AUTHOR Neff, Norman D.
TITLE A Logic Programming Testbed for Inductive Thought and Specification.
PUB DATE 95
NOTE 5p.; In: "Emerging Technologies, Lifelong Learning, NECC '95"; see IR 017 705.
PUB TYPE Reports - Descriptive (141) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Classroom Techniques; *Computer Science Education; Higher Education; *Induction; Mathematics Education; *Programming Languages; Thinking Skills
IDENTIFIERS Inferential Comprehension

ABSTRACT

This paper describes applications of logic programming technology to the teaching of the inductive method in computer science and mathematics. It discusses the nature of inductive thought and its place in those fields of inquiry, arguing that a complete logic programming system for supporting inductive inference is not only feasible but necessary. A sample dialog from the Prologb system is included, along with an overview of the Prologb language and some details about classroom experiences using the system. (Author/BEW)

A Logic Programming Testbed for Inductive Thought and Specification

by Norman D. Neff

Paper presented at the NECC '95, the Annual National Educational Computing Conference (16th, Baltimore, MD, June 17-19, 1995.

paper

A Logic Programming Testbed for Inductive Thought and Specification

Norman D. Neff
Department of Computer Science
Trenton State College CN 4700
Trenton, NJ 08650-4700
(609) 771-2482
neff@trenton.edu

Key words: inductive inference, logic programming

Abstract

We describe applications of logic programming technology to the teaching of the inductive method in computer science and mathematics. Inductive inference is used in the sense of reasoning from sets of specific examples to plausible general explanations. The paper treats the feasibility of supporting inductive inference through logic programming technology, and argues that a complete logic programming system is required. We include a sample dialog, and discuss our classroom experience using the system.

Introduction

The paper describes applications of logic programming technology to the teaching of the inductive method in computer science and mathematics. Section 2 is a general discussion of the nature of inductive thought and its place in computer science and education. In Section 3 treats the feasibility of supporting inductive thought through logic programming technology, and argues that a complete logic programming system is required. Section 4 is a sample dialog illustrating the Prologb system. An overview of the Prologb language is provided in Section 5. Section 6 covers our experience using the system.

Inductive thought

The inductive method is the primary methodology of the physical sciences. The inductive method comprises two major activities. When a phenomenon is encountered, a set of particular specimens is examined, and a conjectured general pattern is formulated. The conjecture is then tested by attempting to make predictions based on it, and evaluating the reasonableness of consequences of the conjecture. If the conjecture fails any test, the conjecture is modified to encompass the new test observation. As the conjecture passes an increasing number of tests, its subjective likelihood of correctness increases. When the likelihood of correctness exceeds a certain threshold, the conjecture becomes a serious theory or model. In the physical sciences, this means that the theory is accepted, until evidence contradicting it is found, or until it is replaced by a more powerful theory that successfully explains even more observations.

Mathematics, especially higher mathematics, is often regarded as a demonstrative, rather than an inductive, science. Early in our mathematical education, we experience Euclid's axiomatic development. Geometry is presented as an exercise in pure logic, in which a small set of axioms is progressively expanded to a complete body of theory. In fact, Euclid's work was the synthesis of centuries of previous inductive, experimental geometrical discovery [2]. Polya [4] argues that the inductive method is also a vital component of modern mathematics. New conjectures are spawned by observation of specific examples, and by the mental processes of generalization, specialization, and analogy. Conjectures are then tested, resulting either in rejection of the conjectures and formation of modified conjectures, or in increasing confidence in the conjectures. In the mathematical context, the final test of a conjecture is the creation of a formal deductive proof.

Computer science is also subject to Polya's observations, because computer science is a mathematical science. Rigorous mathematical theories underlie many significant areas of computer science, such as: algorithm analysis, artificial intelligence, formal languages, graphics, language design, and database design.

With reference to "ordinary programming," there is some controversy as to the role of verification by rigorous mathematical analysis. Computer programs are typically developed through a highly inductive process of interleaved stages of design, coding, and testing. Programs are expected to always perform correctly, for all possible inputs. Since the number of possible inputs generally far exceeds the feasible number of test runs, no program's correctness can be guaranteed by testing, unless it can be argued that the chosen suite of tests will uncover all errors. At any rate, it is clear that inductive thought is prominent in programming, and that it must be supplemented, as in mathematics, by some final analytical stage.

Logic programming as a testbed for inductive thought

The ideal testbed for inductive thought would present the user with a series of specific examples. After each example, the user would have the opportunity to review the set of examples, and to form conjectures and predictions about the observed process. At any time, the user should have the ability to test any of his predictions. Finally, the user should be able to fully specify a proposed model for the process, and to observe the examples generated by her proposed model. So that many examples can be efficiently covered, the instructor should be able to quickly configure the testbed to exhibit desired behavior. Such a testbed could be used as a basis for discovery lectures, and for laboratory experiences.

A candidate for the basis for the testbed is the logic programming language Prolog. The Prolog program is a database containing logical clauses defining predicates. The Prolog system then acts as an acceptor/generator for the predicates. For example, the clauses might define the predicate orderedList(L), meaning that L is an ordered list of integers, such as [1,3,4,6]. To generate ordered lists, one gives the system the input query consisting of the predicate symbol applied to an uppercase logical variable, e.g. orderedList(X). The system then generates ordered lists one by one, as a byproduct of a theorem proving algorithm. The system can also be used as an acceptor, whereby it is given the query consisting of the predicate symbol applied to a specific object, e.g. orderedList([1,3,4,6]). The response to this query is "yes", or "solution found". A remarkable property of logic programming is that both behaviors, generator and acceptor, arise from one and the same logical database.

Unfortunately, the Prolog system is not adequate as a general inductive testbed. For technical reasons, Prolog uses a search strategy that is incomplete, meaning that it may fail to generate some examples, and may fail to accept certain correct examples. Correct use of Prolog system requires a certain amount of detailed knowledge of the Prolog search method. For this reason, Prolog use is generally confined to specialized courses in which the study of the search method can be justified.

We have implemented an alternative, complete, logic programming system, known as Prologb. In Prologb, all examples are generated, and all correct examples are accepted. When the system is given an incorrect example, it may reject the example or nonterminate. A theoretical result due to Church[1], stating that the general predicate calculus is algorithmically undecidable, implies that this nontermination problem is inherent in all computer-based systems that encompass predicate logic.

Using Prologb for Inductive Discovery: An Example

The following short sample dialog gives the flavor of the student interaction occurring as the system is used. In practice, more complicated situations can be explored by longer dialogs.

By running two independent copies of the acceptor/generator, the generation sequence in one window may be supplemented at will by tests in a second window.

```
1. The system asks for a query:
     Query? (or command) :

2. To generate examples, we enter
     example(X)

The successive examples generated are
     X = nil

 3. (Comment) the empty list
     X = [a]
     X = [b]

4. (Comment) all lists of length 1 or less, using objects a and b
     X = [a]     X = [a,b]    X = [b,a]    X = [b,b]

5. (Comment) all lists of length 2 or less, using objects a and b, repeats allowed
     X = [b,b,a] X = [b,b,a]  X = [a,a,a]  X = [b,a,a]  X = [a,a,b]
     X = [b,a,b] X = [a,a,a]  X = [b,a,a]  X = [a,b,b]  X = [b,b,b]

6. (Comment) 7 of the 8 possible lists of length three or less: missing is [a,b,a]
     X   [b,b,a,a]  X    [b,b,a,b]  X = [b,b,a,a]  X = [a,b,b,a]
     ...

7. (CONJECTURE) No list contains a,b,a as a sublist
8. Testing [b,a,b,a],
     Query? (or command) :
     example([b,a,b,a])
     no solution

9. Testing [b,b,b,a,b,b],
     Query? (or command) :
     example([b,b,b,a,b,b])
     Solution found:
```

The Prologb language

The syntax of the Prologb language is that of a subset of Prolog. For an overview, we refer to Figure 1, which contains a simple Prologb program exhibiting the behavior seen in the previous section. The program is an acceptor/generator for the set of all lists over {a,b} that do not contain the sublist [a,b,a].

```
%aba not allowed

alpha(a).  alpha(b).

example([]).
example([a]).
example([b]).
example([A,B]):-alpha(A),alpha(B).

example([A,a|X] ):-example([a|X]),  alpha(A).
example([b,b,a|X]):-example([b,a|X]).
example([A,b,b|X]):-example([b,b|X]),alpha(A).
```

Figure 1

The first line, starting with the symbol "%", is a comment line. The next nonblank line contains two facts stating that the symbols "a" and "b" are the two objects satisfying the predicate alpha, that is, they are the two symbols in the alphabet. The next three lines are facts stating that [],[a], and [b] satisfy the predicate example. The fourth line

example([A,B]):-alpha(A),alpha(B).

is a *rule* containing the logical connective ":-" , which means if, and, on the right hand side, the connective "," which means *and*. The identifiers starting with uppercase letters are universally quantified logical variables. Thus, the meaning of the line is "For all A and B, the list [A,B] is an example if A is in the alphabet and B is in the alphabet". The remaining rules express constraints on longer lists so that the sublist [a,b,a] may never appear. For example, the line

example([b,b,a | X]):-example([b,a | X]).

expresses the idea that any example list beginning with "b,a" may be expanded to another example list by adding a "b" at the front. Since [a,b,a] sublists are to be avoided, there is no rule permitting the addition of "a" at the front in such cases.

Experience using Prologb

We have used Prologb in several introductory and advanced mathematics and computer science courses, including Discrete Mathematics, Compilers and Interpreters, and a new course, Discrete Structures of Computer Science, which explores structural concepts in the context of exploratory declarative programming. The Discrete Structures course was one of three courses developed for a new curriculum[1] that integrates mathematics with computer science, starting with entering students' first course and laboratory experiences [5,6].

Prologb courseware has been developed in several problem domains, including formal language theory, modeling automata, semantic nets, general relations, graphs and digraphs, Peano arithmetic, and binary and modular arithmetic.

Online computer access to Prologb permits instructors to incorporate induction into classroom discussions. In general, classroom demonstrations often elicit many conjectures from the group. In the inductive context, incorrect conjectures are not "wrong answers"; they are a normal part of the inductive process. Every incorrect conjecture leads to an exploration of its consequences, and that exploration then leads to a better conjecture.

The Prologb system is also a vehicle for laboratory experiences in computer science and mathematics courses. Examination of the lab reports indicates that beginning students were able to handle simple inductive exercises, in which students are asked to enter various inputs into some prepared environment, and to then predict/explain the resulting output. In most cases, students were also able to handle programming assignments, in which the goal was the creation of a new Prologb program produce a desired result. The students were able to quickly assimilate Prologb because it is a language that simply and faithfully executes logical specifications.

References

[1] Church, Alonzo. A note on the Entsheidungsproblem. J. Symbolic Logic (1), 1936.
[2] Kneebone, G.T. Mathematical Logic and the Foundations of Mathematics. p 134. D. Van Nostrand, London.
[3] Neff, Norman. A Logic Programming Environment for Teaching Mathematical Concepts of Computer Science. SIGCSE Bulletin 25 (1), March 1993.
[4] Polya, George, Induction and Analogy in Mathematics, v.1 of Mathematics and Plausible Reasoning, Princeton University Press, Princeton, N.J., 1954.
[5] Wolz, Ursula and Conjura, Edward. Integrating Mathematics and Programming Into a Three Tiered Model For Computer Science Education. SIGCSE Bulletin 26(1), March 1994.
[6] Wolz, Ursula and Conjura, Edward. Abstraction To Implementation: A Two Stage Introduction to Computer Science. NECC Proceedings '94

[1] Supported by National Science Foundation Grant DUE-9254108.