A Pragmatic Approach to Resolution-Based Theorem Proving

Judy A. Townley

Received April 1979; revised November 1979

Resolution theory offers a simple, complete method for proving theorems but is generally considered impractical. The theorems we are interested in proving arise in the analysis of programs and usually involve quantification. We have developed a system for proving these theorems using resolution, but have embedded in it a simplifier as the central component. The simplifier is an integrated collection of algorithms for normalizing arithmetic, relational, and logical expressions. The knowledge in the simplifier is encoded in procedures, rather than as axioms or rules. We use the simplifier to prove certain theorems, reduce the clutter in theorems, and reduce the cost of unification. Inherent in the normal form algorithms is the notion of strengthening (e.g., inferring $a = b$ from $a < b$ AND $b < a$). We have incorporated the notion into the unification algorithm as well. The design of the system permits its use along a spectrum from pure resolution to resolution with interpretation of the arithmetic and relational operators. Strengthening is a heuristic that permits the movement along this spectrum. We call the approach $i$-resolution. $i$-resolution does not preserve completeness; it does define a means for approaching completeness efficiently and systematically. It thus attempts to provide a pragmatic approach to mechanical theorem proving.

KEY WORDS: Completeness; inference rules; resolution; simplifiers; theorem proving; unification; verification.

1. INTRODUCTION

Much of the current research in advanced programming systems is revealing new opportunities for—indeed a greater reliance upon—mechanical theorem proving. The direction of these advances indicates a need for a systematic,
yet practical, method for proving theorems. The results described below attempt to meet this need.

In Sec. 1 we describe the motivation for our work and the types of problems to which we hope the results will be relevant. A brief overview of resolution and other theorem proving systems is provided in Sec. 2. In Sec. 3 we discuss the theorem proving system we have developed, discuss its instantiation mechanisms, and remark on its completeness. We have applied the theorem prover to several examples considered difficult to prove mechanically. The results of these experiments are described in Sec. 4. We outline in Sec. 5 the direction of our future work based on the theorem prover.

The optimization of programs, their verification, the detection of errors, and the validation of integrity and protection constraints all require some form of theorem proving. In some systems restrictions are placed on the language or limitations are placed on the applicability of the program processor to avoid the need for a theorem prover. These constraints, however, significantly decrease the generality of the resulting software.

Consider, for example, a source-to-source optimizer that includes common subexpression elimination among its potential transformations. With the assistance of a theorem prover the optimizer could eliminate program expressions that do not necessarily look similar but can be proven to compute the same result. Without a theorem prover the optimizer is likely to detect only syntactically similar expressions. In general a program optimizer must ensure that certain enabling conditions are met before applying a transformation, and that the expected functionality of the program is preserved if the transformation is made. Both involve proving theorems.

Program verification also involves proving theorems, though various verification systems\textsuperscript{5,10–12,15,17,22} place a varying amount of importance on the theorem prover. Most use a natural deduction, or goal-oriented, proof procedure (about which more is said in Sec. 2.3) rather than resolution. Various heuristics guide the incorporation of deduction rules and the instantiation of quantified variables. Special algorithms are required to handle existential quantifiers in those systems that permit them. A more rigorous inference system and a more systematic approach to instantiating axioms would facilitate the integration of extensions and provide greater power to a verification system.

Recent work on the development of programming environments has demonstrated the usefulness of various other preruntime processors. These include facilities to detect obvious program errors, such as subscripts out of range, improper arguments appearing in a procedure call, and the use of undefined variables.\textsuperscript{6,8,9,14,18} The addition of a theorem prover and the ability to prove simple theorems would significantly enhance the power of these processors as well.