1986). A survey and classification of some program transformation approaches and techniques. *IFIP TC2 Working Conference on Program Specification and Transformation.* North-Holland.

Felty, A. & Miller, D.A. (1988). Specifying theorem provers in a higher-order logic programming language. In E. Lusk & R. Overbeek (Eds.), Ninth International Conference on Automated Deduction, Argonne, Illinois, pp. 61-80. Berlin: Springer-Verlag LNCS 310.

Goldfarb, W.D. (1981). The undecidability of the second-order unification problem. *Theoretical Computer Science, 13,* 225-230.

Gordon, M.J., Milner, R. & Wadsworth, C.P. (1979). *Edinburgh LCF.* Springer-Verlag LNCS 78.

Hannan, J. & Miller, D. (1988). Uses of higher-order unification for implementing program transformers. In R.A. Kowalski & K.A. Bowen (Eds.) *Logic programming: Proceedings of the Fifth International Conference and Symposium, Volume 2,* pp. 942-959. Cambridge, MA: MIT Press.

Harper, R., Honsell, F. & Plotkin, G. (1987). A framework for defining logics. *Symposium on logic in computer science,* pp. 194-204. IEEE. An extended and revised version is available as Technical Report CMU-CS-89-173, School of Computer Science, Carnegie Mellon University.

Hill, W.L. (1987). Machine learning for software reuse. *Proceedings of IJCAI* (pp. 338-344).

Hindley, J.R. & Seldin, J.P. (1986). *Introduction to combinators and λ-calculus.* Cambridge University Press. London Mathematical Society Student Texts: 1.

Hirsh, H. (1987). Explanation-based generalization in a logic-programming environment. *Proceedings of IJCAI* (pp. 221-227).

Hirsh, H. (1988). Reasoning about operationality for explanation-based learning. *Proceedings of the Fifth International Machine Learning Conference.*

Hirsh, H. (1989). *Incremental version-space merging: A general framework for concept learning.* Ph.D thesis, Stanford University, Stanford, CA.

Huet, G. (1975). A unification algorithm for typed λ-calculus. *Theoretical Computer Science, 1,* 27-57.

Huet, G. & Lang B. (1978). Proving and applying program transformations expressed with second-order patterns. *Acta Informatica, 11,* 31-55.

Hughes, G.E. & Cresswell, M.J. (1968). *Introduction to modal logic.* London: Methuen and Co., Ltd.

Jaffar, J. & Lassez, J.-L. (1987). Constraint logic programming. *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich* (pp. 111-119). ACM Press.

Kedar-Cabelli, T. & McCarty, L.T. (1987). Explanation-based generalization as resolution theorem proving. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 383–389). Also available as Technical Report ML-TR-10, Department of Computer Science, Rutgers University.

Keller, R. (1988). Defining operationality for explanation-based learning. *Artificial Intelligence, 35,* 227–241.

Knight, K. (1989). Unification: A multi-disciplinary survey. *ACM Computing Surveys, 21,* 93–124.

Laird, J.E., Rosenbloom, P.S. & Newell, A. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33,* 1–64.

Lee, P., et al. (1988). The Ergo Support System: An integrated set of tools for prototyping integrated environments. In P. Henderson (Ed.) *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (pp. 25–34). ACM Press. Also available as Ergo Report 88–054, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Mahadevan, S. (1990). *An apprentice-based approach to learning problem-solving knowledge* (Technical Report ML-TR-30). New Brunswick, N.J.: Rutgers University. Ph.D. Dissertation.

Miller, D. (1990). A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, (Ed.). *Extensions of logic programming* (pp. 253–282). Springer-Verlag LNCS475.

Miller, D. & Nadathur, G. (1986). Some uses of higher-order logic in computational linguistics. *Proceedings of the Twenty-Fourth Annual Meeting of the Association for Computational Linguistics* (pp. 247–255).

Miller, D.A. & Nadathur, G. (1987). A logic programming approach to manipulating formulas and programs. *Symposium on Logic Programming. San Francisco*: IEEE.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of AAAI* (pp. 564–569).

Minton, S. et al. (1989). Explanation-based learning: A problem-solving perspective. *Artificial Intelligence, 40,* 63–118. Special issue on machine learning. Also available as Technical Report CMU-CS-89-103, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Mitchell, T.M. Keller, R.M. & Kedar-Cabelli, S.T. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1,* 47–80.

Mitchell, T.M. Mahadevan, S. & Steinberg, L. (1985). LEAP: A learning apprentice for VLSI design. *Proceedings of IJCAI* (pp. 573–580). Morgan Kaufmann.

Mostow, J. (1989). Design by derivational analogy: Issues in the automated replay of design plans. *Artificial Intelligence, 40,* 119–184. Special issue on machine learning.

Nadathur, G. & Miller, D. (1988). An overview of λProlog. In R.A. Kowalski & K.A. Bowen (Eds.) *Logic programming: Proceedings of the Fifth International Conference and Symposium, Volume 1* (pp. 810–827). Cambridge, MA: MIT Press.

Partsch, H. & Steinbrüggen, (1983). Program transformation systems. *Computing Surveys, 15,* 199–236.

Pereira, F.C.N. (1990). Prolog and natural-language analysis: Into the third decade. In S. Debray & M. Hermenegildo, (Eds.). *Proceedings of the North American Conference on Logic Programming* (pp. 813–832). MIT Press.

Pfenning, F. & Elliott, C. (1988). Higher-order abstract syntax. *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia* (pp. 199–208). ACM Press. Available as Ergo Report 88–036, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Prieditis, A.E. & Mostow, J. (1987). Prolearn: Toward a Prolog interpreter that learns. *Proceedings of AAAI.*

Shavlik, J. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning, 5,* 39–70.

Sterling, L. & Shapiro, E. (1986). *The art of Prolog: Advanced programming techniques.* MIT Press.

Thistlewaite, P.B., McRobbie, M.A. & Meyer, R.K. (1988). *Automated theorem-proving in non-classical logics.* London: Pitman.

Wallen, L.A. (1987). *Automated proof search in non-classical logics: Efficient matrix proof methods for modal and intuitionistic logics.* Ph.D. thesis, University of Edinburgh.