Machine Learning for Software Reuse

Walter L. Hill

Hewlett-Packard Laboratories
P.O. Box 10490, Palo Alto CA 94303-0971
(415) 857-7310
whill@hplabs.hp.com

Abstract

Recent work on learning apprentice systems suggests new approaches for using interactive programming environments to promote software reuse. Methodologies for software specification and validation yield natural domains of application for explanation-based learning techniques. This paper develops a relation between data abstractions in software and explanation-based generalization problems and shows how explanation-based learning can be used to generalize program abstractions to promote their reuse. This method is applied in the design of a system called LASR (Learning Apprentice for Software Reuse) which will acquire programming knowledge by capturing and generalizing interconnections between abstract data type theories. The technical role of theories in *defining* learned concepts in this application suggests their more general use in representing problems in explanation-based learning.

1. Introduction

There has been considerable attention given recently to making use of formal program specifications to promote software reuse and to developing programming methodologies suited to that purpose [4,12]. Among the requirements for software to be reusable are that it possess higher levels of robustness and generality than are usual in ordinary programming practice. The purpose of this paper is to show how explanation-based learning can be used with formal specifications to capture and generalize program abstractions developed in practice to increase their potential for reuse. This particular approach to applying machine learning in software is motivated especially by the explicit domain knowledge embodied in data type specifications and the mechanisms for reasoning about such knowledge used in validating software. These deductive methods from software engineering fit well with the requirements for explanation-based learning problems in which a single training example is explained (validated) in terms of a domain theory. The training examples in our setting correspond to (semantically correct) interconnections between software components. In generalizing from these interconnections, new data abstractions are formed which yield minimal requirements for reuse. Such synthesized abstractions can also be useful in their own right in suggesting alternatives in program derivations.

The software principles involved in this application are more abstract than those most commonly used in programming practice. It is generally hoped that time will close this gap. Nevertheless, the level of abstraction seems most appropriate both for this application of machine learning and also for promoting advances in software reuse. The discussion and examples given below attempt to show that the implications of those principles are in fact quite concrete and relevant. The notion of a *theory* is of central importance in much modern software research. Mechanisms for composing theories support construction of complex abstractions. It turns out to be natural in this application of explanation-based learning to define the *learned concepts* in terms of theories. This theory interpretation of learned concepts seems applicable in representing other explanation-based learning problems.

It is desirable to incorporate these methods for generalizing software with an interactive programming environment such as Goguen describes in [4]. While supporting practical system development, such a system would automatically add new generalized components to its software base for reuse. Such a system would be a learning apprentice in the sense of [10]. This paper also describes the design of a system called LASR (Learning Apprentice for Software Reuse) which is under development at Hewlett-Packard. Its purpose is to determine the potential for applying the ideas presented here to practical software engineering.

2. Learning from Explanations

Explanation-based learning is a comparatively recent paradigm in machine learning concerned with generating concepts using an explanation of a single training example in terms of a domain theory. The papers [3] and [11] contain extensive discussion and references to work in explanation-based learning. The methods of explanation-based generalization developed in [11] in particular derive a concept definition by analyzing a proof which accounts for the training example as a logical consequence of domain axioms. It is the use of deductive reasoning in a formal domain which gives a basis for applying explanation-based learning to software.

The explanation-based approach to machine learning is often contrasted with so-called similarity-based methods which derive conceptual classifications by noticing patterns in multiple observations. Similarity-based learning is more computationally intensive, typically involving searching in a large space of possible concepts (e.g., combinations of constraints on feature values). It is data-

intensive, and its results are always subject to revision since further observations may invalidate empirical generalizations. Similarity-based learning is essential, however, especially in the absence of sufficient domain knowledge. In [2], similarity-based learning is applied to program synthesis by considering learned concept predicates as declarative computer programs.

In using an explanation to generalize a training example, one obtains a characterization of a family of examples of a more general goal concept, including the given example. The learned concept definition is thus a specialization of that goal concept. The process of specialization (goal regression) transforms the goal concept over inference steps in the explanation until it is "operational". Operationality of a concept definition depends on the particular generalization problem. In the application of these methods to software, we define operationality in terms of levels of abstract machine. Following [11], Figure 1 shows the requirements for an explanation-based generalization problem. For such a problem, the generalization process consists of first constructing an explanation of the training example in terms of the domain theory, and then using that explanation to accomplish the generalization.

Given:
* Goal Concept
  A concept definition describing the concept to be learned.
* Training Example
  An example of the goal concept.
* Domain Theory
  A set of rules and facts to be used in explaining how the Training Example is an example of the Goal Concept.
* Operationality Criterion
  A predicate over concept definitions, specifying the form in which the learned definition must be expressed.

Determine:
* A generalization of the training example that is a sufficient concept definition for the Goal Concept and that satisfies the Operationality Criterion.

Figure 1. Explanation-Based Generalization Problem

We describe below examples of explanation-based generalization problems in designing software and mechanisms for solving them automatically. For further discussion and examples, see [11].

3. Domain Theories in Software

In order to apply explanation-based learning to software, it is necessary to associate goal concepts and domain theories with computer programs. In doing so, it is reasonable to expect relationships between goal concepts and program abstractions, and between explanation and program verification. One rather obvious approach to applying explanation-based learning to procedural abstractions in software is to use Hoare-style verifications to derive the characteristic behaviors of procedures in imperative languages, notably in terms of weakest preconditions. Under various names, there is already a considerable literature on this subject in both the software and AI communities [7,11,14]. While that approach does merit elaboration from the point of view of machine learning, the goal of this paper is to discover other applications of explanation-based learning in software which seem more enlightening and more promising for future developments both in software engineering[1] and machine learning.

In modern programming methodology, formal theories of data abstractions in programs play an important role in designing and validating software and in promoting its reuse [4,12]. Such theories are represented explicitly in languages such as OBJ [6] and the Larch shared language [8]. As will be shown below, these theories give rise naturally to applications of explanation-based learning by providing domain theories for explanation-based generalization problems. The *theories* discussed here, so-called many-sorted equational theories, correspond to collections of abstract data types. Such a *theory* consists of finite sets of sorts (the "data types"), operations on them, and equations relating those operations. Those equations, or equational axioms, provide a logical theory completely characterizing their associated "data types". Figure 2 gives a simple presentation of a theory of stacks. The presentation of a theory can be thought of as having two parts. The first declares the operations and their types; such declarations are conventional in the specification parts of modules in languages like Ada and Modula-2. The second part provides the semantic constraints which the operations satisfy. Operations are side-effect free; in Figure 2 it is necessary to provide separate top and pop operations.

```
THEORY Stack

SORTS  stack element

OPS    empty: -> stack
       push: element stack -> stack
       pop: stack -> stack
       top: stack -> element
       empty?: stack -> boolean

VARS   e: element
       s: stack

EQNS   pop(empty) = empty
       pop(push(e,s)) = s
       top(push(e,s)) = e
       empty?(empty) = true
       empty?(push(e,s)) = false
```

Figure 2. Data Theory of Stacks

1. We have learned quite recently (subsequent to submitting this paper) of current work by C.A.R. Hoare, He Jifeng, and J.W. Sanders [15-17] which, although not employing explanation-based methods, contains ideas quite similar to some of those presented here.

To make data theories useful in constructing programs, there are mechanisms for composing and refining them. A particularly important construct in this regard is the *theory morphism*. A *theory morphism* from a theory T1 to a theory T2 maps sorts and operations of T1, respectively, to sorts and operations of T2. Moreover, it preserves the equations of T1 in the sense that each axiom of T1, when rewritten in terms of the sorts and operations of T2, can be deduced in T2. We will call T1 the *source theory* and T2 the *target theory*.

Figure 4[2] gives an example of a morphism from the theory of stacks to the theory of arrays with a distinguished index. Note that this morphism *implements* stack operations in a (programming) language based on arrays and natural numbers. From that point of view, the morphism plays the role of a program, the theories are specifications, and the proof that the axioms are preserved is a validation of the program. We will see how this leads naturally to a generalization of the program using the validation as explanation.

The example in Figure 5 of the Integer-Array theory can be interpreted as an instance of a parameterized Array theory with an unconstrained "element" sort analogous to the one in Figure 2. It turns out [4] that theory morphisms in general provide the bindings for instantiating generic parameters. What Goguen calls *views* in [4] are essentially theory morphisms.

4. Using Validations to Generalize Software

In adapting explanation-based generalization to data abstractions, the data theories provide domain theories for explaining explicit morphisms, which take the role of training examples. Concept definitions in this setting are presentations of target data theories of morphisms. Intuitively, such concept definitions specify a language for interpreting (or implementing[3]) the morphism's source theory. The object of generalization is to use a proof that a particular interpretation is valid to reduce the language (indeed, its semantics as well as its syntax) to one which provides a minimal valid interpretation. In other words, the generalization provides requirements for an interpreting language.

2. A little care must be taken in interpreting this definition. The target theory is obtained by extending the I-Array theory to make the OPS rules shown in Figure 4 into equational axioms. This requires adding new operations to the theory corresponding to push, pop, etc. We call the operation <_,_> and the operations imported from theories Nat and Array *primitive*. Primitive operations are used to characterize operationality. It is conventional in presenting morphisms to suppress obvious correspondences, e.g. (element => element) associating stack elements with array elements.

3. An implementation or *data refinement* is a morphism which has reasonable behavior on models for its source and target theories. While any morphism can be generalized, many readers will find the notion of implementation more intuitive.

What most distinguishes this form of explanation-based learning is that the learned concepts are presented explicitly in terms of theories; there is no difference in kind between the domain theory used for the explanation and the new target theory obtained from generalization. Nevertheless, the differences between this approach and, for example, that of [11] are only in interpretation; the learned concepts in the examples in that paper are described in terms of single predicates, but could also be presented in terms of theories in an appropriate logic. Making theories explicit is advantageous for representing and reasoning about complex abstractions [1]. Also, in contrast to attempting to reuse individual procedures or predicates, theories provide a granularity which better promotes software reuse much as classes do in object-oriented programming.

4.1 Validations are Explanations

A morphism M between data theories T1 and T2 can be used for constructing explanation-based generalization problems. The requirements shown in Figure 1 are met by defining the following correspondences:

Training Example:
The mapping on sorts and operations defined by M.

Domain Theory:
The data theory T2.

Goal Concept:
A morphism M' from T1 to a subtheory T2' of T2

The Operationality Criterion is defined by the requirement that the axioms defining the target theory be expressed in terms of primitive operations of T2.

An explanation is, of course, a proof that M is a morphism. In other words, it is a collection of proofs for the axioms in the source theory when interpreted in the target theory.

4.2 Goal Regression in Equational Theories

Validations of morphisms between the equational data theories described above are collections of proofs that equations corresponding to axioms of one theory follow from the axioms of the other. A simple example of such a proof from the Stack-with-Offset morphism of Figure 4 is the following:

```
assign(a,j,e)[i] = if i=j then e else a[i]
assign(a,n+1,e)[i] = a[i] if not(i=n+1)
assign(a,n+1,e)[i] = a[i] if i<=n
<assign(a,n+1,e),n> = <a,n>
pop(<assign(a,n+1,e),n+1>) = <a,n>
pop(push(e,<a,n>)) = <a,n>
```

Proof steps in equational logic use so-called rules of equational reasoning (reflexive, symmetric, and transitive laws, together with substitution and instantiation) and may also use axioms and inference rules from first-order predicate logic, and reasoning about representing and

distinguishing terms. The text [9] treats equational reasoning for data theories.

Viewing each axiom in the source theory of a morphism as a goal, the equational inference steps transform it into formulas, ultimately in the target theory. This transformation process constitutes *goal regression* of the axiom, and corresponds to the regressing of goal concepts through explanation steps as described in [11], with the rules of equational reasoning explicitly added. The algorithm of [11] for using regression to extract the generalization from an explanation of a goal applies directly here, except formulas for (conjunctive) branches in the proof are collected as distinct axioms for the derived *theory*. Since the validation assures that the regressed goals follow from the original implementing theory, the generalized theory is the subtheory consisting of the regressed goals and the sorts and operations used to express them.

Theories describing learned concepts here are in a sense more "rigid" than simple predicates would be for the same purpose. There is a certain give-and-take in regressing a collection of axioms. While the collection of regressed source axioms (or their conjunction) may be distinct from the original target axioms, the theory they determine may be equivalent. In that case, the effort of regressing the axioms is wasted since the original target axioms could be used just as well. The significance of this distinction between theory and predicate becomes more apparent in observing that *structural* properties of a theory morphism may detect such equivalence. Although it isn't immediately obvious, this is the case, for example, if the operations in the source theory are explicit generators. For example, a stack s can always be written in the form pop(s′) from the axiom pop(push(e,s))=s. It is not true that regressing that axiom over the given proof above would weaken the target axiom

<a,n> = <b,m> if m=n and a[i]=b[i] for i<=n

for the morphism in Figure 4 to

<assign(a,n+1,e),n> = <a,n>

since pop(<a,n>) = <a,n-1> can be applied inductively to recover the original axiom. An example of non-trivial goal regression is given below.

5. Examples

5.1 Stacks as Arrays with Distinguished Index

Figure 4 describes a theory morphism refining the abstract stack data type to that of arrays with a distinguished index, called here the theory of i-arrays. The I-Array theory in Figure 3 imports the sorts and operations of the theories Nat, of natural numbers, and Array. In particular, I-Array uses the operations "assign" from Array and "+" from Nat.

What is interesting about this example is that if the I-Array theory is replaced by any other target in defining this morphism, then the generalization recovers precisely the I-Array theory. In effect, this morphism is the *result* of

```
THEORY I-Array / Nat Array
SORTS i-array
OPS <_,_>: array nat -> i-array
VARS a,b: array
     m,n: nat
EQNS <a,n> = <b,m> if m=n and a[i]=b[i] for i<=n
```

Figure 3. Data Theory of I-Arrays

```
MORPHISM Stack-with-Offset  Stack => I-Array
SORTS (stack => i-array)

VARS e: element
     a: array
     n: nat
OPS (empty => <new-array,0>)
    (push(e,<a,n>) => <assign(a,n+1,e),n+1>)
    (pop(<a,n>) => <a,n-1>)
    (top(<a,n>) => a[n])
    (empty?(<a,n>) => if n=0 then true else false)
```

Figure 4. Stack Implementation Morphism

generalization. In practice, an implementation of stacks is with respect to some richer theory. Although the theories discussed here are not adequate to specify conventional programming languages such as Lisp or C, it is nevertheless helpful to think of the operations of a theory as the primitives for a programming language it defines. The generalization process here is analogous to going from a program in Lisp, say, to a "more abstract" program which could be transformed into another language such as C. This process, however, should not be confused with that of translating between two specific languages. It is better to think in terms of extracting an application from a larger system in which it has been implemented to retarget it to other, possibly smaller systems. Figure 4 prescribes an implementation of stacks in any theory (language) containing pairs of arrays and natural numbers. The recovery of the I-Array theory from a validation of Figure 4 which was discussed above shows that this morphism can't be generalized further.

5.2 Optimizing Functions on Arrays

In this example, integer arrays are implemented as arrays with an extra slot to hold the value of some given function of the array. The particular example[4] with $f(a) = \sum_{m=0} a[m]$ can be generalized to other integer-valued array

4. The upper summation index is suppressed to make the formulas more readable. The sums are over all (finitely many) non-zero array values starting at the lower summation index. The size operation in this theory provides the upper bound in such sums and is used in reasoning about them. $f(a) = \sum_{m=0} a[m]$ should be read $f(a) = \sum_{m=0}^{size(a)} a[m]$

functions. The motivation for this data refinement operation is that, in practice, often when an array instance is created, certain functions may be evaluated repeatedly for it, in which case storing the value as part of the array entity makes sense in the implementation. Needless to say, this is a very concrete case of a general programming strategy.

```
THEORY Integer-Array / Nat Integer
SORTS integer-array
OPS new-array: -> integer-array
    assign: integer-array nat integer -> integer-array
    _[_]: integer-array nat -> integer
    size: integer-array -> nat
    sum: integer-array -> integer
VARS a: integer-array
     j,m,n: nat
     i: integer
EQNS
(1)  assign(a,n,i)[m] = if n=m then i else a[m]
(2)  new-array[n] = 0
(3)  size(new-array) = 0
(4)  size(assign(a,n,i)) = max(size(a),n)
(5)  sum(new-array) = 0
(6)  sum(assign(a,n,i)) = Σ_{j=0} a[j] - a[n] + i
```

Figure 5. Data Theory of Integer-Arrays

```
MORPHISM Array-with-Sum   Integer-Array => Integer-Array
VARS a: array
     n: nat
     i: integer
OPS (a[n] => a[n+1]')
    (size(a) => max(size'(a)-1,0))
    (sum(a) => a[0]')
    (assign(a,n,i) =>
        assign'(assign'(a,n+1,i),0,Σ_{j=0} a[j+1]'- a[n+1]' + i))
```

Figure 6. Caching Sums Morphism

The implementation is given by the morphism shown in Figure 6 which maps the theory Integer-Array (Figure 5) to itself. Note that for convenience in this example, integer arrays are initialized to zero. We haven't bothered bounding the arrays; the size operation just keeps track of the highest index for which a (possibly) non-zero value has been set. In Figure 6, to distinguish operations in the source and target theories, those in the latter are primed, e.g. assign' is the assign operation in the target theory.

A validation of the morphism in Figure 6 consists of proofs for each of the 6 equational axioms for the Integer-Array theory. Given such a validation, the morphism is generalized by constructing a new target theory from those axioms regressed over the proofs. It is not difficult to construct such a validation for which regressions of axioms (1) through (5) essentially reproduce those axioms as a group, except axiom (1) is weakened to

$$\text{assign}'(a,n,i)[n]' = \text{if } n>0 \text{ then if } m=n \text{ then } i \text{ else } a[n]' \qquad (1a)$$

The following formulas show the main steps in a proof of axiom 6 (using (1a)):

$$\text{assign}'(a,0,t)\,[0]' = t$$

$$\text{assign}'(a,0,\textstyle\sum_{m=1} a[m]')\,[0]' = \sum_{m=1} a[m]'$$

$$\text{assign}'(\text{assign}'(a,n+1,i),0,\textstyle\sum_{m=0} a[m+1]' - a[n+1]' + i)[0]' = \sum_{m=0} a[m+1]' - a[n+1]' + i$$

$$\text{assign}(a,n,i)[0]' = \textstyle\sum_{m=0} a[m+1]' - a[n+1]' + i$$

$$\text{sum}(\text{assign}(a,n,i)) = \textstyle\sum_{m=0} a[m] - a[n] + i\,.$$

For this derivation, the regressed goal is then

$$\text{assign}'(\text{assign}'(a,n+1,i),0,\textstyle\sum_{m=0} a[m+1]' - a[n+1]' + i)[0]' = \sum_{m=0} a[m+1]' - a[n+1]' + i\,.$$

Using (1a), this reduces to

$$\text{assign}'(a,0,\textstyle\sum_{m=1} a[m]')\,[0]' = \sum_{m=1} a[m]' \qquad (1b).$$

The generalized theory is gotten by replacing the axiom (1) with (1a) together with the weaker regressed goal (1b). In that theory, the 0 array slot is required to behave in the usual way only if it contains the sum of the other array values. This generalization allows many alternative specializations, for example,

$$\text{assign}(a,0,i)[n] = \text{if } n=0 \text{ then } \textstyle\sum_{m=1} a[m] \text{ else } a[n] \qquad (1c)$$

which leads to an array-like data type in which, however, assignments to slot 0 are ignored and slot 0 always evaluates to the sum of the other slots, or

$$\text{assign}(a,0,i)[n] = \text{if } n=0 \text{ then } \max(i, \textstyle\sum_{m=1} a[m]) \text{ else } a[n] \qquad (1d)$$

in which assignments are effective only for values greater than the sum of the other slots. These special types are reminiscent of structures which occur especially in hardware optimizations in which the behavior of special registers or memory addresses may differ in similar ways from ordinary array slots. In this case, explanation-based generalization yields an abstraction which can suggest more efficient alternative specializations.

6. Learning Apprentice Systems

Learning apprentice systems were introduced in [10]. A learning apprentice is a knowledge-based interactive system which assists a user working in a problem-solving domain. It acquires new knowledge in the course of normal use by capturing and generalizing the problem-solving steps carried out by the user. The system recycles this

knowledge by using it to make suggestions in subsequent problem-solving situations.

The objective of learning apprentice systems is to reduce the knowledge acquisition bottleneck by minimizing the separate investment of effort by experts and knowledge engineers in creating and maintaining the knowledge-base. Instead, it provides a mechanism for adding to the knowledge base automatically through expert use. In the domain of software development, programming languages and software libraries typically constitute invested expert knowledge. As discussed below, a learning apprentice for software development is a programming environment which makes components of user programs, as they are created, effectively reusable.

A particular learning apprentice system called LEAP is described in [10]. LEAP applies explanation-based learning to VLSI design. It suggests structural decompositions of VLSI circuits described by functional specifications. The LEAP user may accept its advice or else construct an alternative refinement. In the latter case, LEAP will attempt to validate the user's solution and use the validation (explanation) to generalize it.

LEAP's knowledge is in the form of rules: IF a circuit performs a certain function (e.g., conversion of a serial signal to parallel), THEN it may be implemented using a particular network (e.g., a shift register). LEAP's generalization component takes a particular implementation and abstracts both sides of the rule.

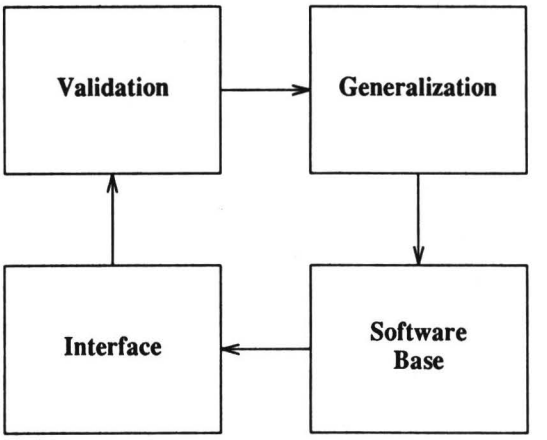
Current work on learning apprentice systems centers on applying explanation-based learning. That approach is especially appropriate for applications to programming since domain theories are inherent in software, as we've discussed in this paper. Nevertheless, there is every reason to expect future learning apprentice systems to benefit from and in fact stimulate research in other machine learning paradigms.

The knowledge acquisition bottleneck addressed by learning apprentice systems has a direct counterpart in software engineering. Programming expertise is invested, for example, in developing component libraries (functions, procedures, modules, classes, etc., depending on the programming paradigm). The selection and creation of library components to promote reuse require considerable effort which is distinct from the primary problem-solving activity of programming. A learning apprentice system for programming should promote reuse of "live" code. In other words, it should take over responsibility for generalizing and validating program components and cataloging them.

The use of software validation is especially interesting here. It is generally recognized that validation is important for the reliability of reusable code. In this application, the use of validations to make particular software more generic adds a new dimension and helps offset the additional cost of validation.

7. LASR: A Learning Apprentice for Software Reuse

LASR is an experimental learning apprentice system which applies explanation-based learning to abstract data types using the principles developed in this paper. It brings together essential ideas from [4] and [10]. Its primary purpose is for use in testing the reuse potential of the interconnections derived from explanation-based generalization described in this paper and for experimenting with programming methodologies which promote such reuse. LASR is being developed currently at Hewlett Packard. The following diagram gives an overview of the components of LASR which deal explicitly with reuse.



The LASR interface allows the user to specify a module to be refined. The system first attempts to match the specification against possible refinements in its software base. The software base consists of *views* (essentially theory morphisms) which are represented as pairs of data theories together with a mapping from sorts and operations of one to the other, and a proof that the mapping is valid, i.e., that it preserves the properties of the operations. Views are retrieved from the software base using consistent matching of terms in the view's source specification with the specification provided by the user. The view's source specification may contain more operations and axioms than are used in the matching. For example, if the software base contains an implementation of deques (two-ended stacks), that implementation could be suggested for implementing stacks. This corresponds at least in part to generalizing the left-hand side of LEAP's rules and to the notion of implementation inheritance in [13].

If no suitable match for the specification is found, or if the proposed implementations are rejected, then the user refines the specification directly. The system attempts to validate the refinement step using an equational theorem prover. If a validation is obtained, then it is used to generalize the view provided by the refinement. It is the generalization, of course, which is added to the software base.

8. Future Research

Along with using LASR to evaluate the benefits of learning apprentice systems for practical software engineering, there are a number of ways in which the methods described here may be fruitfully extended. For example, equational theories have important technical limitations to their use in specifying software systems. There are other logical systems with different properties such as Horn-clause logic, temporal logic and process logic which, like equational logic, may be used for representing and reasoning about programs. The notion of an *institution* [5] gives a common framework for treating such systems. It is desirable to carry over the application of explanation-based methods and goal regression for equational theories to theories in other institutions[5] and to seek a more intrinsic characterization[6] of the generalized objects obtained from their application. Also the notion of morphism discussed here inhibits transformations on theories which are well motivated in practice [12]. It is important to determine how the methods presented here can be applied to that programming knowledge as well. In a different direction, it is quite natural to ask how reuse based on the idea here of weakening the specification of the base theory in an implementation is related to reuse using other forms of generalization, e.g. through parametrization or inheritance.

9. Conclusions

The application of explanation-based learning presented in this paper provides a new approach to generalizing software developed in practice. It depends on and complements other research into the problem of software reuse based on formal methods for program specification. The abstract data types synthesized using explanation-based generalization provide minimal requirements for the valid reuse of interconnections between software components. At the same time, they can be used directly in modifying program derivations.

The creation of reusable software components constitutes a knowledge acquisition bottleneck for which a learning apprentice system can prove valuable. The results of this paper provide the basis for developing such a system as we are now doing (LASR).

The adaptation of explanation-based generalization is framed in terms of *theories*. Theories provide a granularity more suitable for application to software reuse than single goal predicates do and are more suitable for representing and reasoning about complex abstractions. They may also prove valuable in other applications of explanation-based learning.

---

5. The work reported in [15-17] suggests in particular applications involving non-determinism, and has been applied to deriving a minimal lowest layer of a multi-layered communications network architecture [18].

6. Intuitively, explanation-based generalization here can be thought of as the adjoint to the functor which maps validated morphisms to their validation.

Acknowledgements

I would like to thank Tom Mitchell for introducing me to explanation-based learning and learning apprentice systems, and especially for a discussion which led to this work. Discussions with C.A.R. Hoare and He Jifeng were very helpful for understanding their notion of weakest prespecification in data refinement; they also suggested the interpretation of generalizations in terms of adjoint functors. I am very grateful to Alan Snyder, Martin Griss, and Ira Goldstein of HP Labs for enabling me to pursue this research.

References

[1] Burstall, R.M., Goguen, J.A. Putting Theories Together to Make Specifications. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 1045-1058

[2] Cohen, B., Sammut, C. Program Synthesis through Concept Learning. In A. Bierman, G. Guiho, Y. Kodratoff (Eds.) *Automatic Program Construction Techniques* New York: Macmillan, 1984

[3] DeJong, G., Mooney, R. Explanation-Based Learning: An Alternative View. *Machine Learning*, Vol. 1, No. 2, 1986, pp. 145-176

[4] Goguen, J.A. Reusing and Interconnecting Software Components. *IEEE Computer*, Vol. 19, No. 2, 1986, pp. 16-28.

[5] Goguen, J.A., Burstall, R.M. Introducing Institutions. In E. Clark, D. Kozen (Eds.) *Proc. Logics of Programming Workshop. Lecture Notes in Computer Science*, Vol. 164, Springer-Verlag, 1984, pp. 221-256

[6] Goguen, J.A., Tardo, J. An Introduction to OBJ: A Language for Writing and Testing Software Specifications. In *Specification of Reliable Software*, IEEE, 1979

[7] Gries, D. *Science of Programming*. New York: Springer. 1981.

[8] Guttag, J.V., Horning, J.J., Wing, J.M. Larch in Five Easy Pieces. *Technical Report* 5, Digital Equipment Corp. - Systems Research Center, 1985

[9] Liskov, B., Guttag, J. *Abstraction and Specification in Program Development* Cambridge, Mass.: MIT Press, 1986, Chapter 10

[10] Mitchell, T.M., Utgoff, P.E., Banjeri, R.B. LEAP: A Learning Apprentice for VLSI Design. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985, pp. 573-580

[11] Mitchell, T.M., Keller, R.M., Kedar-Cabelli, S.T. Explanation-Based Generalization: A Unifying View. *Machine Learning*, Vol. 1, No. 1, 1986, pp. 47-80

[12] Scherlis, W.L. Abstract Data Types, Specialization, and Program Reuse. (Preliminary Version) *IFIP International Workshop on Advanced Programming Environments*, 1986

[13] Snyder, A. Encapsulation and Inheritance in Object-Oriented Programming Languages. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM, 1986, pp. 38-45

[14] Waldinger, R. Achieving several goals simultaneously. In E.Elcock, D. Michie (Eds.), *Machine Intelligence* 8, 1977 Also in B.L. Webber, N.J. Nilsson (Eds.), *Readings in Artificial Intelligence*.

[15] Hoare, C.A.R., He, J. The Weakest Prespecification. *Fundamenta Informaticae* IX, 1986, pp. 51-84, 217-252

[16] Hoare, C.A.R., He, J., Sanders, J. W. Prespecification in Data Refinement. Oxford University preprint, November, 1986

[17] He, J., Hoare, C.A.R., Sanders, J. W. Data Refinement Refined. Report of Programming Research Group, Oxford, February, 1987

[18] He, J. Personal communication, April, 1987