AUTOMATIC PROGRAM ANNOTATION

Nachum Dershowitz
Stanford Artificial Intelligence Laboratory
Stanford, California 94305

ABSTRACT

Techniques were developed by which an Algol-like program, given together with its specifications, may be documented automatically. This documentation expresses invariant relationships that hold between program variables at intermediate points in the program, and explains the actual workings of the program regardless of whether the program is correct. These techniques, formulated as deduction rules for both guaranteed invariants and candidate invariants, represent a unification of existing approaches, and sometimes improve upon them.

INTRODUCTION

Program annotation is the task of discovering a set of *invariant assertions* and documenting a program with them. These invariants describe the workings of the program, independent of its correctness or incorrectness, by detailing relationships between the different variables at specific points. Not only are invariants necessary for the verification of a program by the invariant-assertion technique, but they may also be used to prove termination or incorrectness of a program, to analyze the efficiency of the program, aid in the optimization of the program, and guide program debugging and modification.

This research is aimed at unifying previous approaches to annotation (*e.g.,* Wegbreit [1974] and Katz and Manna [1976]) and extending some of the earlier techniques. The methods developed have been implemented in QLISP and a catalog of annotation rules has been compiled.

THE RULES

The annotation techniques are expressed as rules, each rule consisting of a set of antecedents and a consequent. There are two basic types of rules: algorithmic and heuristic.

- *Algorithmic rules* derive invariant relations between variables directly from the program statements. These rules may be further subdivided into two categories:
  - *assignment rules,* which yield *global invariants* that hold at all labels in the program, based only upon the assignment statements of the program;
  - *control rules,* which yield *local invariants* associated with specific points in the program, based upon the control structure of the program.
- *Heuristic rules* have *candidate invariants* as their consequents. These candidates, though promising, are not guaranteed to be invariants and must be checked by a verifier. Initially all programmer supplied assertions are candidates.

The assignment rules relate to particular operators occurring in the assignment statements of the program. For example, there are three rules for multiplication: a *multiplication rule,* which gives the range of variables which are updated by multiplication by a constant expression; a *set multiplication rule* for the case where a variable is multiplied by other variables whose ranges are already known; and a *multiplication relation rule* which relates variables that are always multiplied by similar expressions. Corresponding rules are formulated for other operators.

Consider the *multiplication rule.*

$$\frac{P:\ u := u0 \mid u \cdot u1 \mid u \cdot u2 \mid \ldots}{P:\ \{\ u \in u0 \cdot u1^{N} \cdot u2^{N} \cdot \ldots\ \}}$$

The antecedent states that the *only* assignments to the variable u in the program P are $u := u0$, $u := u \cdot u1$, $u := u \cdot u2$, etc., where the expressions $u0$, $u1$, $u2$, ... are of constant value within P. When this condition holds, the global invariant $u \in u0 \cdot u1^{N} \cdot u2^{N} \cdot \ldots$ holds throughout execution of P. That is, once u has been assigned a value in P, it belongs to the set $u0 \cdot u1^{N} \cdot u2^{N} \cdot \ldots$, where N is the set of natural numbers and for any expression α, $\alpha(S1, S2, \ldots, Sm)$ denotes the set of elements $\alpha(s1, s2, \ldots, sm)$ such that $s1 \in S1$, $s2 \in S2$, ..., $sm \in Sm$. From such an invariant, specific properties, such as a bound on u, may be derived. Note that no restrictions are placed on the order in which the assignments to u are executed.

An assignment $u := \alpha(s)$, where it is known that $s \in S$, may be viewed as the nondeterministic assignment $u :\in \alpha(S)$ of an arbitrary element of $\alpha(S)$ to u. Thus the above rule might be considered as a special case of a more general *set multiplication rule.*

$$\frac{P:\ u :\in S0 \mid u \cdot S1 \mid u \cdot S2 \mid \ldots}{P:\ \{\ u \in S0 \cdot \Pi S1 \cdot \Pi S2 \cdot \ldots\ \}}$$

where ΠS denotes the closure of S under multiplication, *i.e.,* the set of products of elements of S. If S contains the single element s, then $\Pi S = s^{N}$.

The following *multiplication relation rule* relates different variables appearing in the program:

$$\frac{P:\ (u,v) := (u0,v0) \mid (u \cdot \alpha^{u1}, v \cdot \alpha^{v1}) \mid (u \cdot \beta^{u1}, v \cdot \beta^{v1}) \mid \ldots}{P:\ \{\ u^{v1} \cdot v0^{u1} = v^{u1} \cdot u0^{v1}\ \}}$$

where α, β, ..., are *arbitrary* (not necessarily constant) expressions. The simultaneous assignments in the antecedent of the rule may represent individual assignments executed along the same program path, such that whenever, for example, $u := u \cdot \alpha^{u1}$ is executed, so $v := v \cdot \alpha^{v1}$ for the *same* value of the expression α.

While the previous rules completely ignored the control structure of the program, the *control rules* derive important local invariants by taking the program's structure into consideration. For example, the rule

$$\frac{\{\ A\ \}\ ;\ \textbf{loop}\ L:\ ;\ S\ ;\ \{\ B\ \}\ \textbf{repeat}}{L:\ \{\ A \vee B\ \}}$$

derives a loop invariant from the known invariants preceding the loop and following the loop body. It states that when control is at the head L of the loop-body, either the loop has just been entered, in which case the invariant A holds, or the loop-body has just been executed and B holds.

In contrast to the above rules, which are algorithmic in the sense that they derive relations that are *guaranteed* to be invariants, the other class of rules, *heuristic rules,* can only *suggest* candidate invariants. An example is the following *generalization heuristic,* which is valuable for programs with universally quantified output specifications. Given a loop invariant $B(k)$ at label L, where k (a variable or expression) counts the number of loop iterations, we have as a candidate for a loop invariant that B holds for previous values of k as well. That is,

$$\frac{\{\ k = 0\ \}\ ;\ \textbf{loop}\ L:\ \{\ B(k)\ \}\ ;\ S\ ;\ \{\ k = k'+1\ \}\ \textbf{repeat}}{L:\ \{?\ B[0:k]\ ?\}}$$

where k is the value of k when last at L and $B[0:k]$ is short for "B holds for all values between 0 and k". The "{?" and "?}" indicate that the consequent is only a candidate which must be verified. Another example is the *top-down heuristic*

$$\frac{\textbf{loop}\ S1\ ;\ L:\ \{\ A\ \}\ ;\ \textbf{until}\ B\ ;\ S2\ \textbf{repeat}\ ;\ \{?\ C\ ?\}}{L:\ \{?\ A \wedge B \supset C\ ?\}}$$

which may be used to push candidates backwards into a loop.

REFERENCES

Katz, S.M. and Z. Manna [Apr. 1976], *Logical analysis of programs,* CACM, Vol. 19, No. 4, pp. 188-206.

Wegbreit, B. [Feb. 1974], *The synthesis of loop predicates,* CACM, Vol. 17, No. 2, pp. 102-112.